Funktionale Programmierung

Rekursion

Programm

Übersicht

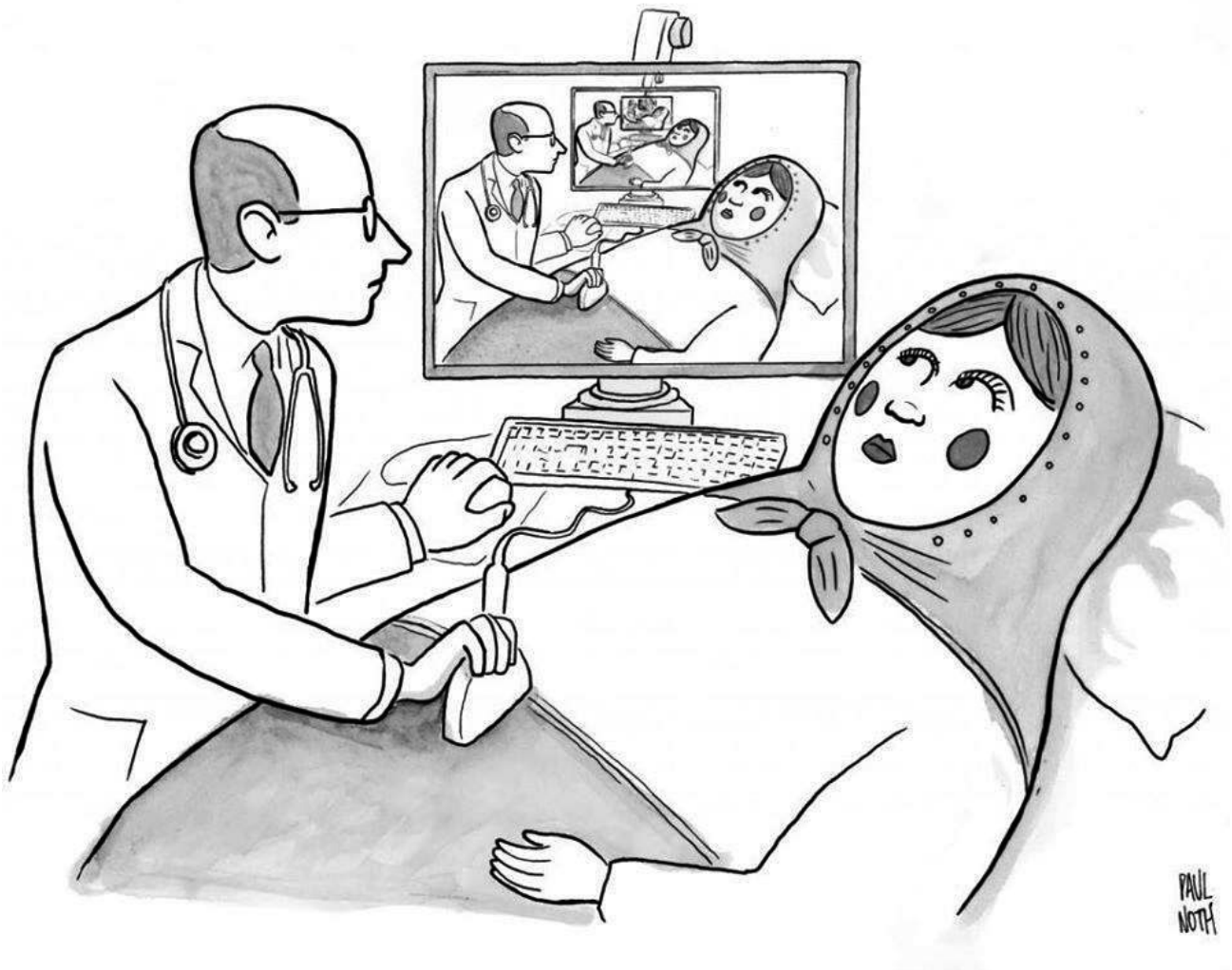
Woche	Thema	Praktika
8	Organisatorisches, historisches, Begriff der funktionalen Programmierung, Einführung F#	1
9	Funktionen und Typen I: Werte, Unions (Listen), Produkte	2
10	Funktionen und Typen II: Options, Funktionstyp, “partial application”, Currying	2,3
11	Funktionen und Typen III: Kombinatoren und höhere Funktionen	3
12	Rekursion I: Formen der Rekursion	4
13	Rekursion II: Fixpunkte	4,5
14	Rekursion III: Rekursion und Compiler Optimierungen	5

Programm

Details

- Formen der Rekursion
  - Primitive Rekursion
  - Wertverlaufsrekursion
  - Strukturelle Rekursion
  - Rekursiver Malwettbewerb
- Fixpunkte
  - Fixpunkte - Konstruktion und Fixpunktsatz
  - Fixpunktkombinatoren
  - Zaubern mit dem Fixpunktkombinator
- Rekursion und Compiler - Optimierungen (Endrekursion)
  - Akkumulator - Pattern
  - Continuation - Pattern

Formen der Rekursion



Rekursive Definitionen zeichnen sich durch die Bezugnahme auf das zu definierende Objekt (in einer einfacheren Form) aus.

$$2^0 = 1$$

$$\underbrace{2^{n+1}}_{\text{definiere}} = 2 \cdot \underbrace{2^n}_{\text{Selbstbezug}}$$

Eine rekursive Definition einer Funktion "ist" ein Algorithmus, der aus bereits vorhandenen Funktionswerten weitere Funktionswerte (der dadurch definierten Funktion) generiert.

$2^0 = 1$	Der Funktionswert an der Stelle 0 ist bekannt.
$2^{n+1} = 2 \cdot 2^n$	Durch das Verdoppeln eines bekannten Funktionswertes erhält man den "nächsten" Funktionswert.

Die Auswertung einer rekursiv definierten Funktion erfolgt “rückwärts”:

$$
\begin{aligned}
2^3 &= 2^{2+1} \\
&= 2 \cdot 2^2 \\
&= 2 \cdot 2^{1+1} \\
&= 2 \cdot (2 \cdot 2^1) \\
&= 2 \cdot (2 \cdot 2^{0+1}) \\
&= 2 \cdot (2 \cdot (2 \cdot 2^0)) \\
&= 2 \cdot (2 \cdot (2 \cdot 1)) \leftarrow \text{ bekannter Wert eingesetzt} \\
&= 2 \cdot (2 \cdot 2) \\
&= 2 \cdot 4 \\
&= 8
\end{aligned}
$$

Wesentliche Zutaten einer rekursiven Definition[1]:

- Bekannte Funktionswerte
- Funktion G zum Erweitern der Menge der bekannten Werte

$2^0 = 1$	bekannter Wert
$2^{n+1} = 2 \cdot 2^n$	$G(x) = 2 \cdot x$

[1] Wie wir sie bis jetzt kennen.

Dies entspricht im Allgemeinen dem einfachsten Rekursionsschema:

$$f(0) = c$$
$$f(n+1) = G(f(n))$$

wobei

- c eine Konstante
- G eine Funktion (ein Algorithmus)

Mit zusätzlichen Parametern ergibt sich daraus das sogenannte Schema der "primitiven Rekursion":

$$f(0, \vec{x}) = c(\vec{x})$$
$$f(n+1, \vec{x}) = G(f(n, \vec{x}), n, \vec{x})$$

wobei

- c eine Funktion (in den Variablen $\vec{x}$)
- G eine Funktion (ein Algorithmus)

Die zusätzlichen Parameter erlauben etwas mehr Freiheit, beispielsweise zum Definieren der allgemeinen Exponentialfunktion:

$$x^0 = 1 \qquad c(x) = 1$$
$$x^{n+1} = x \cdot x^n \qquad G(a, x) = x \cdot a$$

Manchmal, zum Beispiel bei der Definition der Fakultätsfunktion, erweist sich auch der Parameter n als nützlich:

$$0! = 1$$
$$n+1! = (n+1) \cdot n!$$

$$c = 1$$
$$G(a, n) = (n+1) \cdot a$$

Aufgabe

Implementieren Sie das Schema der primitiven Rekursion als höhere Funktion.

```
let rec primRec g c n x = ...
```

Aufgabe

Implementieren Sie das Schema der primitiven Rekursion als höhere Funktion.

```
let rec primRec g c n x =
    if n = 0 then c x
    else g (primRec g c (n - 1) x) n x
```

Manchmal wird in rekursiven Definitionen nicht bloss auf einen, sondern auf mehrere "Vorgänger" Bezug genommen:

$$
\begin{aligned}
fib(0) &= 0 \\
fib(1) &= 1 \\
fib(n) &= fib(n-1) + fib(n-2)
\end{aligned}
$$

Dies stellt einen Spezialfall der "Wertverlaufsrekursion", worin auf den gesamten Wertverlauf (bis zu einem Punkt) der zu definierenden Funktion Bezug genommen werden kann, dar:

$$f(n) = G(f \upharpoonright n)$$

oder mit Parametern:

$$f(n, \vec{x}) = G(f \upharpoonright n, n, \vec{x})$$

Bemerkung

Unter geeigneten Bedingungen [2] lässt sich jede Wertverlaufsrekursion auch als primitive Rekursion realisieren.

[2] im Wesentlichen Codierung von endlichen Sequenzen

Aufgabe

Implementieren Sie die Wertverlaufsrekursion als Funktion höherer Ordnung.

```
let rec courseOfValueRec g n x = ...
```

Aufgabe

Implementieren Sie die Wertverlaufsrekursion als Funktion höherer Ordnung.

```
let rec courseOfValueRec g n x =
    List.init n (fun i -> courseOfValueRec g i x)
    |> g <| n <| x
```

Aufgabe

Implementieren Sie mithilfe der Wertverlaufsrekursion eine Funktion g, die für jedes $n \in \mathbb{N}$ die Anzahl Binärstrings der Länge n, die "111" nicht als Teilstring haben, zurückgibt. Implementieren Sie eine Funktion `enumerator`, die alle "111 - freien" Binärstrings einer bestimmten Länge aufzählt. Benutzen Sie die `enumerator` Funktion um die Funktion g an zufälligen Werten (< 30) zu testen.

Formen der Rekursion

Wertverlaufsrekursion

```
let rec g = function
    | n when n < 3 -> pown 2 n
    | n -> g (n - 1) + g (n - 2) + g (n - 3)
```

Achtung: Variablennamen (`enumerator`, `extender`) wurden abgekürzt damit alles auf die Breite einer Folie passt.

```
let rec enum n =
    let ext = function
        | 1::1::xs -> [0::1::1::xs]
        | xs -> [0::xs;1::xs]
    if n = 0 then [[]]
    else [for x in enum (n-1) do yield! (ext x)]
```

Achtung: Variablennamen (`enumerator`, `extender`) wurden abgekürzt damit alles auf die Breite einer Folie passt.

```
let counter = enum >> List.length

let r = System.Random()
for i in 0..10 do
    let case = r.Next(20)
    let x, y = g case, counter case
    let result = if x = y
        then "O.K."
        else "Failed!"
    printfn "Case: %i --> %s" case result
```

- Bis jetzt haben wir uns bei rekursiven Aufrufen stets auf Funktionswerte mit "kleineren" Argumenten gestützt (z.B. $2^4 = 2 \cdot 2^3$).
- Wir haben Rekursion entlang der "üblichen Ordnung" $<$ der natürlichen Zahlen angewendet.
- Rekursion kann entlang jeder Relation angewendet werden, die keine unendlich absteigenden Ketten[3] erlaubt (Wohlfundiertheit).

[3] Ist die Relation nicht wohlfundiert, kann nicht garantiert werden, dass die definierte Funktion total und/oder eindeutig ist oder überhaupt existiert.

Die allgemeine Rekursion einer Funktion $f : X \to Y$ lässt sich wie folgt formulieren

$$f(x) = G(f \restriction R(x), x)$$

wobei $R : X \to \mathcal{P}(X)$.

Bemerkung

Das Rekursionsschema, wie es oben formuliert ist, führt nur dann zu einer totalen, wohldefinierten Funktion f, wenn die von R induzierte Relation wohlfundiert ist.

Die Funktion $col : \mathbb{N} \to \mathbb{N}$ sei durch folgende Vorschrift gegeben:

$$col(x) = \begin{cases} \frac{x}{2} & \text{falls } x \text{ gerade} \\ 3x + 1 & \text{sonst} \end{cases}$$

Die n-te Collatz[4] - Sequenz C_n erhält man dadurch, dass man mit n beginnend so lange die Funktion col anwendet, bis der Wert 1 erreicht wird. Die Folge C_{17} ist demnach durch

$$17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

gegeben. Der Ausdruck $|C_n|$ bezeichne die Länge der Sequenz C_n. Es gilt zum Beispiel $|C_{17}| = 13$.

[4] https://de.wikipedia.org/wiki/Collatz-Problem

Aufgabe

Implementieren Sie die Funktion $L := x \mapsto |C_x|$.

Aufgabe

Implementieren Sie die Funktion $L := x \mapsto |C_x|$.

```
let rec L = function
    | 1 | 0 -> 1
    | n -> 1 + L (col n)
```

Aufgabe (Bonus)

Definieren Sie mit Hilfe des Schemas der allgemeinen Rekursion die Funktion L. Geben Sie dafür explizit die Funktionen G und R an.

Aufgabe (Bonus)

Definieren Sie mit Hilfe des Schemas der allgemeinen Rekursion die Funktion L. Geben Sie dafür explizit die Funktionen G und R an.

$$R(n) = \begin{cases} \{col(n)\} & \text{falls } n \neq 1 \\ \varnothing & \text{sonst} \end{cases}$$

$$G(X, n) = \begin{cases} 1 + a & \text{falls } X = \{(k, a)\} \text{ mit } k \in \mathbb{N} \\ 1 & \text{sonst} \end{cases}$$

Ein (in der Anwendung sehr wichtiger) Spezialfall der allgemeinen Rekursion ist die sogenannte strukturelle Rekursion. Sie dient dazu induktiv (oft ebenfalls rekursiv genannt) definierte Typen zu "dekonstruieren". Ein typisches Beispiel einer induktiven Definition ist die von Listen:

```
type 'a lst =
    | Nil
    | Cons of 'a * 'a lst
```

Mit struktureller Rekursion und Pattern - Matching lassen sich elegant Funktionen auf Listen definieren:

```
let rec length = function
    | Nil -> 0
    | Cons (x,xs) -> 1 + (length xs)
```

Bemerkung

Wir haben bereits gesehen, wie F# für Listen "Syntactic-Sugar" bereitstellt, `Nil` *ist* `[]` *und* `Cons` *ist* `::`.

So wie sehr viele DSL's, sind auch (einfache) arithmetische Terme induktiv definiert:

$$term ::= int \mid term + term \mid term \cdot term$$

Aufgabe

Definieren Sie einen Typ `term`, der die oben gegebene Spezifikation erfüllt. Implementieren Sie eine Funktion `eval: term -> int`, die Terme auswertet.

Aufgabe

Definieren Sie einen Typ `term`, der die oben gegebene Spezifikation erfüllt. Implementieren Sie eine Funktion `eval: term -> int`, die Terme auswertet.

```
type term =
    | Num of int
    | Add of term * term
    | Mult of term * term

let rec eval = function
    | Num x -> x
    | Add (t,s) -> eval t + eval s
    | Mult (t,s) -> eval t * eval s
```

Formen der Rekursion

Rekursion ist universell...

Vereinfacht gesagt, kann Alles was mit den üblichen Mitteln der Programmierung erreicht werden kann auch mittels Rekursion umgesetzt werden. Aufgrund dieser Tatsache sind Formalismen, die vollständig auf Iterationen verzichten, aber Rekursion in einer allgemeinen Form unterstützen, Turing - mächtig.
Zur Veranschaulichung das Beispiel einer einfachen WHILE-Schleife:

```
while p do
    effect i
    p <- f i
    i <- i + 1
```

Rekursiv umgesetzt:

```
let rec myWhile p f i eff =
    if p then eff i; myWhile (f i) f (i+1) eff
```

Formen der Rekursion

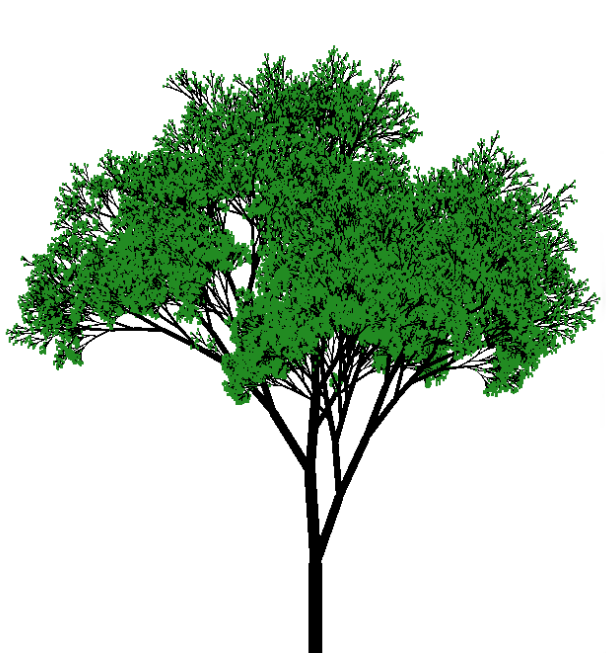
Malwettbewerb

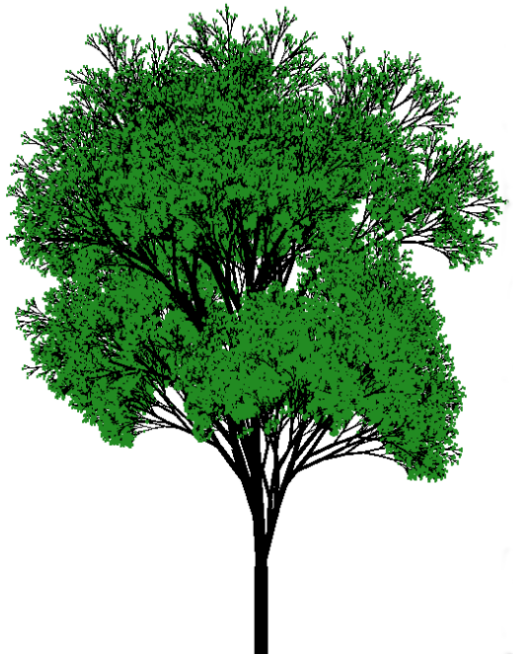
Bevor wir uns dem nächsten “ernsten” Abschnitt widmen, wollen wir uns ein wenig freien Lauf lassen und nebenbei die Konzepte, die wir gerade gelernt haben, festigen. Laden Sie von OLAT die Datei malen.fsx herunter. Nutzen Sie die vorimplementierten Funktionen um einen (bildlich dargestellten) Baum zu erzeugen. Der schönste Baum wird mit einem Freigetränk in der Cafeteria belohnt!

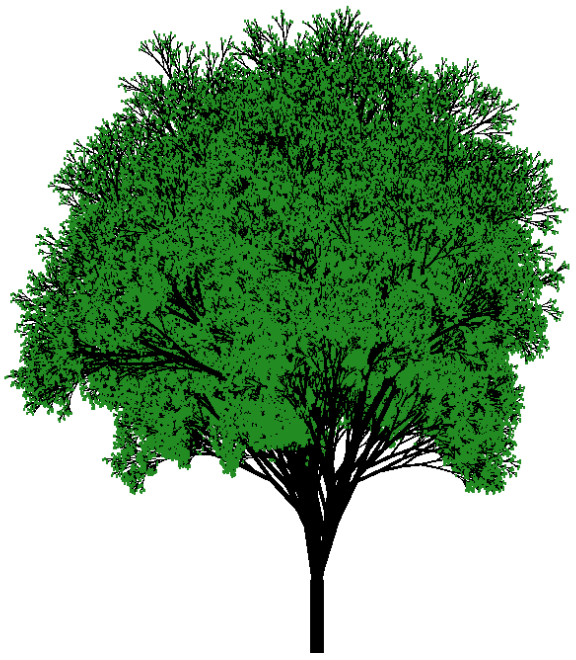
Formen der Rekursion

Malwettbewerb

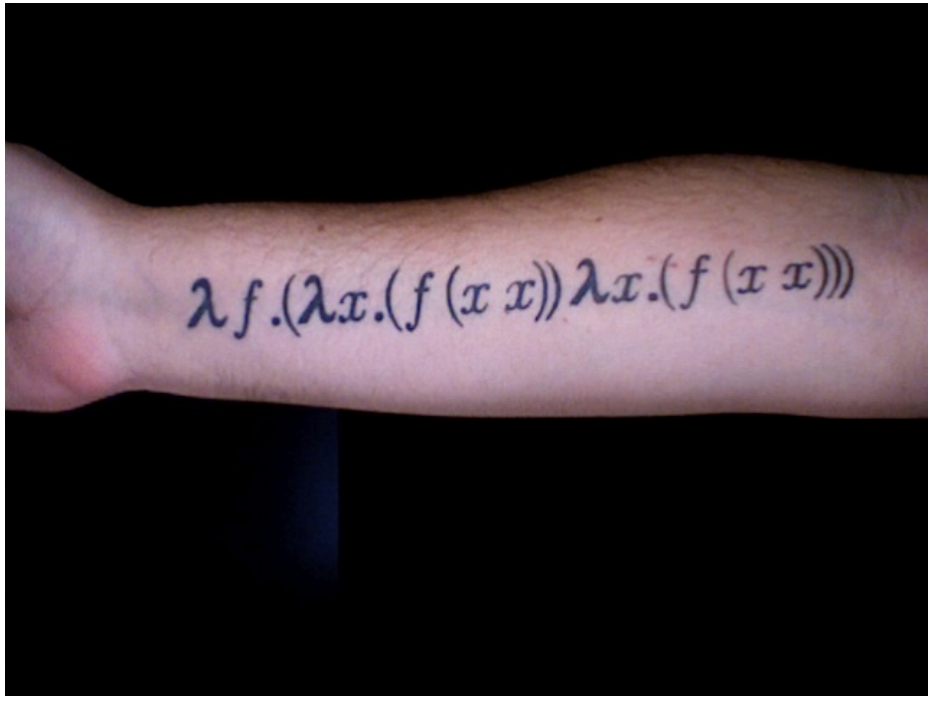
Spielen Sie mit Parametern und Zufallszahlen, um Ihren Bäumen ein lebendiges Aussehen zu verleihen.







Fixpunkte



Fixpunkte

Fixpunkte bieten einen konstruktiven Zugang zu rekursiv definierten Funktionen[5]. Sie formalisieren die Idee des "Aufbauens einer rekursiven Struktur von Unten"[6].

Definition

Als Fixpunkte einer Funktion $F : X \to Y$ werden Elemente $x \in X \cap Y$ mit

$$F(x) = x$$

bezeichnet.

[5] Das heisst zu verstehen, wieso die verschiedenen Rekursionsgleichungen eine Lösung (die zu definierende Funktion) besitzen und wie diese konstruiert wird.

[6] Stichwort: Fixpunktiteration

Fixpunkte

Unter geeigneten Umständen[7], lassen sich Definitionen als Fixpunktgleichungen darstellen:

$$\text{Das } x \text{ mit der Eigenschaft: } f(x) = x.$$

Man kann zum Beispiel die Zahl 0 durch die Gleichung

$$2x = x$$

definieren.

[7] Wenn der Fixpunkt existiert und eindeutig ist.

Fixpunkte

Wir betrachten nun Fixpunkte von Funktionen höherer Ordnung[8]. Wir beginnen mit dem Beispiel

$$\text{expF}(f) = x \mapsto \begin{cases} 1 & \text{falls } x = 0 \\ 2 \cdot f(x-1) & \text{sonst.} \end{cases}$$

Oder äquivalent dazu

```
let expF f x =
    if x = 0 then 1
    else 2 * f (x - 1)
```

[8] In diesem Zusammenhang oft Funktionale genannt.

Fixpunkte

Gehen wir nun davon aus, dass wir einen Fixpunkt H von expF haben. Es gelte also

$$\mathrm{expF}(H) = H$$

und deshalb

$$\mathrm{expF}(H)(x) = H(x).$$

Fixpunkte

Wir untersuchen nun das I/O - Verhalten von H...

Fixpunkte

Wir untersuchen nun das I/O - Verhalten von H...

$$H(0) = \mathrm{expF}(H)(0) = 1$$
$$H(n+1) = \mathrm{expF}(H)(n+1) = 2 \cdot H(n)$$

Fixpunkte

Wir untersuchen nun das I/O - Verhalten von H...

$$H(0) = \mathrm{expF}(H)(0) = 1$$
$$H(n+1) = \mathrm{expF}(H)(n+1) = 2 \cdot H(n)$$

... und erkennen, dass $H(x) = 2^x$ gilt.

Fixpunkte

Wir haben vorher gesehen, dass der Fixpunkt von $expF$, unter der Annahme dass er existiert, den Rekursionsgleichungen der Funktion $x \mapsto 2^x$ genügt. Es bleibt die Frage, welche Umstände die Existenz solch eines Fixpunktes garantieren und wie dieser gegebenenfalls konstruiert werden kann.

Definition

Sind $f, g : X \to X$ partielle Funktionen, dann nennen wir g eine Erweiterung von f, falls $g \restriction Y = f$ für eine Teilmenge $Y \subseteq X$ gilt. Wir schreiben in diesem Fall $f \sqsubseteq g$.

Bemerkung

$f \sqsubseteq g$ gilt genau dann, wenn der Definitionsbereich D_f von f eine Teilmenge des Definitionsbereiches D_g von g ist, und wenn für alle $x \in D_f \cap D_g$ die Identität $f(x) = g(x)$ gilt.

- Die leere Menge $\varnothing$ kann, für alle Mengen X und Y, als partielle Funktion $\varnothing : X \to Y$ mit $x \mapsto$ “undefiniert” aufgefasst werden.

- Die leere Menge $\varnothing$ kann, für alle Mengen X und Y, als partielle Funktion $\varnothing : X \to Y$ mit $x \mapsto$ “undefiniert” aufgefasst werden.
- Eine bemerkenswerte Eigenschaft der Funktion $\varnothing$ ist allerdings die Tatsache, dass sie bezüglich der Relation $\sqsubseteq$ das kleinste Element ist.

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element?

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element? Nein!

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element? Nein!
- ...maximale Elemente?

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element? Nein!
- ...maximale Elemente? Ja, totale Funktionen.

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element? Nein!
- ...maximale Elemente? Ja, totale Funktionen.
- Kann man noch mehr sagen?

- Gibt es bezüglich $\sqsubseteq$ ein grösstes Element? Nein!
- ...maximale Elemente? Ja, totale Funktionen.
- Kann man noch mehr sagen? Jede Kette $\mathcal{K}$ hat eine kleinste obere Schranke $\cup\mathcal{K}$.

- Wie können wir diese Umstände dazu verwenden Fixpunkte “zu bauen”?

- Wie können wir diese Umstände dazu verwenden Fixpunkte “zu bauen”?
- Wir versuchen es am Beispiel expF...

Wir implementieren die Funktion $\varnothing$:

Wir implementieren die Funktion $\varnothing$:

```
let E x = None
```

Wir implementieren die Funktion $\varnothing$:

```
let E x = None
```

Und das um eine optionale Ausgabe angepasste Funktional *expF*:

Wir implementieren die Funktion $\varnothing$:

```
let E x = None
```

Und das um eine optionale Ausgabe angepasste Funktional *expF*:

```
let expF f x =
    if x < 1 then Some 1
    else Option.map ((*) 2) (f (x - 1))
```

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?
  `None`, `None`, `None`,...

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?
  `None`, `None`, `None`,...
- ...`expF E 0`, `expF E 1`, `expF E 2`,...

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?
  `None`, `None`, `None`,...
- ...`expF E 0`, `expF E 1`, `expF E 2`,...
  `Some 1`, `None`, `None`,...

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?
  `None`, `None`, `None`,...
- ...`expF E 0`, `expF E 1`, `expF E 2`,...
  `Some 1`, `None`, `None`,...
- ...`expF (expF E) 0`, `expF (expF E) 1`, `expF (expF E) 2`,...

- Was passiert wenn wir `E 0`, `E 1`, `E 2`,... aufrufen?
  `None`, `None`, `None`,...
- ...`expF E 0`, `expF E 1`, `expF E 2`,...
  `Some 1`, `None`, `None`,...
- ...`expF (expF E) 0`, `expF (expF E) 1`, `expF (expF E) 2`,...
  `Some 1`, `Some 2`, `None`

Um einen Wert für das Argument n zu erhalten, müssen wir das Funktional `expF` $n + 1$-mal auf `E` anwenden.

[9] Entspricht der Exponentialfunktion $\rightarrow$ ausprobieren!

Um einen Wert für das Argument n zu erhalten, müssen wir das Funktional `expF` $n + 1$-mal auf `E` anwenden.
Das können wir automatisieren:

```
let rec iter n f =
    if n < 1 then fun x -> x
    else f >> iter (n - 1) f
```

[9] Entspricht der Exponentialfunktion $\rightarrow$ ausprobieren!

Um einen Wert für das Argument n zu erhalten, müssen wir das Funktional `expF` $n + 1$-mal auf `E` anwenden.
Das können wir automatisieren:

```
let rec iter n f =
    if n < 1 then fun x -> x
    else f >> iter (n - 1) f
```

Wir können nun folgende Funktion[9] deklarieren:

[9] Entspricht der Exponentialfunktion → ausprobieren!

Um einen Wert für das Argument n zu erhalten, müssen wir das Funktional `expF` $n+1$-mal auf `E` anwenden.
Das können wir automatisieren:

```
let rec iter n f =
    if n < 1 then fun x -> x
    else f >> iter (n - 1) f
```

Wir können nun folgende Funktion[9] deklarieren:

```
let exp2 x = iter (x+1) expF E x
```

[9] Entspricht der Exponentialfunktion → ausprobieren!

Um einen Wert für das Argument n zu erhalten, müssen wir das Funktional `expF` $n+1$-mal auf `E` anwenden.
Das können wir automatisieren:

```
let rec iter n f =
    if n < 1 then fun x -> x
    else f >> iter (n - 1) f
```

Wir können nun folgende Funktion[9] deklarieren:

```
let exp2 x = iter (x+1) expF E x
```

...und uns im Folgenden davon überzeugen, dass es sich dabei tatsächlich um einen Fixpunkt des Funktionals expF handelt.

[9] Entspricht der Exponentialfunktion → ausprobieren!

"Mathematisch gesprochen" haben wir eine Funktionenfolge[10] $(f_n)_{n \in \mathbb{N}}$ mit folgenden Eigenschaften definiert:

- Die Funktionenfolge ist eine aufsteigende Kette,...

$$f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$$

[10] Jede Funktion f_n entspricht `iter n expF`.

"Mathematisch gesprochen" haben wir eine Funktionenfolge[10] $(f_n)_{n \in \mathbb{N}}$ mit folgenden Eigenschaften definiert:

- Die Funktionenfolge ist eine aufsteigende Kette,...

$$f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$$

- ... die durch Iteration des Funktionals `expF` gegeben ist

$$f_0 = \varnothing$$
$$f_{n+1} = \mathrm{expF}(f_n).$$

[10] Jede Funktion f_n entspricht `iter n expF`.

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Die Funktion f hat folgende Eigenschaften:

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Die Funktion f hat folgende Eigenschaften:

- Entspricht der Funktion `exp2`.

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Die Funktion f hat folgende Eigenschaften:

- Entspricht der Funktion `exp2`.
- Falls $f(x) = y$, dann gibt es ein $n \in \mathbb{N}$ mit $f_n(x) = y$.

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Die Funktion f hat folgende Eigenschaften:

- Entspricht der Funktion `exp2`.
- Falls $f(x) = y$, dann gibt es ein $n \in \mathbb{N}$ mit $f_n(x) = y$.
- Falls $f_n(x) = y$, dann gilt auch $f(x) = y$.

Weil die Funktionenfolge $(f_n)_{n\geq 0}$ eine Kette ist, gibt es eine kleinste obere Schranke

$$f = \cup_n f_n.$$

Die Funktion f hat folgende Eigenschaften:

- Entspricht der Funktion `exp2`.
- Falls $f(x) = y$, dann gibt es ein $n \in \mathbb{N}$ mit $f_n(x) = y$.
- Falls $f_n(x) = y$, dann gilt auch $f(x) = y$.

Die erste Eigenschaft folgt aus $\texttt{exp2}(x) = \cup_{i \leq x+1} f_i(x)$, die zweite weil f die **kleinste** obere Schranke ist und die dritte daraus, dass f überhaupt eine obere Schranke ist.

Wir wollen uns jetzt davon überzeugen, dass f ein Fixpunkt von expF ist[11].

[11] Und somit auch exp2

Wir wollen uns jetzt davon überzeugen, dass f ein Fixpunkt von expF ist[11].

$$\text{expF}(f)(x) = f(x)$$

[11] Und somit auch exp2

Wir wollen uns jetzt davon überzeugen, dass f ein Fixpunkt von expF ist[11].

$$\text{expF}(f)(x) = f(x)$$

- Fall $x < 1$:

$$f(x) = f_2(x) = \text{expF}(E)(x) = 1 = \text{expF}(f)(x)$$

[11] Und somit auch exp2

Wir wollen uns jetzt davon überzeugen, dass f ein Fixpunkt von expF ist[11].

$$\text{expF}(f)(x) = f(x)$$

- Fall $x < 1$:

$$f(x) = f_2(x) = \text{expF}(E)(x) = 1 = \text{expF}(f)(x)$$

- Fall $x \geq 1$:

$$\begin{aligned} \text{expF}(f)(x) &= 2 \cdot f(x-1) \\ &= 2 \cdot f_n(x-1) \qquad /\text{für ein geeignetes } n \\ &= \text{expF}(f_n)(x) \\ &= f_{n+1}(x) = f(x) \end{aligned}$$

[11] Und somit auch exp2

- Kann man die Konstruktion für den Fixpunkt des Funktionals expF auch für andere Funktionale anwenden?

- Kann man die Konstruktion für den Fixpunkt des Funktionals expF auch für andere Funktionale anwenden?
- JA! Der folgende Fixpunktsatz (und dessen Beweis) verallgemeinern diese Konstruktion.

Definition ([12])

Ein Funktional F heisst:

- Monoton, falls $f \sqsubseteq g \Rightarrow F(f) \sqsubseteq F(g)$
- Stetig, falls für jede Kette $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \ldots$ gilt

$$F(\cup_i f_i) \sqsubseteq \cup_i F(f_i).$$

Bemerkung

Ist F stetig und monoton, dann gilt

$$F(\cup_i f_i) = \cup_i F(f_i).$$

[12] Diese Definitionen werden normalerweise allgemeiner für beliebige "vollständige partielle Ordnungen" gemacht.

Die Funktionale, die in Fixpunktdefinitionen von rekursiven Funktionen auftreten, sind monoton und stetig[13]:

[13] Genaueres dazu in der Lektüre (Fixpunkte) auf OLAT

Die Funktionale, die in Fixpunktdefinitionen von rekursiven Funktionen auftreten, sind monoton und stetig[13]:

- Monotonie: “Das Funktional 'sagt' wie ein Anfangsabschnitt einer Funktion erweitert wird. Wenn schon mehr vorhanden ist, dann ist am Schluss nicht weniger da”.

[13] Genaueres dazu in der Lektüre (Fixpunkte) auf OLAT

Die Funktionale, die in Fixpunktdefinitionen von rekursiven Funktionen auftreten, sind monoton und stetig[13]:

- Monotonie: "Das Funktional 'sagt' wie ein Anfangsabschnitt einer Funktion erweitert wird. Wenn schon mehr vorhanden ist, dann ist am Schluss nicht weniger da".
- Stetigkeit: Dies folgt aus der Tatsache, dass "für die Bestimmung von $F(f)(x)$ jeweils nur eine endliche Anzahl Funktionswerte von f bekannt sein müssen".

[13] Genaueres dazu in der Lektüre (Fixpunkte) auf OLAT

Theorem (Fixpunktsatz)

Gegeben sei ein stetiges, monotones Funktional F, dann gilt

- $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ *ist eine (aufsteigende) Kette,*
- $f = \cup_n f_n$ *ist der kleinste*[14] *Fixpunkt von* F*,*

wobei die Folge der f_n *durch*

$$f_0 = \varnothing$$
$$f_{n+1} = F(f_n)$$

gegeben ist.

[14] Bezüglich $\sqsubseteq$

Beweis.

Der Beweis funktioniert im Wesentlichen wie im Beispiel expF. Mit Induktion nach n sehen wir, dass $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ eine (aufsteigende) Kette ist. Weiter nützen wir aus, dass F monoton und stetig ist, und dadurch $\cup_n F(f_n) = F(\cup_n f_n)$ gilt:

$$F(f) = F(\cup_n f_n) = \cup_n F(f_n) = \cup_n f_{n+1} = \cup_n f_n = f.$$

Somit ist f ein Fixpunkt von F. Wir müssen noch zeigen, dass f der kleinste Fixpunkt ist. Dazu genügt es mit Induktion nach n zu zeigen, dass für alle $n \in \mathbb{N}$ die Beziehung $f_n \sqsubseteq g$ für einen beliebeigen Fixpunkt g von F gilt (Monotonie ausnützen). □

Nachdem wir eine Strategie kennengelernt haben, wie man Fixpunkte von Funktionalen explizit “bauen” kann, wollen wir dies nun ausnützen und im Folgenden einen Fixpunktkombinator, eine Funktion, die von gegebenen Funktionalen den Fixpunkt berechnet, in F# implementieren.

Wir erinnern uns an den Fixpunkt von expF:

Wir erinnern uns an den Fixpunkt von expF:

```
let exp2 x = iter (x+1) expF E x
```

Wir erinnern uns an den Fixpunkt von expF:

```
let exp2 x = iter (x+1) expF E x
```

Weil die gleiche Strategie "Fixpunkte zu bauen" auch für andere Funktionale anwendbar ist, wäre also ein erster Ansatz:

```
let fix1 F x = iter (x+1) F E x
```

Aufgabe

Implementieren Sie den Kombinator `fix1` wie gezeigt. Testen Sie `fix1` mit den Funktionalen expF für die Funktion $x \mapsto 2^x$ und

```
let fakF f x =
    if x < 1 then Some 1
    else Option.map ((*) x) (f (x - 1))
```

für die Fakultätsfunktion.

Aufgabe

Testen Sie `fix1` nun noch mit dem Funktional colF

```
let colF f x =
    if x < 2 then Some 0
    else Option.map ((+) 1) (f (col x))
```

für die Funktion $L = x \mapsto |C_x|$. Was stellen Sie fest?

Der Operator `fix1` funktioniert mit den Funktionalen `expF` und `fakF` gut, aus dem Funktional `colF` resultiert aber eine Funktion mit Definitionslücken. Die Funktionsweise von `fix1` basiert wesentlich darauf, dass man genau weiss, wie oft das gegebene Funktional auf die Funktion `E` angewendet werden muss um eine Funktion zu erhalten, die für das gegebene Argument "genügend weit definiert" ist[15]. Weil dies nicht immer ein gangbarer Weg ist (wie man anhand des Funktionals `colF` sieht), wollen wir den Kombinator dahingehend anpassen, dass er nicht eine feste Anzahl Iterationen vollzieht, sondern "soviele wie nötig".

[15] Dies entspricht im Wesentlichen einer primitiven Rekursion.

Die Kombinatoren `fix2` und `fix2'` wenden das gegebene Funktional so oft an wie nötig (`fix2` mehr als nötig), sie beinhalten implizit die Funktion `iter`:

```
let rec fix2 F x =
    match F E x with
    | Some x -> Some x
    | None -> fix2 (F>>F) x
```

```
let rec fix2' F G x =
   match G E x with
   | Some x -> Some x
   | None -> fix2' F (F>>G) x
```

Mit dem Kombinator `fix2'` kommen wir der Konstruktion, die im Beweis des Fixpunktsatzes angewendet wurde, schon ziemlich nahe. Mit der im Beweis des Fixpunktsatzes verwendeten Notation, entspricht der Kombinator im Wesentlichen der Funktion

$$f(n,x) = \begin{cases} \cup_{i<n} f_i(x) & \text{falls definiert} \\ f(n+1,x) & \text{sonst} \end{cases}$$

Anschaulich könnte man den Unterschied etwa wie folgt beschreiben:

`fix2` und `fix2'`	$\cup_n f_n$
Approximiert den Fixpunkt schrittweise, bis die Funktion für das gegebene Argument eine Rückgabe liefert.	Die gesamte Funktion wird zuerst aufgebaut und dann auf Argumente angewendet.

Anschaulich könnte man den Unterschied etwa wie folgt beschreiben:

`fix2` und `fix2'`	$\cup_n f_n$
Approximiert den Fixpunkt schrittweise, bis die Funktion für das gegebene Argument eine Rückgabe liefert.	Die gesamte Funktion wird zuerst aufgebaut und dann auf Argumente angewendet.

Könnene wir eine Funktion in F# angeben, die der Konstruktion im Beweis noch etwas näher kommt?

Wir verwenden den Kombinator `fix2'`, modifizieren Ihn aber so, dass immer nur die zweite Klausel des Pattern - Matches zum Zug kommt. Wir können diesen Kombinator etwa wie folgt definieren:

```
let rec fixU F G x = fixU F (F>>G) x
```

der Kombinator `fixU` hat das offensichtliche Problem, dass er "unendlich expandiert". Dies ist nicht verwunderlich, da wir ihn ja so geschrieben haben, dass er zuerst die ganze Funktion $\cup_n f_n$ zu konstruieren versucht, bevor diese jemals angewendet wird.

Expansion von `fixU`:

```
fixU F F x
fixU F (F»F) x
fixU F (F»F»F) x
fixU F (F»F»F»F) x
fixU F (F»F»F»F»F) x
...
```

In der Notation des Fixpunktsatzes kann man die Deklaration von `fixU` grob wie folgt verstehen:

$$\underbrace{\cup_n f_{n'}}_{\texttt{fixU F G}} = \underbrace{\cup_n F(f_{n'})}_{\texttt{fixU F (F»G)}}$$

Aus dem Beweis des Fixpunktsatzes kennen wir die Identität:

$$\cup_n F(f_n) = F(\cup_n f_n)$$

Aus

$$\cup_n F(f_{n'}) = F(\cup_n f_{n'}) = F(\cup_n f_n)$$

folgt, dass wir das Funktional F "aus dem Fixpunkt herausziehen" und auf den Parameter G verzichten können:

```
let rec fix F f = F (fix F) f
```

Dies führt zu einer "sehr deklarativen" Definition des Fixpunktoperators. Das Problem der unendlichen Expansion wird dadurch gelöst, dass die im Funktional F vorhandene "Abbruchbedingung" ins Spiel kommt, weil zuerst F angewendet wird.

Im folgenden werden wir die Fixpunktoperatoren mit erstaunlichem Effekt zum Einsatz bringen. Wir erinnern uns dazu an die Funktionen `logger` und `memoizer`:

```
let logger f x =
    printfn "computing value at %A..." x
    f x
```

```
let memoize (m: Dictionary<'a,'b>) f x =
    match m.TryGetValue(x) with
    | (true,y) -> y
    | _ -> let y = f x in m.Add(x,y); y
```

Als nächstes brauchen wir eine einigermassen interessante Funktion mit der wir arbeiten können. Die Fibonacci Funktion eignet sich hier hervorragend:

Aufgabe

Definieren Sie ein Funktional `fibF`, so dass `fix fibF` der Fibonacci Funktion entspricht. Alternativ dazu können Sie auch die Fibonacci Funktion wie üblich rekursiv definieren.

Aufgabe

Definieren Sie ein Funktional `fibF`, so dass `fix fibF` der Fibonacci Funktion entspricht.

```
let fibF f x =
    if x < 2 then x
    else f (x - 1) + f (x - 2)

let fib = fix fibF
```

Weil unsere Fibonacci Funktion ziemlich langsam ist, wollen wir sie memoisieren:

```
let fibM =
    memoize (new Dictionary<int,int>()) fib
```

Aufgabe

Testen Sie das Laufzeitverhalten der Funktion `fibM` und diskutieren Sie Ihre Beobachtungen.

Aufgabe

Testen Sie das Laufzeitverhalten der Funktion `fibM` und diskutieren Sie Ihre Beobachtungen.

Einerseits zeigt die memoisierte Variante der Fibonacci Funktion das erwartete Laufzeitverhalten:

```
> fib 30 -> 01.387
> fib 30 -> 01.339
> fibM 30 -> 01.238
> fibM 30 -> 00.000
```

Andererseits könnte es besser sein:

```
> fibM 30 -> 01.263
> fibM 29 -> 00.787
```

Wieso wurde `fib 29` nicht memoisiert, obwohl es beim Funktionsaufruf von `fibM 30` zweifelsohne berechnet wurde? Wie können wir die Funktion `memoize` dazu bringen auch rekursive Aufrufe zu speichern?

Dasselbe Problem zeigt sich beim loggen von `fib`.

```
>fibL 30;;
computing value at 30...
val it : int = 832040
```

Obwohl zur Berechnung von `fib 30` alle vorhergehenden Funktionswerte (mehrmals) berechnet werden, wird nur der einzige explizite Aufruf aufgezeichnet.

Anschaulich gesprochen wollen wir nicht nur den “aktuellen” Funktionsaufruf aufzeichnen, sondern auch alle von diesem Aufruf getätigten Aufrufe und alle von den von diesem Aufruf getätigten Aufrufe und.... ein Fixpunkt!

Um unseren Fixpunktkombinator anwenden zu können, beginnen wir damit, aus den "Funktionsmodifikatoren" `memoize` und `logger` "Funktionalmodifikatoren" zu machen.

```
let loggerF F f x =
    printfn "computing value at %A..." x
    F f x
```

```
let memoizeF (m:Dictionary<'a,'b>) F f x =
    match m.TryGetValue(x) with
    | (true, y) -> y
    | _ -> let y = F f x in m.Add(x,y); y
```

Nun können wir "rekursiv" memoisierte und geloggte Varianten der Fibonacci Funktion mit Hilfe des Fixpunktkombinators deklarieren:

```
let fibRecL = fix (loggerF fibF)
```

```
let fibRecM =
    let m = new Dictionary<int,int>()
    fix (memoizeF m fibF)
```

Nun wollen wir sehen, ob die Funktionen machen was wir uns erhoffen.
Wir beginnen mit der geloggten Fibonacci Funktion:

Nun wollen wir sehen, ob die Funktionen machen was wir uns erhoffen.
Wir beginnen mit der geloggten Fibonacci Funktion:

```
>fibRecL 3;;
computing value at 3...
computing value at 2...
computing value at 1...
computing value at 0...
computing value at 1...
val it : int = 2
```

Nun wollen wir sehen, ob die Funktionen machen was wir uns erhoffen.
Wir beginnen mit der geloggten Fibonacci Funktion:

```
>fibRecL 3;;
computing value at 3...
computing value at 2...
computing value at 1...
computing value at 0...
computing value at 1...
val it : int = 2
```

Also das sieht schon mal gut aus 😎

Nun die memoisierte Variante:

Nun die memoisierte Variante:

```
> fibRecM 30 -> 00.000
```

Nun die memoisierte Variante:

```
> fibRecM 30 -> 00.000
```

Ok, das sieht zwar sehr gut aus, aber wieso wurde der Wert `fibRecM 30` scheinbar schon vor dem ersten Aufruf gecached?

```
fibRecM 20000 -> 00.506
fibRecM 18000 -> 00.000
fibRecM 25000 -> 00.004
```

```
fibRecM 20000 -> 00.506
fibRecM 18000 -> 00.000
fibRecM 25000 -> 00.004
```

Aha, die memoisierte Funktion hat den Wert nicht gecached, sondern ist einfach grundsätzlich viel effizienter. Ansonsten verhält sie sich genau wie wir uns das gewünscht hätten: Alle Berechnungen die (auch rekursiv) bereits evaluiert wurden stehen bei späteren (auch impliziten rekursiven) Aufrufen zur Verfügung! 😎

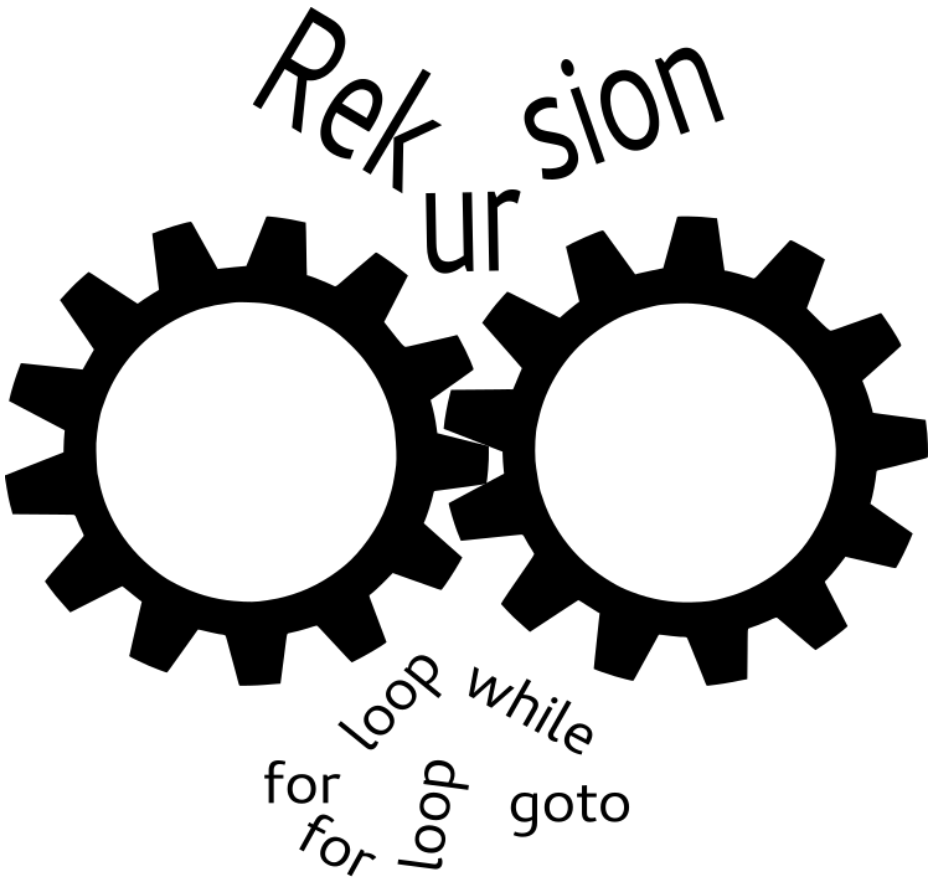
Um zu sehen, wieso die memoisierte Version der Fibonacci Funktion auch bei nicht aufgezeichneten Werten sehr viel schneller als `fib` und `fibM` arbeitet, wollen wir die memoisierte Funktion loggen und untersuchen wie die genannte Beobachtung zustande kommt.

Um eine memoisierte und geloggte Funktion zu erstellen, können wir die Operatoren `loggerF` und `memoizeF` einfach komponieren!

```
let fibRecML =
    let m = new Dictionary<int,int>()
    fix ((memoizeF m << loggerF) fibF)
```

..und sehen, dass der Gewinn in Effizienz dadurch zustandekommt, dass rekursive Aufrufe memoisiert werden und darum nur ein mal berechnet werden müssen:

```
fibRecML 7;;
computing value at 7...
computing value at 6...
computing value at 5...
computing value at 4...
computing value at 3...
computing value at 2...
computing value at 1...
computing value at 0...
```


Rek ur sion
loop while
for Loop
for
goto

In der funktionalen Programmierung werden viele, in anderen Paradigmen typischerweise iterativ formulierte, Algorithmen rekursiv ausgedrückt. Aus dieser Präferenz folgt, dass funktionale Sprachen[16] Optimierungen anbieten, um Rekursion effizient zu übersetzen. Wichtig ist dabei insbesondere, nicht für jeden rekursiven Funktionsaufruf den Aufrufstapel zu vergrössern. Dadurch können Stapelüberläufe auch bei tiefer Rekursion verhindert werden. Die "tail-call" Optimierung behandelt genau diesen Fall.

[16] Genauer Compiler für funktionale Sprachen

Compiler - Optimierungen

- Eine Deklaration liegt in endrekursiver Form oder "tail-recursive" vor, wenn Resultate von rekursiven Aufrufen direkt[17] zurückgegeben werden.
- Endrekursive Funktionsdefinitionen werden vom F# Compiler effizient umgesetzt[18].
- Die meisten (Compiler von) Funktionalen Sprachen übersetzen endrekursive Funktionen so, dass bei deren Ausführung konstanter Stapelspeicher in Anspruch genommen wird. Man spricht dabei von "tail-call" Optimierung.

[17] D.h. ohne weitere Modifikation oder Verwendung.

[18] Entweder iterativ durch Goto's oder als .Net-tail-calls. Ausnahmen: "try-catch" und "try-final" Blöcke (keine echten Ausnahmen) sowie Funktionen mit () Rückgabe.

Endrekursion

Die Deklaration

```
let rec sum = function
    | [] -> 0
    | x::xs -> x + (sum xs)
```

ist nicht endrekursiv, weil das Resultat des rekursiven Funktionsaufrufes als Summand weiterverwendet wird. Dieselbe Funktionalität lässt sich endrekursiv deklarieren:

```
let sumTR' acc = function
    | [] -> acc
    | x::xs -> sumTR (x + acc) xs

let sumTR = sumTR' 0
```

Die einfachste Methode eine rekursive Funktion in eine endrekursive Form zu bringen besteht darin, das Zwischenresultat der Rekursion explizit in einem "Akkumulator - oder Zustands - Parameter" mitzuführen.

Endrekursive Form der Fakultätsfunktion mit einem Akkumulator:

```
let rec fakTR acc n =
    if n < 1 then acc
    else fakTR (n*acc) (n - 1)
```

Aufgabe

Bringen Sie folgende Funktionen in eine endrekursive Form:

```
let rec pow x y =
    if y < 1 then 1
    else x * pow x (y - 1)

let rec isPalindrome w =
    let l = String.length w
    if l < 2 then true
    else
        w.[0] = w.[l-1] && isPalindrome w.[1..l-2]
```

Aufgabe

Bringen Sie folgende Funktionen in eine endrekursive Form.

```
let rec powTR' acc x y =
    if y < 1 then acc
    else powTR' (x*acc) x (y - 1)

let powTR = powTR' 1
```

Aufgabe

Bringen Sie folgende Funktionen in eine endrekursive Form.

```
let rec isPalindromeTR' acc w =
    let l = String.length w
    if l < 2 then acc
    else isPalindromeTR' (acc && w.[0] = w.[l-1])
        w.[1..l-2]

    let isPalindromeTR = isPalindromeTR' true
```

- Nicht alle rekursiven Funktionsdeklarationen lassen sich mittels eines Akkumulators in endrekursive Form bringen.

Endrekursion

Continuation - Pattern

- Nicht alle rekursiven Funktionsdeklarationen lassen sich mittels eines Akkumulators in endrekursive Form bringen.
- Anstelle eines Akkumulator - Parameters, der die bereits geleistete Arbeit repräsentiert, gelingt dies aber durch Mitführen eines Funktionsparameters, der die noch zu leistende Arbeit darstellt.

Die Fakultätsfunktion mit dem Continuation - Pattern umgesetzt:

```
let rec fakC f n =
    if n < 1 then f n
    else fakC (fun x  -> n * (f x)) (n - 1)
```

Aufgabe

Bringen Sie mithilfe des Continuation - Patterns die Funktion `map` in eine endrekursive Form.

```
let rec map f = function
    | [] -> []
    | x::xs -> (f x)::(map f xs)
```

Aufgabe

Bringen Sie mithilfe des Continuation - Patterns die Funktion `map` in eine endrekursive Form.

```
let rec mapC' g f = function
    | [] -> List.rev <| g []
    | x::xs -> mapC' (fun y -> (f x)::(g y)) f xs

let mapC f = mapC' (fun x -> x) f
```

Wir betrachten nun ein etwas komplexeres Beispiel. Zuerst definieren wir einen Typ `binTree` für Binärbäume mit ganzzahligen Knoten:

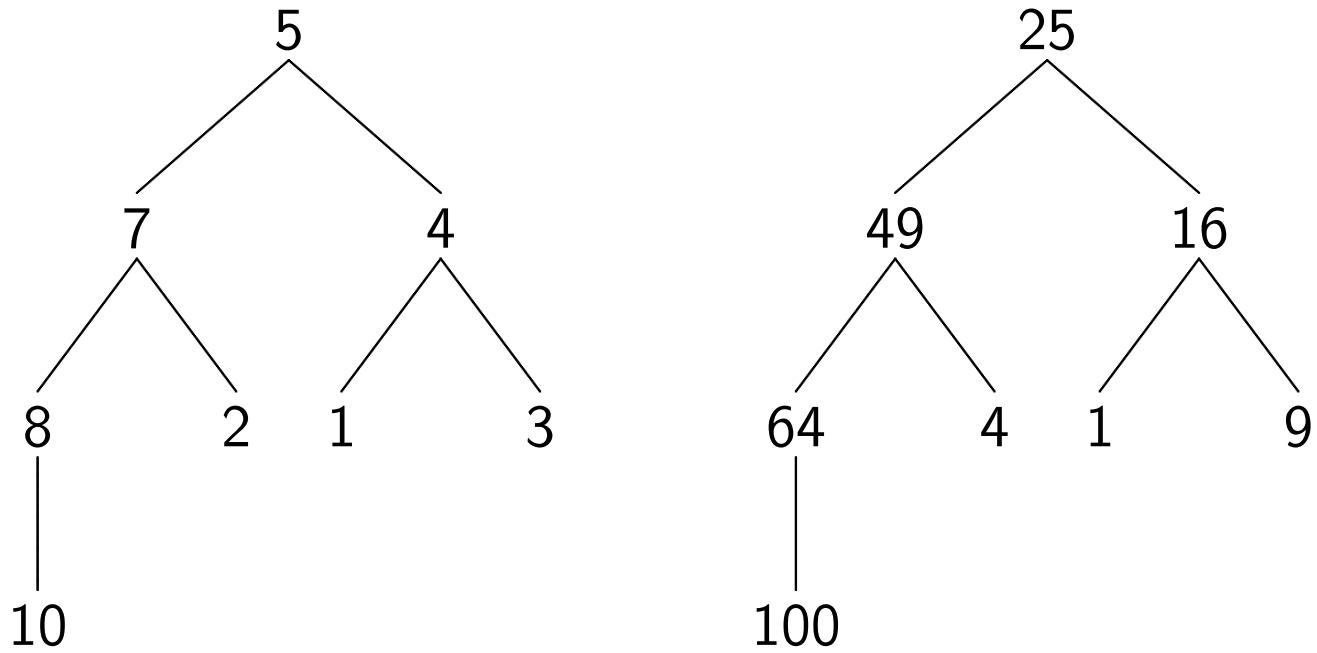
```
type binTree =
    | E
    | Nd of binTree * int * binTree
```

Nun wollen wir die Funktion `treeMap`

```
let rec tMap f = function
    | E -> E
    | Nd (l,x,r) -> Nd (tMap f l, f x, tMap f r)
```

in eine endrekursive Form bringen.

Zur Veranschaulichung der Funktion `treeMap`, illustrieren wir das Beispiel `treeMap (fun x -> x * x)`:



Zur Vorbereitung für unsere Aufgabe implementieren wir ein paar Hilfsfunktionen.

Die Funktion `empty` testet ob ein Baum "leer" ist:

```
let empty = function E -> true | _ -> false
```

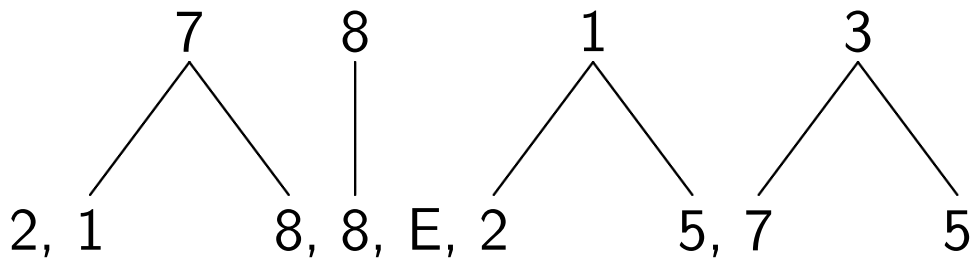
Die Funktion[19]

```
let merge nodes (trees: tree list) =
    let mergeFunction = function
        | (Some x, y, z) -> Node (y,x,z)
        | (None,_,_) -> E
    List.mapi (fun i  x -> mergeFunction (x,
       trees.[2*i],trees.[2*i+1])) nodes
```

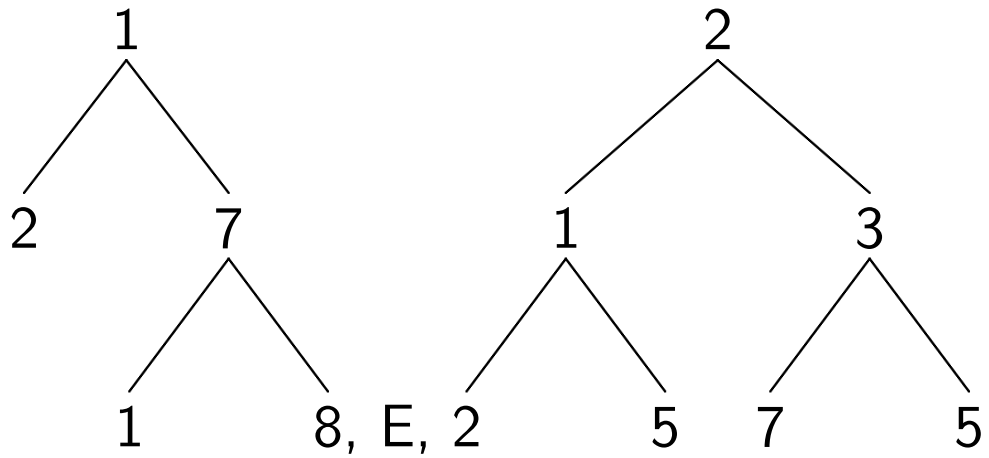
verhält sich wie im Folgenden illustriert.

[19] Vorsicht wegen Zeilenumbrüchen aufgrund zu kurzer Folien.

Ist `nodes = [Some 1; None; Some 2]` und `trees` bestehe aus



Dann ist `merge nodes trees` durch



gegeben.

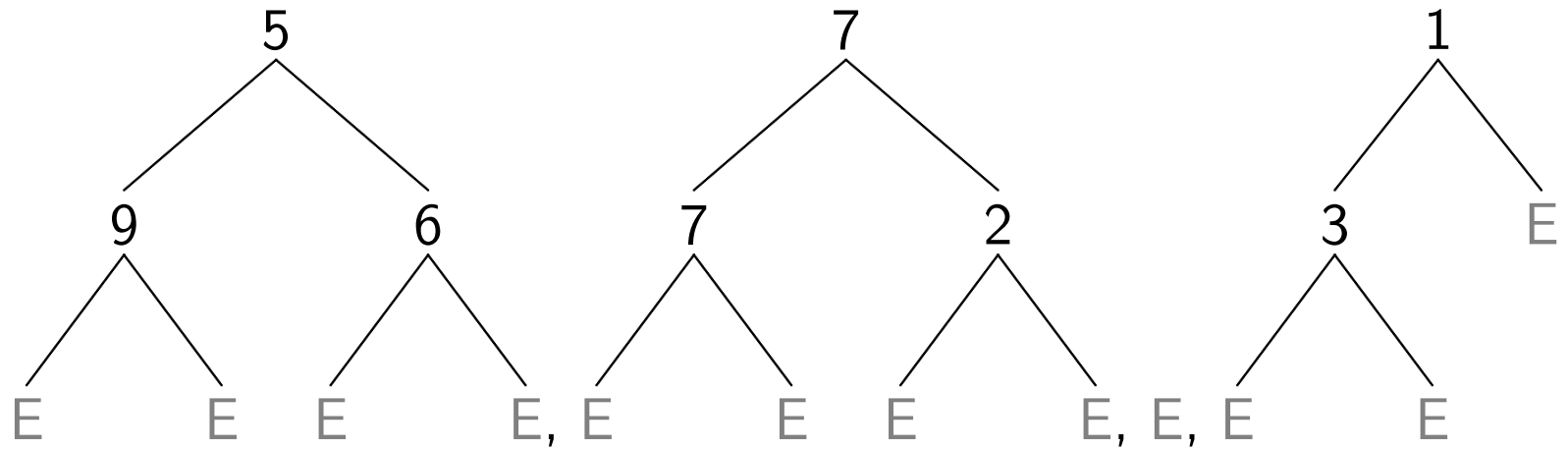
Die Funktion[19]

```
let rec split trees =
    let trees' = List.collect (function E -> [E;E]
        | Node(l,_,r) -> [l; r]) trees
    let nodes =  trees |> List.map (function Node(
       _,x,_) -> Some x | E -> None)
    (nodes, trees')
```

verhält sich "dual" zur Funktion `merge`.

[19] Vorsicht wegen Zeilenumbrüchen aufgrund zu kurzer Folien.

Ist `trees` durch

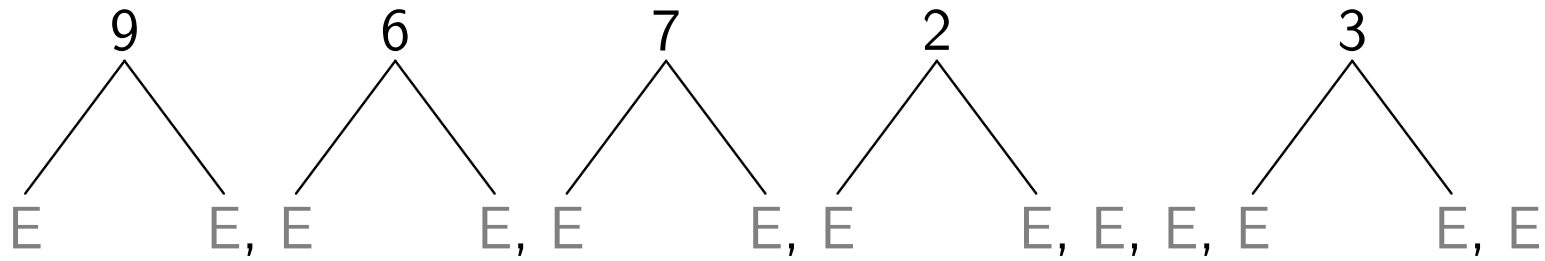


gegeben...

...dann besteht die Rückgabe von `split trees` aus

```
[Some 5; Some 7; None; Some 1]
```

und der Liste von Bäumen



Mithilfe dieser Funktionen können wir eine endrekursive Variante der Funktion `treeMap` wie folgt aufschrieben:

```
let rec mapC' cont f trees =
    if List.forall empty trees then cont trees
    else
        let (nodes, trees') = split trees
        let nodes' = List.map (Option.map f) nodes
        let cont' = (merge nodes') >> cont
        mapC' cont' f trees'

let mapC f t = mapC' List.head f [t]
```

Wir wollen nun einen “Durchlauf” der soeben definierten Funktion nachvollziehen $\Longrightarrow$ Tafel.