



Objektorientierte und Funktionale Programmierung

SS 2013

7 Funktionale Programmierung





7 Funktionale Programmierung ...

Lernziele

- Verständnis funktionaler Programmierkonzepte
 - Funktionen als Werte, Funktionen höherer Ordnung, Polymorphismus, ...
- Auseinandersetzung mit einem nicht-imperativen Programmierparadigma
 - neue Sicht- und Denkweise!
- Vertieftes Verständnis der Rekursion



Literatur

- [Er99], Kap. 1, 2
- [Kr02], Kap. 2, 3, 4
- [Pa00], Kap. 1, 2, 4(.1), 7(.1), 8(.1)
- S. Sabrowski, Schnelleinstieg in Standard ML of New Jersey, 1996.
<http://www-pscb.informatik.tu-cottbus.de/~wwwpscb/studenten/sml.ps>
<http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/gen/ei2/sml.pdf>
- E. Januzaj, SML zum Mitnehmen – Eine Kurzreferenz von
SML-Funktionen
[www.dbs.informatik.uni-muenchen.de/Lehre/Info1/smlref/SML-Kurzreferenz pdf 2.pdf](http://www.dbs.informatik.uni-muenchen.de/Lehre/Info1/smlref/SML-Kurzreferenz%20pdf%202.pdf)



Inhalt

- Konzepte funktionaler Programmiersprachen
- SML: Überblick
- Werte und Ausdrücke
- Tupel, Records und Listen
- Variablen und Funktionen
- Typen und Polymorphismus
- Datentypen und Pattern Matching
- Funktionen höherer Ordnung
- Problemlösung mit Rekursion
- Auswertung funktionaler Programme





Besonderheiten funktionaler Programmiersprachen:

- Sie basieren auf dem Funktionsbegriff der Mathematik
 - ein Programm ist eine Funktion, die aus anderen Funktionen zusammengesetzt ist
- Es gibt keine Variablen, deren Wert verändert werden kann
 - der Variablenbegriff entspricht dem der Mathematik
 - es gibt keine Zuweisungen
 - eine Variable hat an allen Stellen innerhalb ihres Gültigkeitsbereichs immer denselben Wert (referenzielle Transparenz)
- Es gibt weder programmierten Kontrollfluss noch Seiteneffekte
 - keine Anweisungsfolgen, keine Schleifen, ...
 - eine Funktion liefert mit identischen Parametern **immer** dasselbe Ergebnis



(Mathematische) Funktionen

- Eine **Funktion f von A in B** ordnet jedem Element aus der Menge A genau ein Element aus der Menge B zu
 - A ist der Definitionsbereich, B der Wertebereich von f
 - f kann definiert werden durch:
 - eine Aufzählung von Wertepaaren aus $A \times B$
 - eine **Funktionsgleichung**, z.B. $f(x) = \sin(x)/x$
- Eine Funktionsgleichung
 - führt auf der linken Seite Variablen ein, die für Werte aus dem Definitionsbereich stehen
 - hat auf der rechten Seite einen Ausdruck aus Variablen, Konstanten und Funktionen



Funktionen: Begriffe und Schreibweisen

- $A \rightarrow B$ ist die Menge aller Funktionen von A in B
- Ist $f \in A \rightarrow B$, schreiben wir auch $f: A \rightarrow B$
- Für $f: A \rightarrow B$ liefert die **Funktionsanwendung** (**Applikation**) von f auf ein **Argument** $a \in A$ das vermöge f zugeordnete Element aus B
- Schreibweisen für die Funktionsanwendung:
 - $f(a)$ Funktionsschreibweise
 - fa Präfixschreibweise
 - $a1fa2$ Infixschreibweise,
falls $A = A1 \times A2, (a1, a2) \in A$



Funktionen als Werte

- Eine Funktion $f: A \rightarrow B$ ist ein Wert der Menge $A \rightarrow B$
 - genauso, wie $5, 2$ ein Wert der Menge R ist
- Damit können Funktionen
 - als Argumente an andere Funktionen übergeben werden
 - und als Funktionsergebnis auftreten
- Dies führt zu **Funktionen höherer Ordnung**
- Beispiel: Funktionskomposition ◦
 - Der Operator ◦ ist eine Funktion aus der Menge
$$(B \rightarrow C) \times (A \rightarrow B) \rightarrow (A \rightarrow C)$$
 - Funktionsgleichung für ◦: $(f \circ g)(x) = f(g(x))$



Referentielle Transparenz

- Jedes Vorkommen einer Variable (innerhalb ihres Gültigkeitsbereichs) bezeichnet denselben Wert
- Die Semantik hängt nicht wie bei imperativen Sprachen von einem impliziten Zustand (= Speicherbelegung) ab
- Mathematisch ist ein Ausdruck wie $i = i + 1$ sinnlos
 - Gleichung ohne Lösung ($0 = 1$)
- Referentielle Transparenz erlaubt es immer, Variable durch ihre Definition zu ersetzen, ohne die Semantik zu ändern
 - die Auswertung funktionaler Programme basiert genau auf diesem Prinzip
 - dadurch lassen sich Eigenschaften funktionaler Programme einfach(er) beweisen (\rightarrow Bedeutung der funkt. Prog.)



Kein programmierter Kontrollfluß

- In funktionalen Programmen gibt es keine **Anweisungen**
 - keine Zuweisungen, Anweisungsfolgen, Schleifen, bedingte Anweisungen, ...
- Stattdessen: Rekursion und bedingte **Ausdrücke**
- Beispiel: Berechnung der Fakultät von n ($= 1 \cdot 2 \cdot \dots \cdot n$)

In Java:

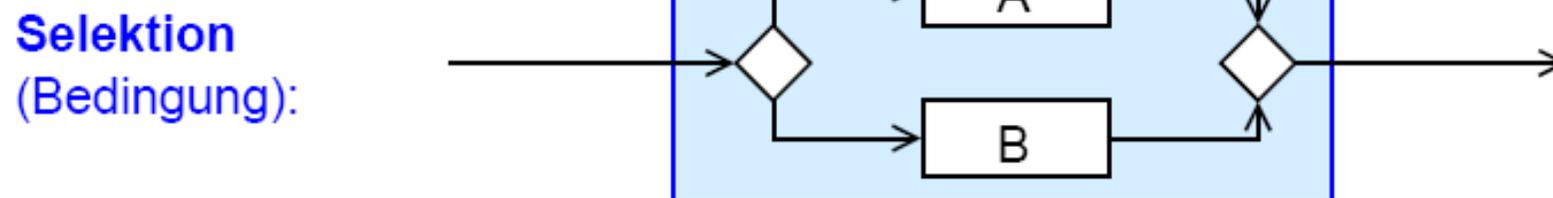
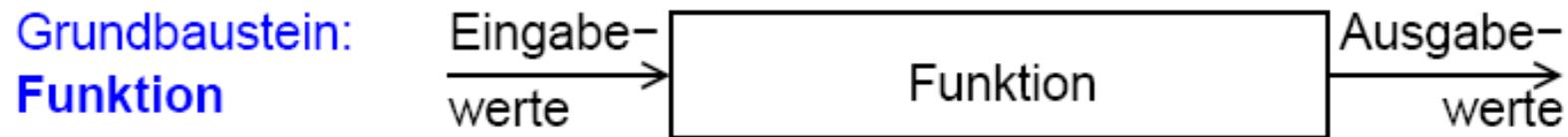
```
int i;  
int fak = 1;  
for (i=1; i<=n; i++)  
    fak = fak * i;
```

Als Funktionsgleichung:

$$\text{fak}(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot \text{fak}(n-1) & \text{falls } n > 0 \end{cases}$$



Strukturen funktionaler Programme





7.2 SML: Überblick

- SML = Standard ML = *Standard Meta Language*
 - ML wurde 1973 als Teil eines Theorembeweislers entwickelt
 - seither viele Dialekte, 1984 "standardisierte" Version SML
 - Referenzimplementierung: "SML of New Jersey" (SML/NJ)
 - interaktiver Compiler für SML
 - frei erhältlich für Windows und Linux (<http://www.smlnj.org/>)
- Eigenschaften von SML
 - streng getypte funktionale Sprache
 - polymorphes Typsystem
 - Syntax nahe an mathematischer Notation
 - enthält auch imperative Konstrukte (in der Vorlesung nicht behandelt)





7.2 SML: Überblick ...

Interaktiver Compiler SML/NJ

- Start mit Kommando `sml`
- Ausgabe des Compilers (auf Folien rot und kursiv):
Standard ML of New Jersey v110.57 [built: Wed Feb ...] -
- Das Promptzeichen `-` zeigt, daß der Compiler eine Eingabe erwartet
 - abgeschlossen mit `;` und Enter-Taste
- Beispiel:
- 5 + 10;
val it = 15 : int
 - *it* ist eine Variable (vom Typ `int`), die das Ergebnis der letzten Berechnung (15) bezeichnet



7.2 SML: Überblick ...

Interaktiver Compiler SML/NJ ...

- Das Promptzeichen = zeigt unvollständige Eingabe an

- Beispiel:

- 5

= + 10;

val it = 15 : int

- Eine Eingabe kann auch durch Drücken von Control-C abgebrochen werden
- Der Compiler wird durch Drücken von Control-D auf der obersten Ebene (Prompt: -) beendet

- ~5

= + 10;

val it = 5 : int



7.2 SML: Überblick ...

- Eingaben können sein:
 - Vereinbarungen von Variablen für Werte (einschließlich Funktionen),
z.B. `val x = 42 (-x; val it = 42 : int)` ; oder `fun f(x) = x+1; (val f = fn : int -> int)`
`(- f(10); val it = 11 : int)`
 - Ausdrücke, z.B. `x - 40`; oder `f x`;
 - Sie werden jeweils durch Semikolon voneinander getrennt
 - Der Compiler prüft die Eingaben auf korrekte Syntax, übersetzt sie und führt sie ggf. sofort aus
 - Die Ausgabe des Compilers ist immer eine Liste der vereinbarten Variablen (mit Angabe von Wert und Typ)
 - `val x = 42; val y = 5;`
 - `val x = 42 : int` `val y = 5 : int`
 - spezielle Variable ist für Ergebnis des letzten Ausdrucks
-



7.2 SML: Überblick ...

- SML-Programme können auch in Dateien abgelegt werden

- Einlesen in den Compiler:

- *use "beispiel.sml";*

← *Datei enthält: val x = 42;*

[opening beispiel.sml]

val x = 42 : int

val it = () : unit

← *Ergebnis der Funktion use*

- x + 2;

val it = 44 : int

- Alternative: Aufruf des Compilers mit Dateiname

- *sml beispiel.sml*

- Syntax für Kommentare:

(Das ist ein Kommentar *)*



7.2 SML: Überblick ...

Fehlermeldungen des Compilers

- Beim Einlesen aus einer Datei:

- *use "beispiel.sml"*

[opening beispiel.sml]

beispiel.sml:1.6-1.11 Error: unbound variable or ...

↑ Ort des Fehlers: Zeile.Spalte - Zeile.Spalte

- die Datei enthielt 1234-hallo;

- Bei interaktiver Eingabe: Zeilennummern teilweise unsinnig

- *1234-hallo;*

stdIn:1.6-5.4 Error: unbound variable or constructor



7.3 Werte und Ausdrücke

- **Werte** sind **Ausdrücke**, die nicht weiter ausgewertet werden können
 - einfache Werte: z.B. Zahlen, Zeichenketten, ...
 - konstruierte Werte: z.B. Tupel, Listen, ...
 - Funktionen, mit der Besonderheit: sie können auf andere Werte angewandt werden
- Alle Werte haben in SML einen eindeutig bestimmten Typ
- Werte, die keine Funktionen sind oder enthalten, heißen **Konstante**
- Aus Werten (incl. Funktionen) können **Ausdrücke** geformt werden, die bei der Auswertung auf Werte reduziert werden
 - d.h. Auswertung im Sinne mathematischer Vereinfachung



7.3 Werte und Ausdrücke ...

Ganze Zahlen (int)

- Übliche Darstellung, negative Zahlen aber mit vorangestellter Tilde (~), z.B. 13, ~5
- Vordefinierte Funktionen auf ganzen Zahlen:
 - binäre Operationen: +, -, *, div, mod
 - zweistellige Funktionen in Infixschreibweise
 - vom Typ $\text{int} * \text{int} \rightarrow \text{int}$ (math.: $Z \times Z \rightarrow Z$)
 - unäre Operationen: ~, abs (Negation, Absolutbetrag)
 - einstellige Funktionen in Präfixschreibweise
 - vom Typ $\text{int} \rightarrow \text{int}$ (math.: $Z \rightarrow Z$)
- * und -> sind **Typkonstruktoren**
 - zur Bildung neuer Typen (kartesisches Produkt bzw. Funktionstyp) aus vorhandenen Typen



7.3 Werte und Ausdrücke ...

Zur Funktionsapplikation

- Die Applikation hat höchste syntaktische Priorität
 - d.h. `abs 4-5` bedeutet `(abs 4) - 5`
 - die Klammern bei z.B. `f(x+y)` sind normale Klammern um den Ausdruck `x+y`, auf dessen Wert `f` angewandt wird
 - Die Applikation ist **linksassoziativ**
 - d.h. `f g x` bedeutet `(f g) x`
 - Beispiel:
 - `~ abs(4-5);`
 - stdIn:1.1 Error: overloaded variable not defined at type*
 - symbol: ~*
 - type: int -> int*
 - Versuch, die Funktion `~` auf die Funktion `abs` anzuwenden
-



7.3 Werte und Ausdrücke ...

Reelle Zahlen (real)

- Übliche Darstellung, aber mit ~ statt -, z.B. 3.0, ~5E2, 0.5E~3
 - Die Operationen +, -, *, ~ und abs sind auch auf reellen Zahlen definiert (d.h. sie sind überladen)
 - Division: / (Typ: real * real -> real)
 - Umwandlung zwischen int und real:
 - real: int -> real
 - floor: real -> int größte ganze Zahl \leq Argument (- floor 3.4; / val it = 3 : int)
 - Keine implizite Typumwandlung:
 - 3.0 * 4;
 - stdIn:1.1-6.3 Error: operator and operand don't agree*
 - operator domain: real * real*
 - operand: real * int*
-



7.3 Werte und Ausdrücke ...

Zeichenketten (string) und Zeichen (char)

- Übliche Darstellung für Strings, z.B. "Ein String" (val it = „Ein String“ : string)
 - Darstellung für Zeichen: #"a"
 - Funktionen für Strings:
 - ^: string * string -> string Konkatenation
 - size: string -> int Länge
 - substring: string * int * int -> string
 - Teilstring, Argumente: Startposition (ab 0) und Länge
 - Beispiel:
 - substring ("abcd" ^ "efgh", 2, 4);
val it = "cdef" : string
 - substring("ab",2,1);
uncaught exception Subscript [subscript out of bounds]
-



7.3 Werte und Ausdrücke ...

Wahrheitswerte (bool)

- Konstanten: true, false
- Operationen: not (Negation), andalso (Und), orelse (Oder)
 - das zweite Argument von andalso bzw. orelse wird nur ausgewertet, falls notwendig
- Vergleichsoperationen (mit Ergebnis vom Typ bool):
 - =, <> für int, char, string und bool
 - <, <=, >, >= für int, real, char und string
- Fallunterscheidung (ternäre Funktion): if ... then ... else ...
 - `val n = 2;`
 - val n = 2 : int*
 - `(if n<>0 then 100 div n else 0) + 10;`
 - val it = 60 : int*



7.4 Tupel, Records und Listen

- Tupel, Records und Listen fassen mehrere Werte zu einer Einheit zusammen
- **Tupel**
 - feste Zahl von Werten, auch mit unterschiedlichen Typen
 - Zugriff auf Komponenten über Positionsindex
- **Record:**
 - feste Zahl von Werten, auch mit unterschiedlichen Typen
 - Zugriff auf Komponenten über beliebige Identifikatoren oder ganze Zahlen
 - d.h. Tupel sind spezielle Records
- **Liste**
 - beliebige, variable Zahl von Werten **desselben** Typs



7.4 Tupel, Records und Listen ...

7.4.1 Tupel

- Schreibweise / Konstruktion von Tupeln:
 - ([<Ausdruck> {, <Ausdruck>}])
- Beispiele:
 - (1-1, true, 5.0, 2<1);
*val it = (0,true,5.0,false) : int * bool * real * bool*
 - (2);
val it = 2 : int
- einstellige Tupel bilden keinen eigenständiger Typ
 - ();
val it = () : unit
 - das nullstellige Tupel hat den Typ unit, der () als einzigen Wert besitzt



7.4.1 Tupel ...

Selektion von Komponenten

- Komponenten eines Tupels können über ihre Position selektiert werden (Zählung ab 1)
 - Operator # `<IntKonstante> <Ausdruck>`
- Beispiele:
 - #1 (1-1, true, 5.0, 2<1);
val it = 0 : int
 - #1 (#2 (1.1,(2,3.3)));
val it = 2 : int
 - Tupel können auch wieder Tupel enthalten
 - #3 (1,2);
stdIn:15.1-15.9 Error: operator and operand don't agree
 - Compiler prüft Zulässigkeit des Selektors



7.4 Tupel, Records und Listen ...

7.4.2 Records

- Schreibweise / Konstruktion von Records:

- { [<Name> = <Ausdruck> {, <Name> = <Ausdruck>}] }

- Beispiele:

- {Name="Joe",Age=35};

- val it = {Age=35,Name="Joe"} : {Age:int, Name:string}*

- die Reihenfolge der Komponenten spielt keine Rolle, die Komponentennamen gehören mit zum Typ

- {2=7, true=false};

- val it = {2=7,true=false} : {2:int, true:bool}*

- Komponentennamen können beliebige Identifikatoren oder ganze Zahlen sein



7.4.2 Records ...

➤ Beispiele...:

- $\{2=9, 1=35, 3="hallo"\} = (35, 9, "hallo");$

val it = true : bool

➤ Records mit Komponentennamen 1...n werden als Tupel interpretiert

Selektion von Komponenten

➤ Analog zu Tupeln über den Operator #

➤ Beispiele:

- $\#Pos \{Ort="Hagen", Pos=(1.0, 2.3)\};$

*val it = (1.0, 2.3) : real * real*

- $\#r (\#2 (3, \{x=1, r=4\}));$

val it = 4 : int



7.4 Tupel, Records und Listen ...

7.4.3 Listen

- Schreibweise / Konstruktion von Listen:

[[<Ausdruck> {, <Ausdruck>}]]

- Beispiele:

- [1,2,3,4];

val it = [1,2,3,4] : int list

- [1,3.0];

stdIn:29.1-29.8 Error: operator and operand don't agree

- alle Listenelemente müssen denselben Typ haben

- [];

val it = [] : 'a list

- leere Liste: der Typ enthält eine freie Typvariable (s. später)

- alternativ auch nil statt [] / (- nil ;)



7.4.3 Listen ...

Operationen auf Listen

- Erstes Element der Liste (**hd** / head) und Restliste (**tl** / tail):

- `hd [1,2,3];`

- `val it = 1 : int`*

- `tl [1,1+1];`

- `val it = [2] : int list`*

- `tl [[1,2],[3],[]];`

- `val it = [[3],[]] : int list list`*

←Ergebnis ist immer Liste!

←Liste von Listen

- Anfügen am Anfang der Liste: `::`

- `1 :: [2, hd[3]];`

- `val it = [1,2,3] : int list`*

- Konkatenation zweier Listen: `@`

- `[1,2] @ [3,4];`

- `val it = [1,2,3,4] : int list`*



7.4.3 Listen ...

Operationen auf Listen ...

- Umkehren einer Liste: rev (reverse bei LISP)

- rev [1,2,3,4];

- val it = [4,3,2,1] : int list*

- hd (rev [1,2,3,4]);

- val it = 4 : int*

←letztes Element der Liste

- Umwandlung von string nach char list: explode

- explode "Bombe";

- val it = [#"B",#"o",#"m",#"b",#"e"] : char list*

- Umwandlung von char list nach string: implode

- implode (rev (explode "Bombe"));

- val it = "ebmoB" : string*



7.5 Variablen und Funktionen

7.5.1 Variablen

- Eine **Variable** ist ein Bezeichner für einen Wert
- Die Zuordnung eines Werts zu einem Bezeichner heißt **Bindung**
 - Bindung ist Paar (Bezeichner, Wert)
- Die Menge der aktuell existierenden Bindungen heißt **Umgebung**
- Die (Werte-)Definition `val <Variable> = <Ausdruck>` erzeugt eine neue Variablen-Bindung
 - der Wert des Ausdrucks wird an die Variable gebunden
- Beispiel:
 - `val Name = "Mi" ^ "In" ^ "er";`
 - `val Name = "Milner" : string`*



7.5.1 Variablen ...

Mehrfache Bindung

- Eine Variable kann nacheinander an verschiedene Werte (auch verschiedenen Typs) gebunden werden:

- `val bsp = 1234;`

- `val bsp = 1234 : int`

- `bsp + 1;`

- `val it = 1235 : int`

- `val bsp = ("Hallo", "Welt");`

- `val bsp = ("Hallo","Welt") : string * string`

- `#1 bsp;`

- `val it = "Hallo" : string`

- auch die Variable `it` wird hier mehrfach gebunden

- Die Umgebung wird somit durch Definitionen verändert

- Beachte: die Werte haben einen Typ, nicht die Variablen!
-

7.5.1 Variablen ...

Pattern Matching

- Eine Definition kann auch mehrere Variablen auf einmal binden
 - linke und rechte Seite dürfen komplexe Werte mit gleicher Struktur sein
 - Tupel, Records, Listen und selbst definierte Datentypen
 - die Bindung der Variablen ergibt sich dann als Lösung der (einfachen) mathematischen Gleichung

- Beispiel:

- val (x,y) = (3,2.0);

val x = 3 : int

val y = 2.0 : real

- val {a=u,2=v} = {2="Hallo",a=43};

val v = "Hallo" : string

val u = 43 : int



7.5 Variablen und Funktionen ...

7.5.2 Funktionen

- Funktionen (als Werte) werden in SML wie folgt dargestellt:
 - `fn <Variable> => <Ausdruck>`
- Beispiel: `fn x => 2 * x` ist die Funktion, die jeder Zahl `x` ihr Doppeltes zuordnet
- Ein solcher Funktions-Wert kann auf andere Werte angewandt werden:
 - `(fn x => 2 * x) 5;`
val it = 10 : int
- Er kann wie jeder andere Wert verwendet werden
 - Bindung an Namen
 - Speichern in Tupeln, Records oder Listen
 - Argument oder Ergebnis von Funktionen



7.5.2 Funktionen ...

Binden von Funktionswerten an Namen (Funktionsdeklaration)

- Syntax genau wie bei anderen Werten, z.B.:
 - `val dbl = fn x => 2 * x;`
 - val dbl = fn : int -> int*
- Compiler gibt statt des Werts nur `fn` aus
- Der Typ (hier: `int -> int`) wird automatisch aus der Funktionsgleichung (`x => 2 * x`) ermittelt (**Typinferenz**)
- Abkürzende Schreibweise:
 - `fun <Variable1> <Variable2> = <Ausdruck>`
 - `<Variable1>`: Funktions-Bezeichner, `<Variable2>`: Argument
 - im Beispiel:
 - `fun dbl x = 2 * x;`
 - val dbl = fn : int -> int*



7.5.2 Funktionen ...

Applikation von Funktionen

➤ Über den Funktions-Bezeichner:

- `dbl 5;`

val it = 10 : int

- `dbl (5+5);`

val it = 20 : int

➤ Direkte Anwendung eines Funktions-Werts auf einen anderen

Wert:

- `(fn x => 2 * x) ((fn x => x + 1) 5);`

val it = 12 : int



7.5.2 Funktionen ...

Typrestriktion

- Eine Funktion zum Quadrieren:
 - fun square x = x * x;
 - val square = fn : int -> int*
- Warum hat diese Funktion den Typ int -> int und nicht real -> real?
 - der Operator * ist überladen für int und real
 - die Typinferenz kann damit den notwendigen Typ des Arguments nicht eindeutig bestimmen
 - SML wählt dann den *Default*-Typ, hier int
- SML erlaubt aber auch, den Typ eines Ausdrucks zu erzwingen (**Typrestriktion**)



7.5.2 Funktionen ...

Typrestriktion ...

➤ Beispiele

```
- val x = 3 : int;  
val x = 3 : int  
- val x = 3 : real;  
stdin:... Error: expression doesn't match constraint  
  
- fun square x : real = x * x;      ← square x ist real  
val square = fn : real -> real  
- fun square (x : real) = x * x;    ← x ist real  
val square = fn : real -> real  
- fun square x = x * x : real;      ← x * x ist real  
val square = fn : real -> real  
- fun square x = x * (x : real);    ← x ist real  
val square = fn : real -> real
```



7.5.2 Funktionen ...

Funktionen mit mehreren Argumenten

- Eine (mathematische) Funktion hat genau ein Argument und genau ein Resultat
- Eine Funktion mit mehreren Argumenten ist genau betrachtet eine Funktion auf einem Tupel:

- fun minimum (x,y) = if x<y then x else y;

*val minimum = fn : int * int -> int*

- Ebenso kann eine Funktion auch ein Tupel von Werten als Resultat liefern:

- fun DivMod (a,b) = (a div b, a mod b);

*val DivMod = fn : int * int -> int * int*

- DivMod (9,4);

*val it = (2,1) : int * int*



7.5.2 Funktionen ...

Pattern Matching

- Pattern Matching ist auch bei Funktionsargumenten möglich
- Dabei können mehrere alternative Muster angegeben werden
- Dies erlaubt z.B. die Funktionsdefinition durch Aufzählung:

```
- fun f 0 = 0  
= | f 1 = 2  
= | f 2 = 3;
```

stdIn:27.5-29.10 Warning: match nonexhaustive

```
val f = fn : int -> int
```

- Warnung, da Funktion nicht für alle int-Werte definiert wird
 - Auch möglich: Ausnahmefälle und allgemeiner Fall
- ```
- fun f 0 = 0
= | f n = n + 1;
```

←wird zuerst geprüft

←falls n ≠ 0



## 7.5.2 Funktionen ...

---

### Pattern Matching: Weitere Beispiele

- Eine Funktion muß jedoch einen wohldefinierten Typ haben:

- fun f (x,y) = x + y
  - = | f (x,y,z) = x + y + z;

*stdIn:1.5-59.26 Error: parameter or result constraint of clauses don't agree [tycon mismatch]*

- der Typ kann nicht gleichzeitig  $\text{int} * \text{int} \rightarrow \text{int}$  und  $\text{int} * \text{int} * \text{int} \rightarrow \text{int}$  sein

- Beispiel: Fakultätsfunktion (rekursive Funktion)

- fun fak 0 = 1
  - = | fak n = n \* fak(n-1);

*val fak = fn : int -> int*

- fak 10;

*val it = 3628800 : int*



## 7.5.2 Funktionen ...

---

### Pattern Matching: Funktionen auf Listen

➤ Länge einer Liste:

- fun len [] = 0

←leere Liste

= | len (x::rest) = 1 + len rest;

←Liste x :: rest

*val len = fn : 'a list -> int*

➤ die Klammern um x::rest sind notwendig

➤ die Funktion kann auf Listen beliebigen Typs ('a list) angewandt werden (**polymorphe Funktion**):

- len [3,3,2,2];

*val it = 4 : int*

- len ["hallo", "welt"];

*val it = 2 : int*

- len [[],[1,2,3],[5,2]];

*val it = 3 : int*



## 7.5.2 Funktionen ...

---

### *Pattern Matching: Funktionen auf Listen ...*

Umkehren einer Liste:

```
- fun rev [] = []
= | rev (x::rest) = rev(rest) @ [x];
val rev = fn : 'a list -> 'a list ← polymorphe Fkt.
- rev [1,2,3,4];
val it = [4,3,2,1] : int list
```

Sortiertes Einfügen in eine Liste:

```
- fun insert (x, []) = [x]
= | insert (x, first::rest) = if (x >= first)
= then first :: insert(x, rest)
= else x :: first :: rest;
val insert = fn : int * int list -> int list
- insert(5, insert(3, insert(4, insert(2, []))));
val it = [2,3,4,5] : int list
```

---



## 7.5.2 Funktionen ...

---

### *Pattern Matching: Funktionen auf Listen ...*

➤ Ein Operator zum sortierten Einfügen in eine

```
- infixr 5 ++;
infixr 5 ++
```

```
- fun x ++ [] = [x]
= | x ++ (first :: rest) = if x >= first
= then first :: x ++ rest
= else x :: first :: rest;
val ++ = fn : int * int list -> int list
- 3 ++ 5 ++ 1 ++ 7 ++ [4];
val it = [1,3,4,5,7] : int list
```



## 7.5.2 Funktionen ...

---

### *Pattern Matching: Funktionen auf Listen ...*

#### ➤ Sortiertes Einfügen in eine Liste von

```
- fun insert (x, []) = [x:real]
= | insert (x, first::rest) = if (x >= first)
= then first :: insert(x, rest)
= else x :: first :: rest;
val insert = fn : real * real list -> real list
```

```
- fun glen [] = 0
= | glen (x::rest) = len x + glen rest;
val glen = fn : 'a list list -> int
```

➤ 

```
- glen [[], [1,2,3], [5,2]];
val it = 5 : int
```



## 7.5.2 Funktionen ...

---

### Statisches Binden

- Was passiert mit `glen`, wenn wir `len` neu definieren?
  - `fun len x = 0;`
  - `val len = fn : 'a -> int`*
  - `glen [[],[1,2,3],[5,2]];`
  - `val it = 5 : int`*
- Die neue Bindung für `len` hat keinen Einfluß auf `glen`
- Maßgeblich für die Semantik einer Funktion ist die Umgebung (d.h. die Bindungen) zum Zeitpunkt ihrer **Definition**, nicht die zum Zeitpunkt ihrer Auswertung (**statisches Binden**)
  - Eigenschaft fast aller funktionaler Sprachen
- Eine einmal definierte Funktion verhält sich damit bei jedem Aufruf gleich



## 7.5.2 Funktionen ...

---

### Freie Variablen in Funktionsdefinitionen

- Funktionsgleichungen können auch Variablen enthalten, die keine Argumente sind (**freie Variablen**)
  - diese Variablen müssen aber an ein Wert gebunden sein
  - auch hier wird statisches Binden verwendet

- Beispiel:

- val pi = 3.14159265;

- val pi = 3.14159265 : real**

- fun area r = pi \* r \* r;

←pi ist freie Variable

- val area = fn : real -> real**

- val pi = 0.0;

←neue Bindung für pi

- val pi = 0.0 : real**

- area 2.0;

←verwendet Bindung zum

- val it = 12.5663706 : real**

Zeitpunkt der Def. v. area



## 7.5.2 Funktionen ...

---

### Lokale Definitionen

- Manchmal sollen Definitionen nur lokal in einem Ausdruck gelten
  - z.B. Einführen einer Variable als Abkürzung für einen Term
- SML bietet dazu let-Ausdrücke an:
  - `let <Deklarationsfolge> in <Ausdruck> end`
  - die lokalen Deklarationen verändern die Umgebung außerhalb des let-Ausdrucks nicht!

- Beispiel:

- `val x = 1;`

- `val x = 1 : int`

↓ *x aus der Umgebung (mit Wert 1)*

- `val res = let val x = x+1 in x * x end;`

- `val res = 4 : int`

↑ *lokal definiertes x (= 2)*

- `x;`

← *dieses x blieb unberührt*

- `val it = 1 : int`



## 7.5.3 Typen und Polymorphismus

---

- In funktionalen Sprachen: Typsystem hat hohen Stellenwert
- Strenge Typisierung: jeder Wert hat einen eindeutigen Typ
  - in imperativen Sprachen meist abgeschwächte Typsysteme, die Uminterpretierung von Typen erlauben, z.B. durch:
    - Typkonversion
    - generische Typen wie `void *` in C oder `Object` in Java, um generische Funktionen zu realisieren
- In funktionalen Sprachen stattdessen flexible Typsysteme
  - Typen von Variablen (inkl. Funktionen) können oft automatisch ermittelt werden: **Typinferenz**
  - Konzepte wie generische Funktionen sind sinnvoll in das Typsystem integriert: **Typpolymorphismus**



## 7.5.3 Typen und Polymorphismus ...

---

### Eigenschaften des SML Typsystems

- Der Typ eines Ausdrucks kann allein aus der syntaktischen Struktur ermittelt werden
  - statische Typprüfung zur Übersetzungszeit
  - keine Laufzeit-Typfehler möglich (vgl. Java!)
    - schnellerer und sichererer Code
- Das Typsystem unterstützt polymorphe Typen
  - Typen können freie Variablen (z.B. ``a`) enthalten
  - Funktionen können für eine ganze Klasse von Typen definiert werden
    - dies erhöht die Wiederverwendbarkeit der Software



## 7.5.3 Typen und Polymorphismus ...

---

### Typausdrücke

- Das Typsystem in SML bildet eine eigene Sprache mit Ausdrücken, die auch an Variable gebunden werden können
- Die Konstanten sind die einfachen Typen:
  - `unit`, `bool`, `int`, `real`, `char`, `string`
- Operationen (**Typkonstruktoren**):
  - Tupel, z.B. `int * int`
  - Records, z.B. `{a:int, b:string}`
  - Listen, z.B. `real list`
  - Funktionstypen, z.B. `string -> int`
- Binden an Variable: `type <Variable> = <Typausdruck>`
  - Beispiel: `type point = real * real`



## 7.5.3 Typen und Polymorphismus ...

---

### Parametrischer Polymorphismus

- Abstraktionsmechanismus für Typausdrücke:
  - durch Variablen in Typausdrücken kann man Typen mit gegebener **Struktur** beschreiben
- Beispiele:
  - `int * string, bool * real` etc. sind alles Paare
  - `int list, real list, (int * bool) list` etc. sind alles Listen
- In einem Typausdruck steht eine **Typvariable**, z.B. `' a` oder `' b` für einen beliebigen Typ
  - vorgestellter Apostroph zur syntaktischen Unterscheidung
- Typvariablen mit zwei Apostrophen (z.B. `' ' a`) stehen für beliebige Typen, auf denen Gleichheit definiert ist



## 7.5.3 Typen und Polymorphismus ...

---

### Parametrischer Polymorphismus ...

- Die Menge aller Paar-Typen ist damit: `'a * 'b`
- Menge aller Listen-Typen: `'a list`
- Menge aller Funktionstypen, die zu einer Liste einen Wert ihres Elementtyps liefern: `'a list -> 'a`
- Definition eines polymorphen Typs: die Bindung der Typvariablen erfolgt durch Auflisten vor dem zu definierenden Typ:
  - `type 'a idPair = 'a * 'a;`
  - `type ('a,'b) pairList = ('a * 'b) list;`
- **Instanziierung** eines Typs: Angabe von Werten für Typvariablen
  - `(2,2) : int idPair;`
  - `[(1, "foo"), (2, "bar")] : (int, string) pairList;`



## 7.5.3 Typen und Polymorphismus ...

---

### Polymorphe Funktionen

- Parametrischer Polymorphismus erlaubt die typsichere Definition generischer Funktionen

- Beispiel: erstes Element eines Tupels

```
- fun first (a,b) = a;
val first = fn : 'a * 'b -> 'a
```

- Beispiel: Länge einer Liste

```
- fun length l = if l=[] then 0 else 1 + length(tl l);
val length = fn : ''a list -> int
```

- Argument vom Typ ''a list, da der Vergleich zweier Listen auf dem Vergleich der Elemente basiert

```
- fun length [] = 0
= | length (hd::tl) = 1 + length(tl);
val length = fn : 'a list -> int
```



## 7.5.3 Typen und Polymorphismus ...

---

### Typinferenz: Wie bestimmt man den Typ eines Ausdrucks?

- Beispiel: `fun f (x, y) = x + 1 = y;`
  - Starte mit dem allgemeinsten möglichen Typ für jedes Element des Ausdrucks:
    - `type(x) = 'a, type(y) = 'b, type(1) = int,`  
`type(f) = 'c -> 'd`
  - Füge Gleichungen hinzu, die sich aus der Struktur des Ausdrucks ergeben und löse das Gleichungssystem:
    - aus `x+1` folgt `'a = type(1)` und `type(x+1) = 'a`
    - aus `x+1=y` folgt `'b = type(x+1)` und `type(x+1=y) = bool`
    - aus `(x, y)` folgt `type((x, y)) = 'a * 'b`
    - aus `fun f(x, y) = x+1=y` folgt: `'c = type((x, y))` und `'d = type(x+1=y)`
  - Lösung: `type(f) = int * int -> bool`
-

# Zur Klausur OFP im SS 2013

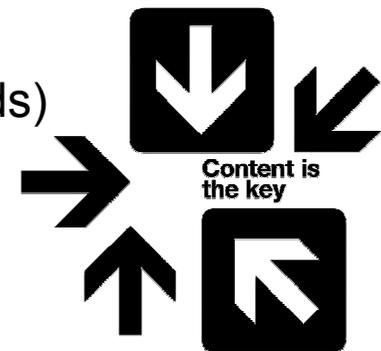


- 1. Termin: **Mo., 05.08.2013, 10:15 - 12:15 Uhr**  
AR/E/1-8101 - Audimax
- 2. Termin: **Fr., 20.09.2013, 10:15 - 12:15 Uhr**  
AR/E/2-9202/03 – Turnhalle



## Aufteilung des Stoffes:

- ca. 20% UML
- ca. 60% Java-Programmierung
  - u.a. Vererbung, Exceptions, Collections, GUIs, (Threads)
- ca. 5-10% Entwurfsmuster
- ca. 10-15% SML-Programmierung
  - u.a. Typen, Funktionen





## Fragestellungen:

- UML-Klassendiagramme (OOA / OOD) zeichnen
- Umsetzung von UML nach Java oder umgekehrt
- einfache Programmieraufgaben  
i.a. als Lückentext (wenige Zeilen)
- vorgegebene Programme verstehen  
Was ist die Ausgabe?  
Wo sind Fehler im Programm?
- Bei SML: Typ/Wert von Ausdrücken/Funktionen
- Wissensfragen zum Ankreuzen
- **Musterklausur:** siehe Duesie





## 7.6 Datentypen und Pattern Matching

---

- Typen, die mit `type` an Variablen gebunden werden, stellen lediglich abkürzende Schreibweisen dar
  - innerer Aufbau der Datenstruktur ist offengelegt
- Eine SML `datatype` Deklaration definiert einen Typ zusammen mit seinen **Konstruktoren**
  - spezifiziert wird nur, wie Werte dieses Typs **erzeugt** werden, nicht wie sie gespeichert oder dargestellt werden
  - tatsächlich wird in SML nie eine Implementierung für die Konstruktoren angegeben!
  - SML-Funktionen arbeiten lediglich auf der syntaktischen Struktur des Terms, der eine Datenstruktur erzeugt hat
    - Berechnung basiert auf der **Termalgebra**



## 7.6 Datentypen und Pattern Matching ...

---

### Variantentypen

- Ein Datentyp besteht aus einer Menge typisierter Konstruktoren, von denen jeder eine Variante des Typs beschreibt

- Syntax für die Definition eines Datentyps:

```
datatype [<Typvariablen>] <Typname> =
 <Konstruktor> [of <Typ>]
 { | <Konstruktor> [of <Typ>] }
```

- `<Typvariablen>` definiert die auf der rechten Seite vorkommenden Typvariablen bei polymorphen Typen
  - z.B. `datatype 'a tree = ...`
- `<Typ>` gibt den Argumenttyp des jeweiligen Konstruktors an
  - das Ergebnis ist immer vom definierten Typ `<Typname>`



## 7.6 Datentypen und Pattern Matching ...

---

### Variantentypen ...

- Beispiel: Datentyp für geometrische Objekte

```
type point = real * real;
datatype geo = POINT of point
 | CIRCLE of point * real
 | RECT of {lowLeft:point, upRight:point};
```

- (vgl. mit dem Ansatz der objektorientierten Programmierung!)
- Erzeugung von Werten des Datentyps **geo**:
  - `val p = POINT (1.0,2.0);`
  - `val p = POINT (1.0,2.0) : geo`
  - `val c = CIRCLE ((2.0, 3.5), 1.0);`
  - `val c = CIRCLE ((2.0,3.5),1.0) : geo`
- `POINT (1.0,2.0)` und `CIRCLE ((2.0,3.5),1.0)` sind Werte!
  - sie werden als **Terme** bezeichnet



## 7.6 Datentypen und Pattern Matching ...

---

### Pattern Matching

- Auf die Varianten eines Datentyp-Werts kann wieder durch *Pattern Matching* zugegriffen werden
  - ein Muster ist aus Konstruktoren und Variablen gebildet
  - es passt auf einen Term derselben Struktur
  - die Variablen im Muster werden dann an die Argumente der entsprechenden Konstruktoren im Term gebunden
- Beispiel:
  - `val CIRCLE (m, r) = c; ←c wie auf voriger Folie gebunden`
  - `stdIn:26.5-26.21 Warning: binding not exhaustive`
  - `CIRCLE (m, r) = ...`
  - `val m = (2.0, 3.5) : real * real`
  - `val r = 1.0 : real`
  - Warnung, da Muster nicht alle Varianten von `geo` abdeckt



## 7.6 Datentypen und Pattern Matching ...

---

### Pattern Matching: Beispiel

- Berechnung des Flächeninhalts eines geometrischen Objekts:
  - fun area (POINT \_) = 0.0
  - = | area (CIRCLE (\_, r)) = 3.1415926 \* r \* r
  - = | area (RECT {lowLeft=(x1,y1), upRight=(x2,y2)}) =
  - = abs ((x1-x2) \* (y1-y2));
  - val area = fn : geo -> real
  - area (CIRCLE((0.0,0.0),1.0));
  - val it = 3.1415926 : real
- In Mustern kann anstelle einer Variable ein Unterstrich ( \_ ) als anonyme Variable (*wildcard*) auftreten
  - für diese "Variable" wird keine Bindung erzeugt
- Funktionsdefinitionen verlaufen mit Hilfe von Mustern "entlang" der Definition der Datentypen

## 7.6 Datentypen und Pattern Matching ...

### Rekursive Datentypen

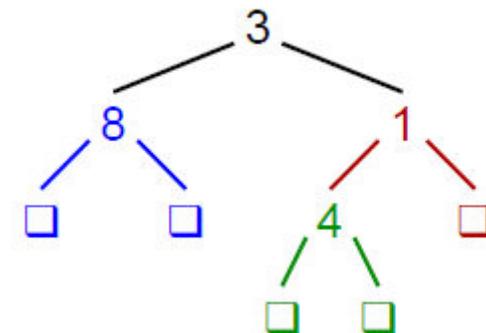
- Bei der Definition der Varianten darf auch der gerade definierte Datentyp selbst verwendet werden
  - dies führt zu **rekursiven Typdefinitionen** und damit zu rekursiven Datenstrukturen
- Beispiel: ein binärer Baum ganzer Zahlen

```
datatype tree = EMPTY
 | NODE of int * tree * tree;
```

- Der Term

```
NODE (3, NODE (8, EMPTY, EMPTY) ,
 NODE (1, NODE (4, EMPTY, EMPTY) ,
 EMPTY))
```

entspricht dann dem rechts stehenden Baum





## 7.6 Datentypen und Pattern Matching ...

---

### Bäume als polymorphe Datentypen

- Die Knoten eines Binärbaums können im Prinzip beliebige Daten enthalten
- Modellierung in SML durch einen polymorphen Datentyp:
  - `datatype 'a tree = EMPTY`
  - = `| NODE of 'a * 'a tree * 'a tree;`
  - datatype 'a tree = EMPTY | NODE of 'a \* 'a tree \* 'a tree*
  - `val t1 = NODE(3, EMPTY, EMPTY);`
  - val t1 = NODE (3, EMPTY, EMPTY) : int tree*
  - `val t2 = NODE("Hallo", EMPTY, EMPTY);`
  - val t2 = NODE ("Hallo", EMPTY, EMPTY) : string tree*
- Damit können auch generische, polymorphe Funktionen für Bäume definiert werden



## 7.6 Datentypen und Pattern Matching ...

---

### Einige generische Funktionen auf Bäumen

➤ Anzahl der Knoten eines Baums:

```
- fun nodes EMPTY = 0
= | nodes (NODE (_, l, r)) = 1 + nodes l + nodes r;
val nodes = fn : 'a tree -> int

- nodes EMPTY;
val it = 0 : int

- (* Baum von Folie 64 *)
- val baum = NODE (3, NODE (8, EMPTY, EMPTY),
= NODE (1, NODE (4, EMPTY, EMPTY), EMPTY));
val baum = NODE (3, NODE (8, EMPTY, EMPTY), NODE (1, NODE #,
 EMPTY)) : int tree

- nodes baum;
val it = 4 : int
```



## 7.6 Datentypen und Pattern Matching ...

---

### Einige generische Funktionen auf Bäumen ...

- Beispiel zur Auswertung der Funktion nodes:

nodes (NODE (3, NODE (8, EMPTY, EMPTY), EMPTY) )  
⇒ 1 + nodes (NODE (8, EMPTY, EMPTY) ) + nodes EMPTY  
⇒ 1 + (1 + nodes EMPTY + nodes EMPTY) + 0  
⇒ 1 + (1 + 0 + 0) + 0  
⇒ 1 + (1 + 0) + 0 ⇒ 1 + 1 + 0 ⇒ 2 + 0  
⇒ 2

- (Die jeweils ausgewerteten Funktionen sind blau unterstrichen)
- Die Auswertung erfolgt durch Einsetzen der Werte für Argumente, Variablen und Funktionen
  - eingebaute Funktionen (wie + ) werden direkt ausgewertet

## 7.6 Datentypen und Pattern Matching ...

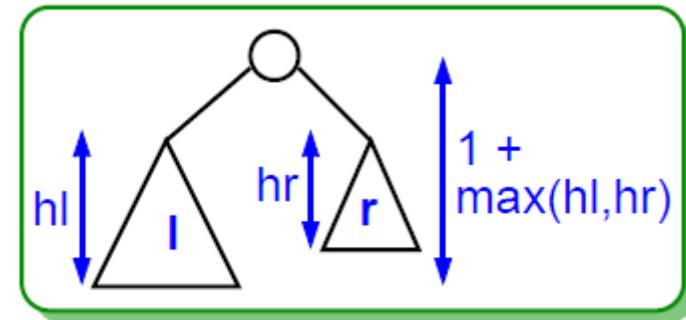
### Einige generische Funktionen auf Bäumen ...

#### ➤ Höhe eines Baums:

```

- fun height EMPTY = 0
= | height (NODE (_, l, r)) =
= let
= val hl = height l;
= val hr = height r;
= in
= 1 + (if hl > hr then hl else hr)
= end;

```



```

val height = fn : 'a tree -> int

```

- der bedingte Ausdruck berechnet das Maximum der Höhen von linkem und rechtem Unterbaum
- der let-Ausdruck verhindert deren doppelte Berechnung

# 7.6 Datentypen und Pattern Matching ...

## Einige generische Funktionen auf Bäumen ...

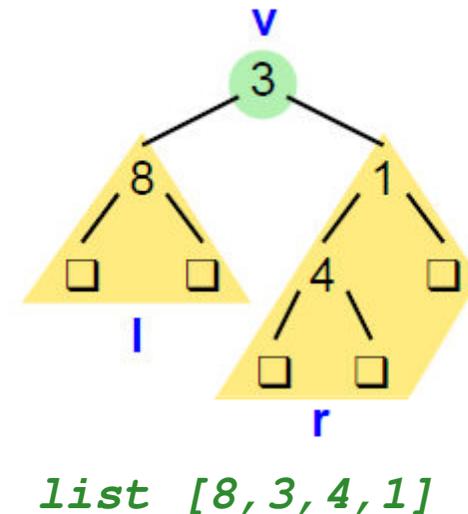
➤ Umwandlung eines Baums in eine Liste:

```

- fun inorder EMPTY = []
= | inorder (NODE (v, l, r)) = inorder l
= @ [v]
= @ inorder r;
val inorder = fn : 'a tree -> 'a list

- inorder baum;
val it = [8,3,4,1] : int list

```



➤ es wird immer erst der linke Unterbaum bearbeitet, dann der aktuelle Knoten, dann der rechte Unterbaum

➤ Inorder-Durchlauf durch den Baum



## 7.7 Funktionen höherer Ordnung

---

- Bisher haben wir Funktionen nur auf Datenwerte angewendet
- **Funktionen höherer Ordnung** sind Funktionen, die
  - andere Funktionen als Argument haben
  - oder Funktionen als Ergebnis liefern
- Sie sind daran erkennbar, daß ihr Typ mindestens **zweimal** den Typkonstruktor -> enthält
- Viele Operationen lassen sich mit Funktionen höherer Ordnung elegant und allgemein formulieren



## 7.7 Funktionen höherer Ordnung ...

---

### Beispiel: Filtern einer Liste

- Programmiere eine Funktion, die aus einer Liste alle Elemente ausfiltert, die eine bestimmte Bedingung erfüllen
  - die Bedingung wird der Funktion als Argument übergeben

```
- fun filter (p, l) = []
= | filter (p, x::l) = if p x
= then x :: filter (p, l)
= else filter (p, l);
val filter = fn : ('a -> bool) * 'a list -> 'a list
- val l = [1,2,3,4,5,6,7,8];
val l = [1,2,3,4,5,6,7,8] : int list
- filter (fn x => x mod 2 = 0, l); ← gerade Elemente
val it = [2,4,6,8] : int list
- filter (fn x => x > 4, l); ← Elemente > 4
val it = [5,6,7,8] : int list
```



## 7.7 Funktionen höherer Ordnung ...

---

### Currying

- Funktionen mit mehreren Argumenten haben wir bisher immer als Funktionen auf einem Tupel aufgefaßt:

```
- fun mult (a,b) = a * b;
val mult = fn : int * int -> int
- mult (2,3);
val it = 6 : int
```

- Wir können eine solche Funktion aber auch anders realisieren:
  - die Funktion hat nur ein Argument
  - sie liefert aber als Ergebnis eine **Funktion**, die auf das zweite Argument angewendet werden kann

- Der Aufruf sähe dann so aus:

```
- mult 2 3; ←Ergebnis von mult 2 ist eine Funktion!
val it = 6 : int
```



## 7.7 Funktionen höherer Ordnung ...

---

### Currying ...

- Die Umwandlung einer Funktion mehrerer Argumente in mehrere Funktionen mit einem Argument heißt **Currying**
  - nach dem Logiker Haskell B. Curry
- Mathematisch: Currying verwandelt eine Funktion des Typs

$$(A1 \times A2 \times \dots \times A_n) \rightarrow B$$

in eine Funktion des Typs

$$A1 \rightarrow A2 \rightarrow \dots \rightarrow A_n \rightarrow B$$

- d.h. die Anwendung dieser Funktion auf ein  $a1 \in A1$  liefert eine Funktion aus  $A2 \rightarrow \dots \rightarrow A_n \rightarrow B$



## 7.7 Funktionen höherer Ordnung ...

---

### Currying ...

- Wie kann eine solche Funktion in SML definiert werden?
- Anwendung des bisher Gelernten:
  - die Funktion, die ihr Argument mit einem Wert  $x$  multipliziert, ist
$$\text{fn } y \Rightarrow x * y$$
    - das  $x$  ist dabei eine freie Variable, deren Wert durch die Umgebung bestimmt wird

- also können wir schreiben:

```
- fun mult x = fn y => x * y;
val mult = fn : int -> int -> int
- mult 2 3; ← = (mult 2) 3 = (fn y => 2 * y) 3
val it = 6 : int
```



## 7.7 Funktionen höherer Ordnung ...

---

### Kurzschreibweise für *curried* Funktionen

#### ➤ Statt

```
fun <Variable> <Muster1> = fn <Muster2> => ... =>
 fn <Muster3> => <Ausdruck>
```

kann man auch kürzer schreiben:

```
fun <Variable> <Muster1> ... <Muster3> = <Ausdruck>
```

#### ➤ Eine Variable darf dabei immer nur in einem Muster auftreten

#### ➤ Beispiele:

```
- fun mult x y = x * y;
val mult = fn : int -> int -> int
- fun und false _ = false
= | und _ false = false;
= | und true true = true;
val und = fn : bool -> bool -> bool
```



## 7.7 Funktionen höherer Ordnung ...

### Bedeutung des Currying

- Currying ermöglicht die partielle Applikation von Funktionen
  - d.h. durch Fixierung eines Arguments erhält man eine neue Funktion, die bereits teilweise ausgewertet ist
- Beispiel: Summe von Funktionswerten  $f(1) + \dots + f(n)$

```
- fun sumf 1 f = f 1
= | sumf n f = sumf (n-1) f + f n;
val sumf = fn : int -> (int -> int) -> int
- sumf 5 (fn x => x * x); ← = 1 + 4 + 9 + 16 + 25
val it = 55 : int
- val sum2f = sumf 2; ← = fn f => f(1) + f(2)
val sum2f = fn : (int -> int) -> int
- sum2f (fn x => x); ← = 1 + 2
val it = 3 : int
```



## 7.7 Funktionen höherer Ordnung ...

---

### Bedeutung des Currying ...

➤ Auswertung des Ausdrucks `sumf 2`:

➤ ausführliche Definition der Funktion `sumf` ist:

```
fun sumf 1 = (fn f => f 1)
 | sumf n = (fn f => sumf (n-1) f + f n);
```

➤ damit wird `sumf 2` wie folgt ausgewertet:

```
sumf 2
=>fn f => sumf (2-1) f + f 2
=>fn f => sumf 1 f + f 2
=>fn f => (fn f => f 1) f + f 2
=>fn f => f 1 + f 2
```



## 7.7 Funktionen höherer Ordnung ...

---

### Oft genutzte Funktionen höherer Ordnung

➤ Filtern einer Liste (curried Version):

```
- fun filter p [] = []
= | filter p (x::l) = if p x
= then x :: filter p l
= else filter p l;
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

➤ Anwendung einer Funktion auf alle Elemente einer Liste:

```
- fun map f [] = []
= | map f (x::l) = f x :: map f l;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn x => x*x) [1,2,3,4]; ← quadriere die Elemente
val it = [1,4,9,16] : int list
```



## 7.7 Funktionen höherer Ordnung ...

---

### Oft genutzte Funktionen höherer Ordnung ...

- Reduzieren einer Liste mit einer binären Operation
  - z.B. Reduktion von `[1, 2, 3]` mit `+` liefert `1+2+3`
    - `fun foldr f u [] = u`
    - = `| foldr f u (x::l) = f (x, foldr f u l);`
    - `val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`
  - `foldr` reduziert die Liste von **rechts**, d.h. die Funktion `f` wird **rechtsassoziativ** angewandt
  - der Aufruf `foldr op+ 0 [1, 2, 3]` berechnet daher exakt
$$\text{op+}(1, \text{op+}(2, \text{op+}(3, 0))) = 1 + (2 + (3 + 0))$$
- Anmerkung: `op` erlaubt es einen Infix-Operator als Funktion (in Präfix-Notation) zu verwenden, z.B. `op+ (2, 3)`
- `op-` ?



## 7.7 Funktionen höherer Ordnung ...

---

### Oft genutzte Funktionen höherer Ordnung ...

➤ Funktionskomposition  $\circ$

- `infix 3  $\circ$` ;  $\leftarrow$ Infixschreibweise, Priorität 3

`infix 3  $\circ$`

- `fun (f  $\circ$  g) x = f(g(x));`

`val  $\circ$  = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b`

➤ Anmerkung:

- die Funktionen `map`, `foldr` und  `$\circ$`  sind in SML/NJ standardmäßig definiert



## 7.7 Funktionen höherer Ordnung ...

---

### Beispiele zur Anwendung der Funktionen

- Berechne die Länge für jeden String in einer Liste:

```
- map size ["Dies", "ist", "ein", "Satz"];
val it = [4, 3, 3, 4] : int list
```

- Berechne die Summe über eine Liste ganzer Zahlen:

```
- val sum = foldr op+ 0; ←rechte Seite ist Funktion!
val sum = fn : int list -> int
- sum [1, 2, 3, 4];
val it = 10 : int
```

- Gesamtlänge aller Strings in einer Liste:

```
- val gsum = sum o (map size);
val gsum = fn : string list -> int
- gsum ["Dies", "ist", "ein", "Satz"];
val it = 14 : int
```



## 7.7 Funktionen höherer Ordnung ...

---

### Beispiele zur Anwendung der Funktionen ...

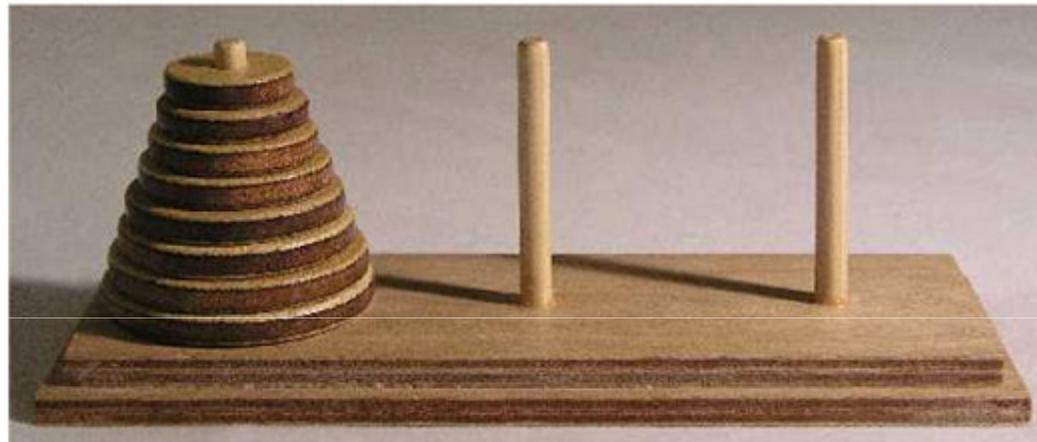
- **Listenkonkatenation:** `fun l1 @ l2 = foldr op:: l2 l1`
  - z.B.: `[1,2]@[3] = foldr op:: [3] [1,2] = 1::(2::[3])`
- **Listenlänge:** `fun length l = foldr (fn (x,y) => y+1) 0 l`
  - wenn wir `fn (x,y) => y+1` einmal abkürzend mit `f` bezeichnen, wird z.B. `length [6,7,8]` wie folgt berechnet:  
`f(6, f(7, f(8, 0))) = f(6, f(7, 1)) = f(6, 2) = 3`
- **Letztes Listenelement:** `val last = hd o rev`
  - d.h. `last l = hd (rev l)`
- **String-Umkehr:** `val revert = implode o rev o explode`
  - d.h. `revert s = implode(rev(explode s))`

## 7.8 Problemlösung mit Rekursion

---

### 7.8.1 Die Türme von Hanoi

- Gegeben ist ein Turm von Holzscheiben:



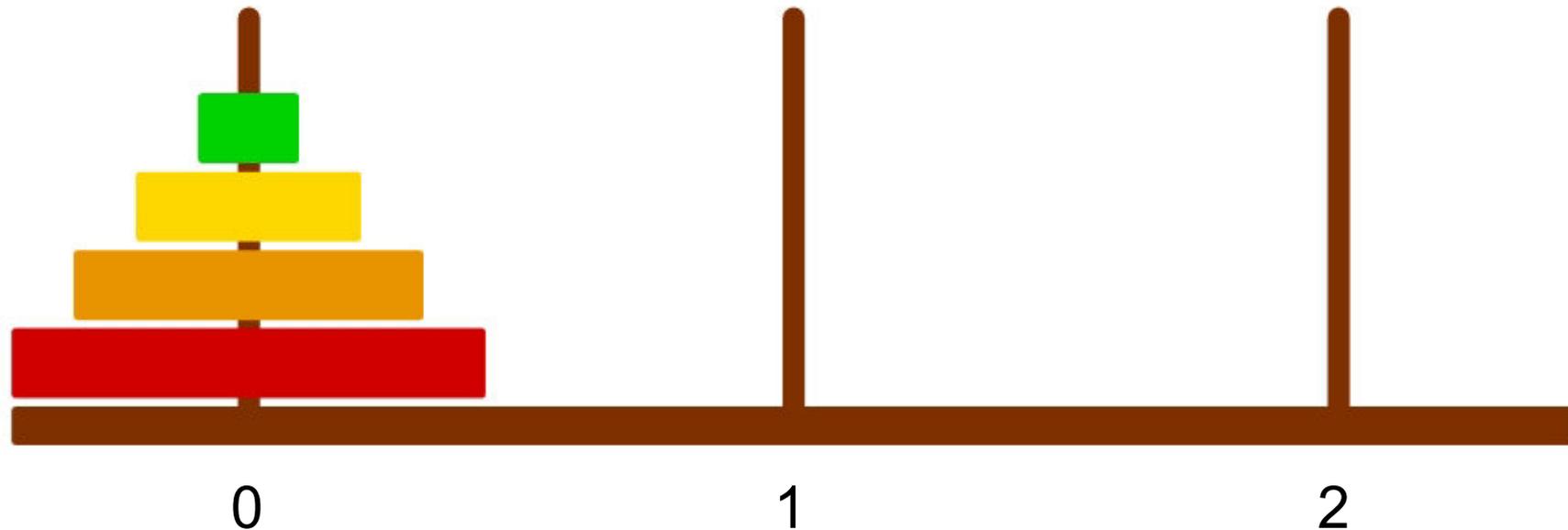
- Der Turm ist vom linken auf den rechten Stab zu verschieben, wobei
  - immer nur jeweils eine Scheibe bewegt werden und
  - nie eine größere auf einer kleineren Scheibe liegen darf

## 7.8.1 Die Türme von Hanoi ...

---

### Lösung des Problems:

Ausgangssituation:



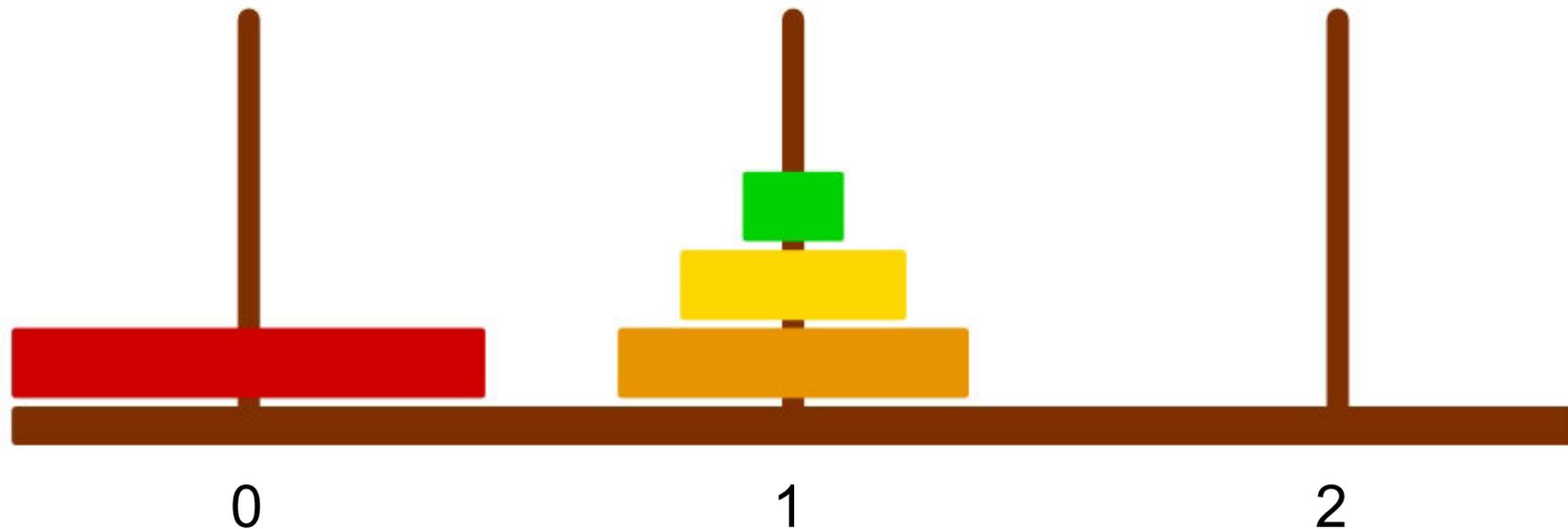


## 7.8.1 Die Türme von Hanoi ...

---

### Lösung des Problems: ...

Schiebe die oberen  $n-1$  Scheiben von Pos. 0 auf Pos. 1:



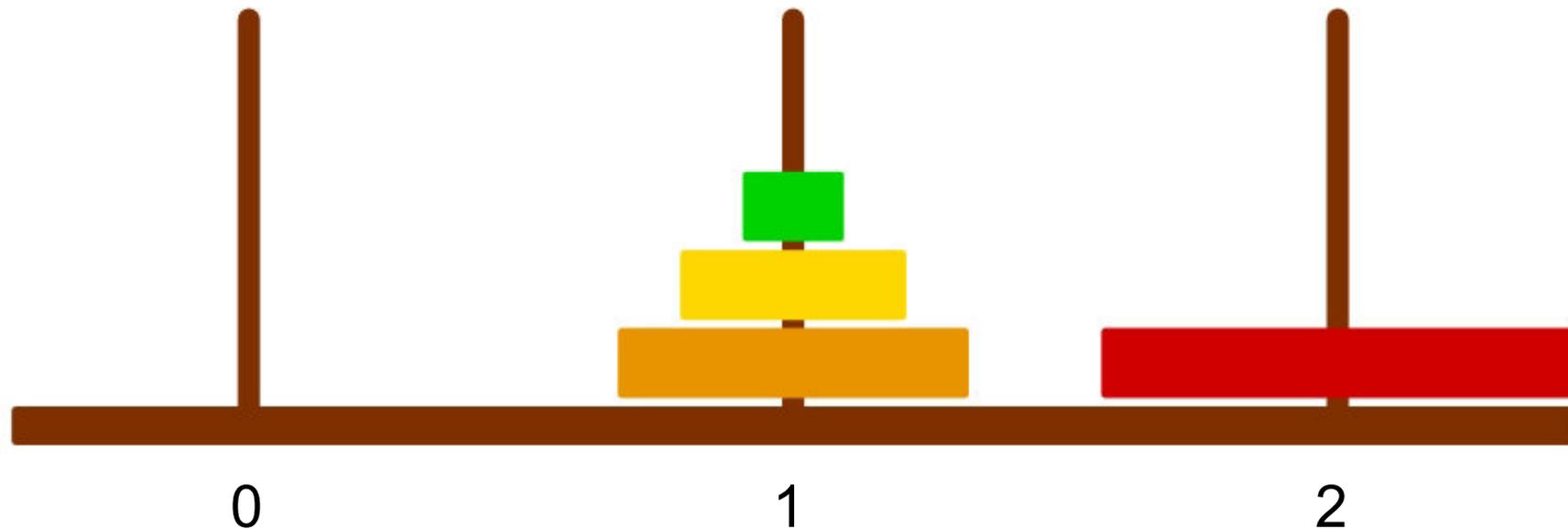


## 7.8.1 Die Türme von Hanoi ...

---

### Lösung des Problems: ...

Schiebe die unterste Scheibe von Pos. 0 auf Pos. 2:

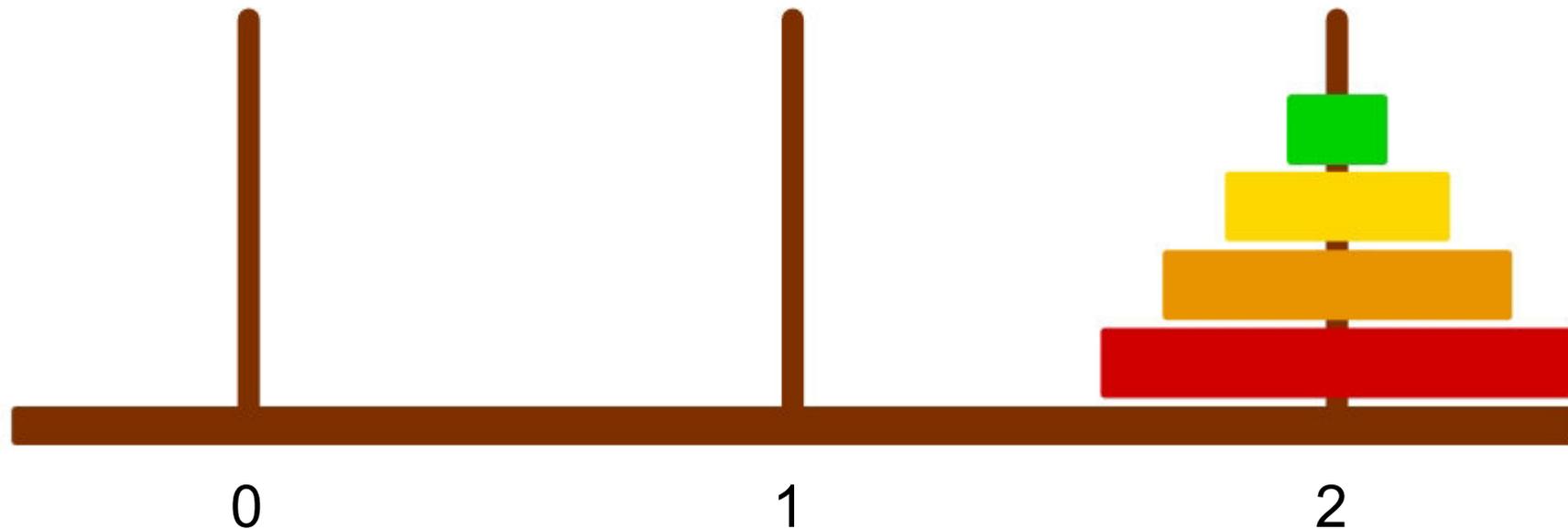


## 7.8.1 Die Türme von Hanoi ...



### Lösung des Problems: ...

Schiebe die oberen  $n-1$  Scheiben von Pos. 1 auf Pos. 2:



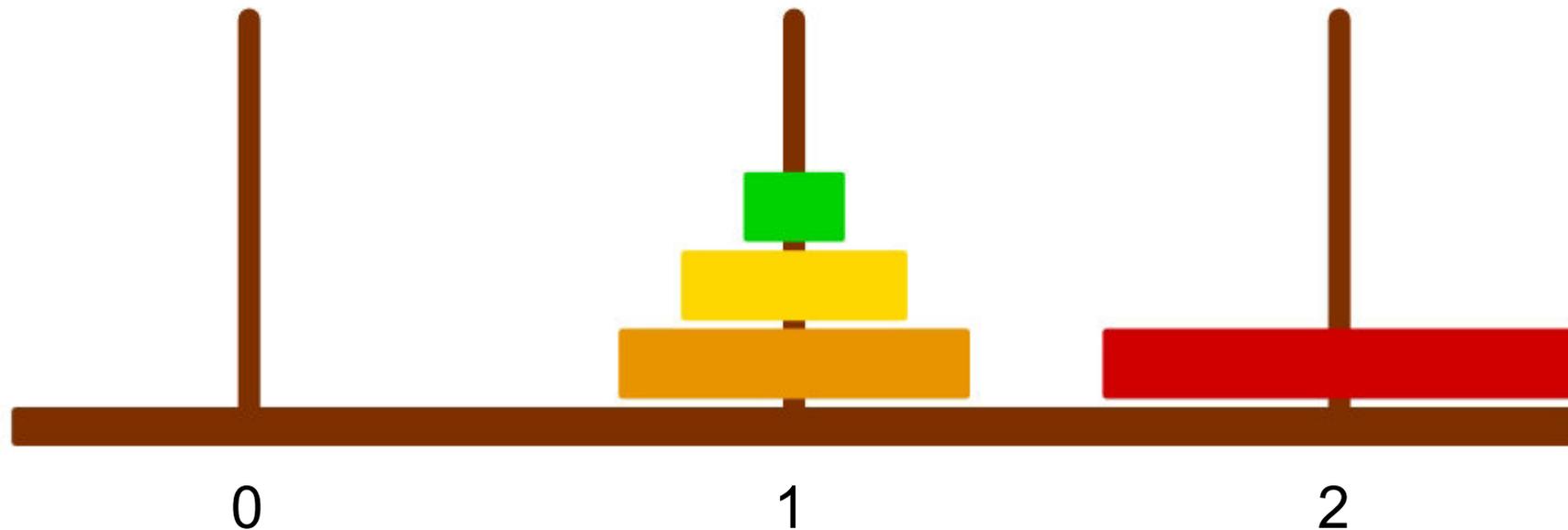


## 7.8.1 Die Türme von Hanoi ...

---

Und wie verschieben wir einen Turm der Höhe  $n-1$ ?

Antwort: Genauso! (Prinzip der Rekursion!)



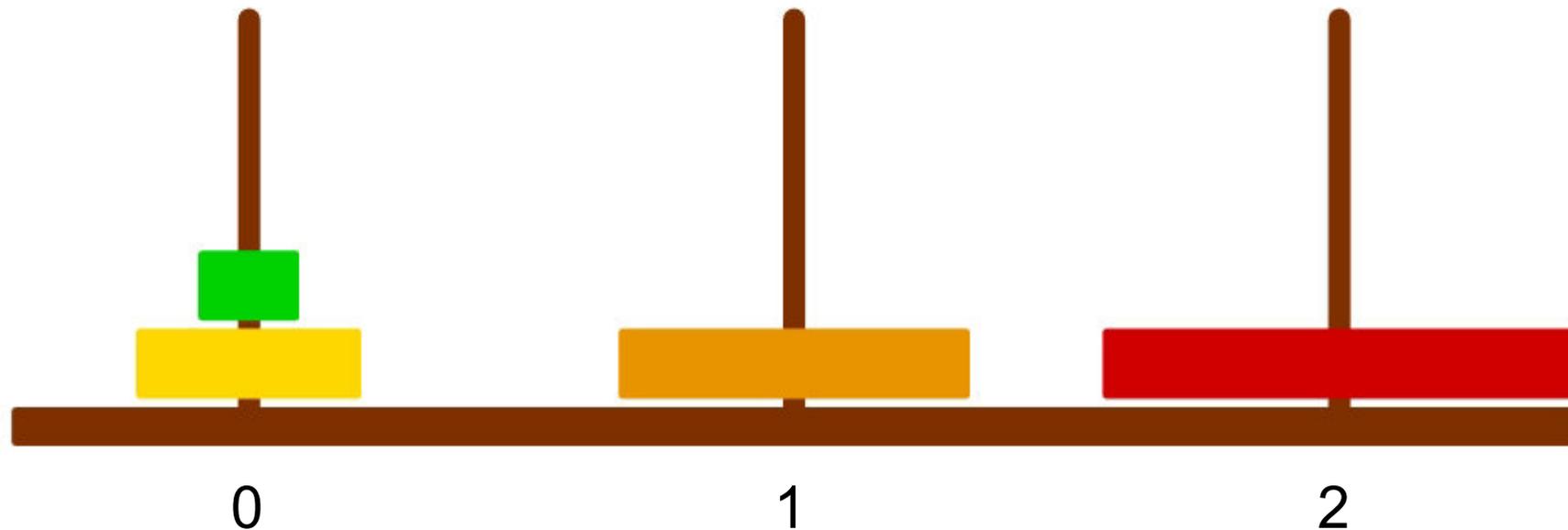


## 7.8.1 Die Türme von Hanoi ...

---

Und wie verschieben wir einen Turm der Höhe  $n-1$ ? ...

Schiebe die obersten  $n-2$  Scheiben von Pos. 1 auf Pos. 0



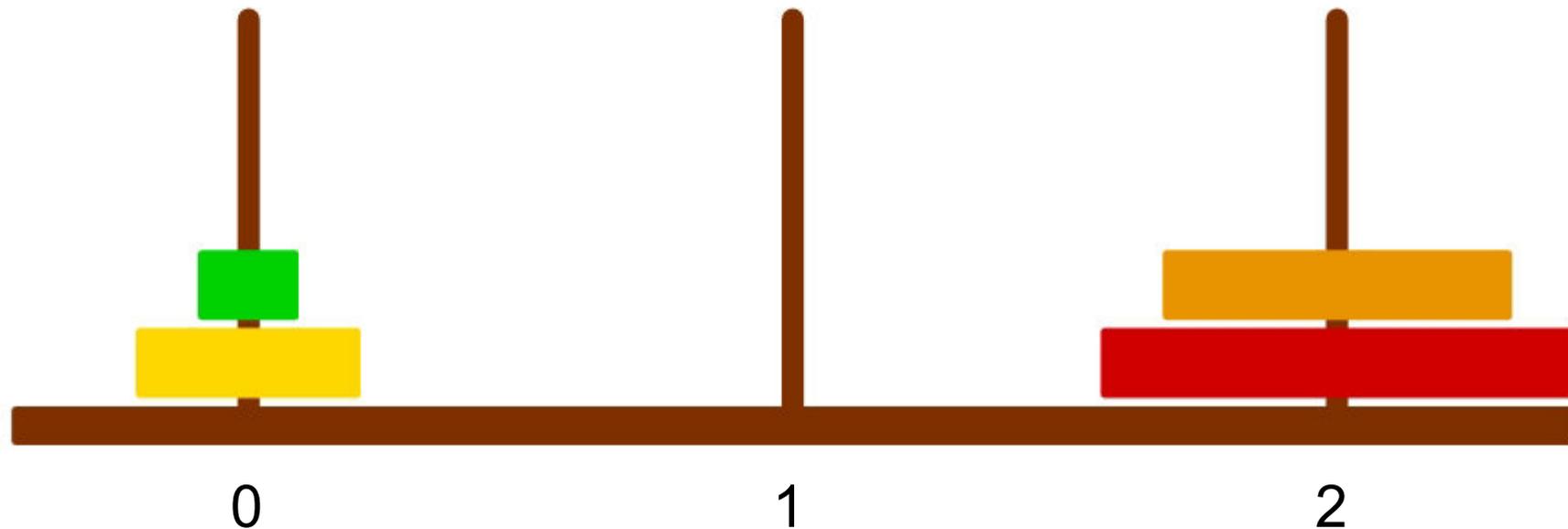


## 7.8.1 Die Türme von Hanoi ...

---

Und wie verschieben wir einen Turm der Höhe  $n-1$ ? ...

Schiebe die unterste Scheibe von Pos. 1 auf Pos. 2



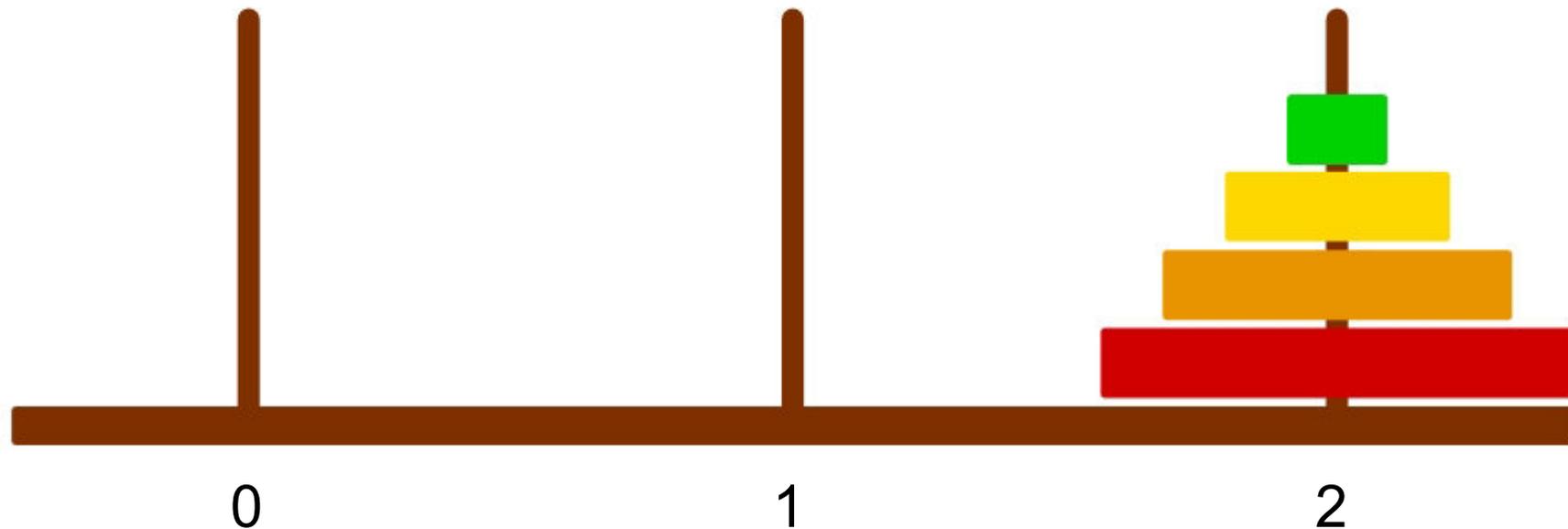


## 7.8.1 Die Türme von Hanoi ...

---

Und wie verschieben wir einen Turm der Höhe  $n-1$ ? ...

Schiebe die obersten  $n-2$  Scheiben von Pos. 0 auf Pos. 2





## 7.8.1 Die Türme von Hanoi ...

---

### SML-Programm für die allgemeine Lösung

```
- (* Die folgende Spezialanweisung sorgt dafür, daß der *)
- (* Compiler auch längere Listen vollständig ausgibt. *)
- Control.Print.printLength := 100;
val it = () : unit

- fun hanoi (1,A,B,C) = [(A,C)]
= | hanoi (n,A,B,C) = hanoi (n-1,A,C,B)
= @ [(A,C)]
= @ hanoi (n-1,B,A,C);
val hanoi = fn : int * 'a * 'a * 'a -> ('a * 'a) list

- hanoi (4, 0, 1, 2);
val it =
 [(0,1), (0,2), (1,2), (0,1), (2,0), (2,1), (0,1), (0,2), (1,2),
 (1,0), (2,0), (1,2), (0,1), (0,2), (1,2)] : (int * int) list
```

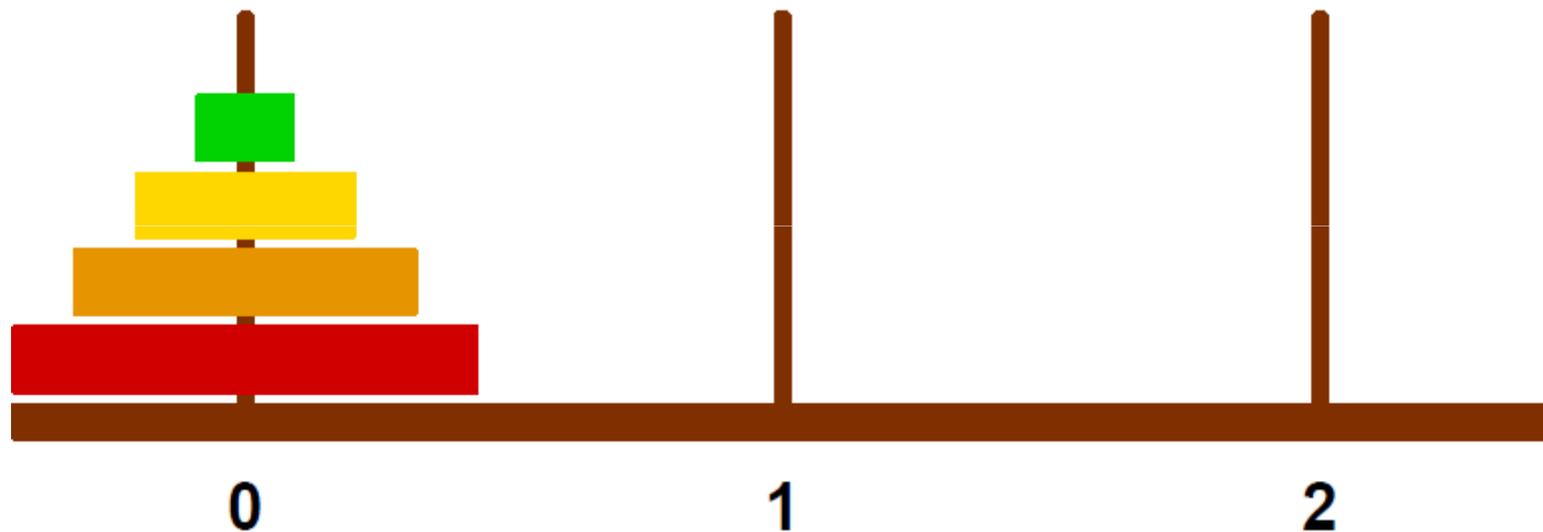


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 1:



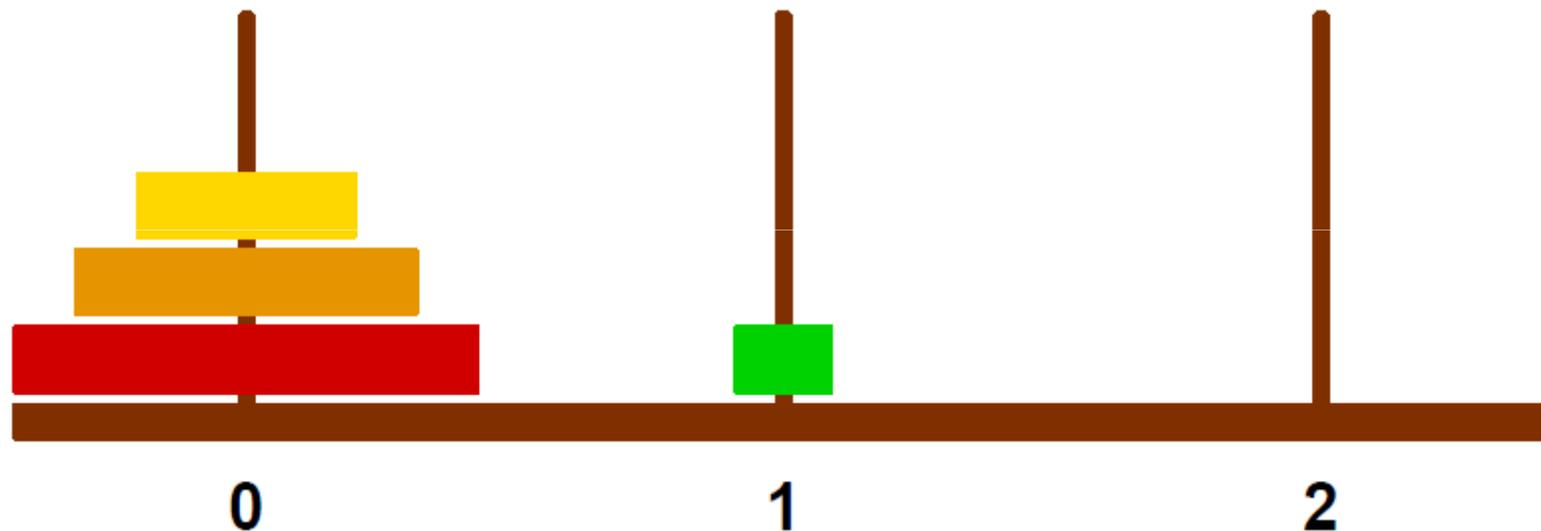


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 2:



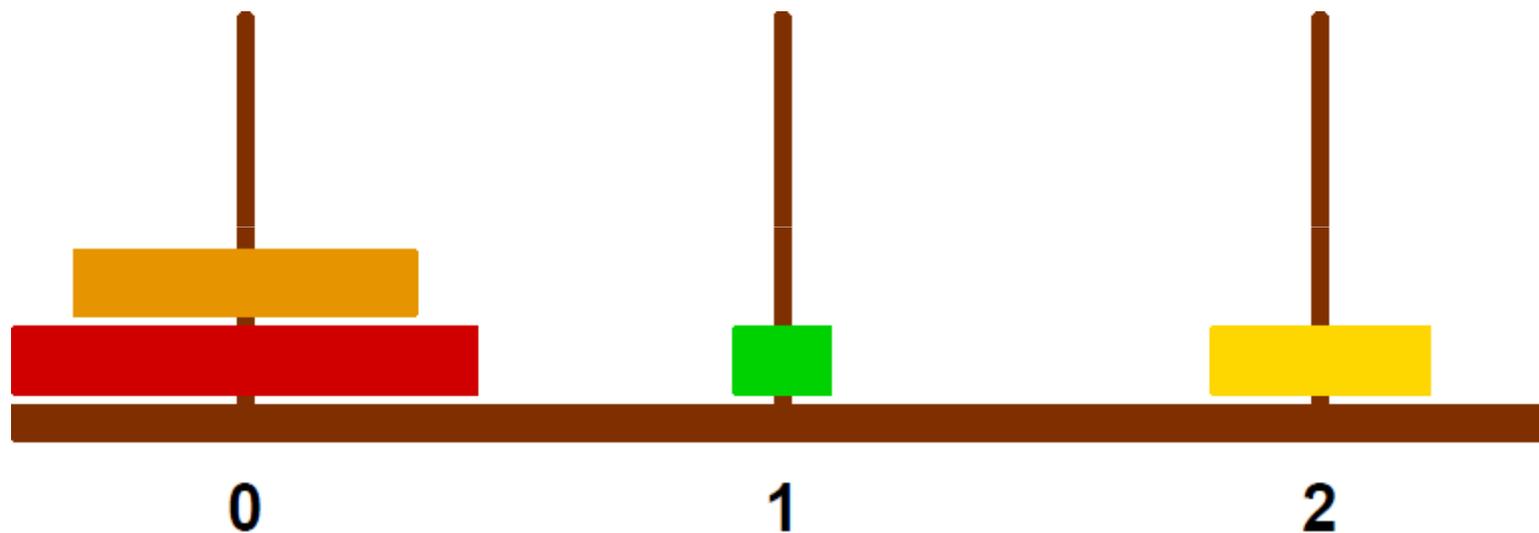


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 3:



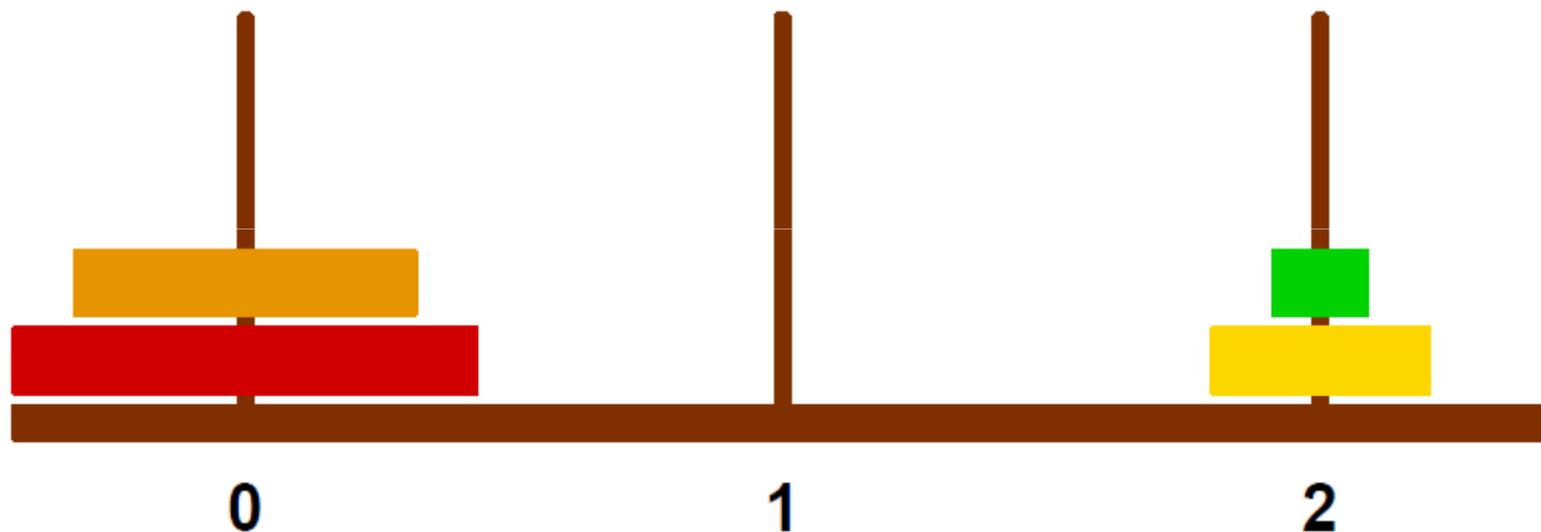


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 4:



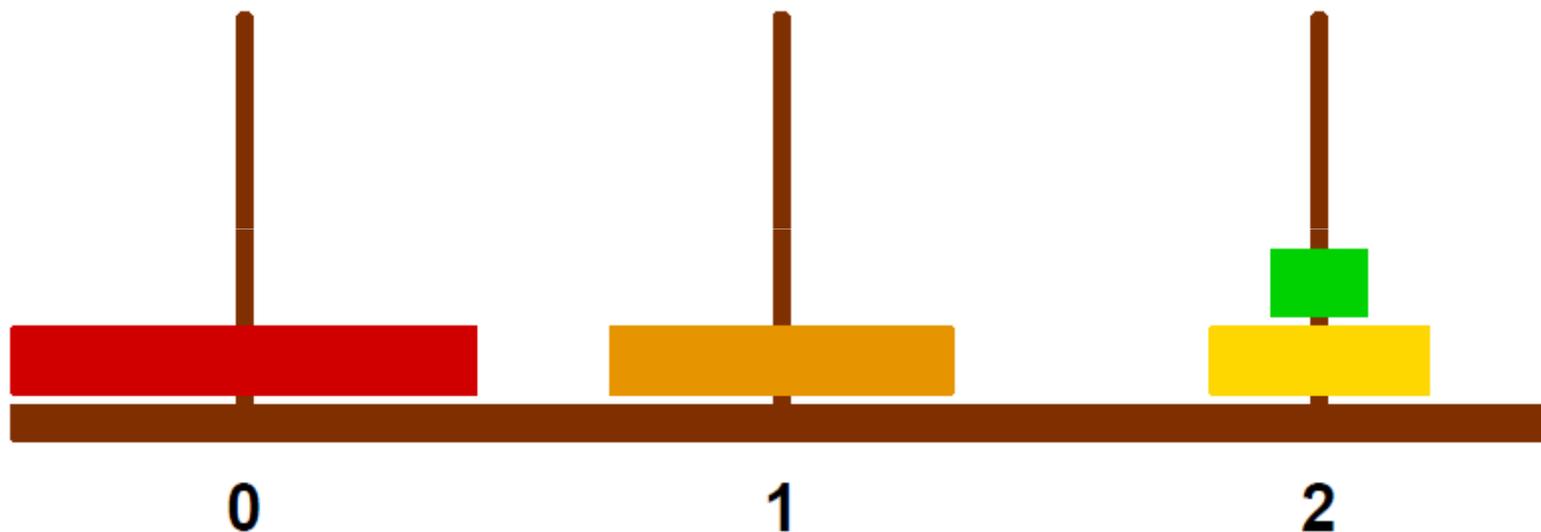


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 5:



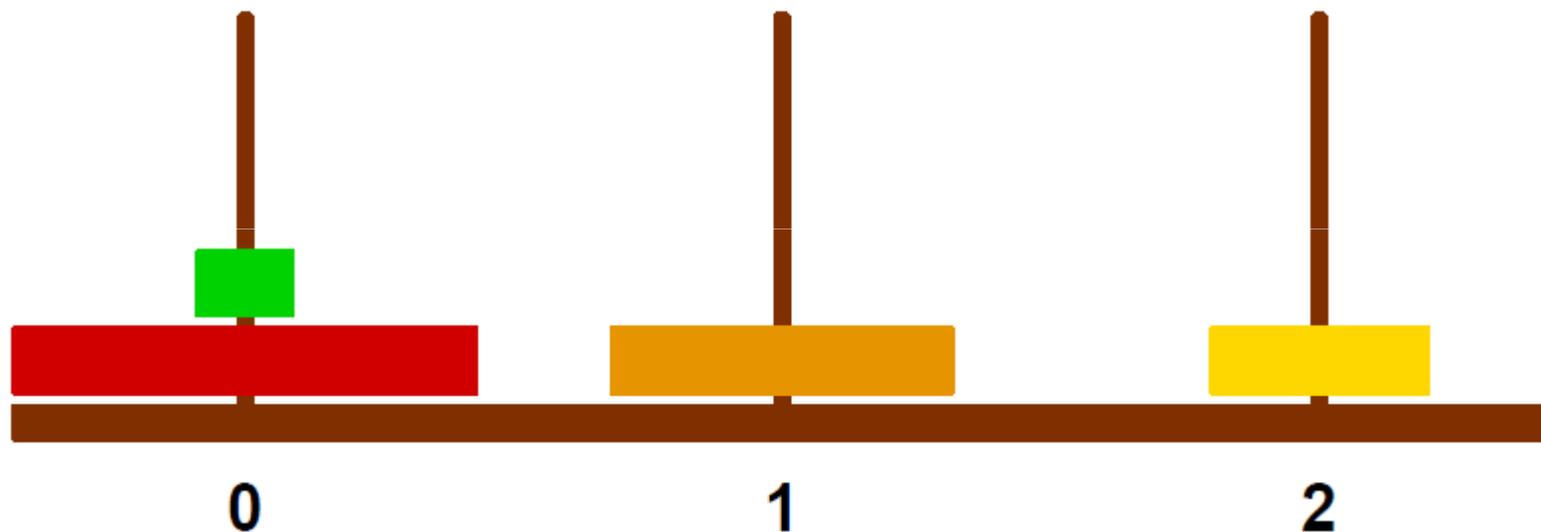


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 6:



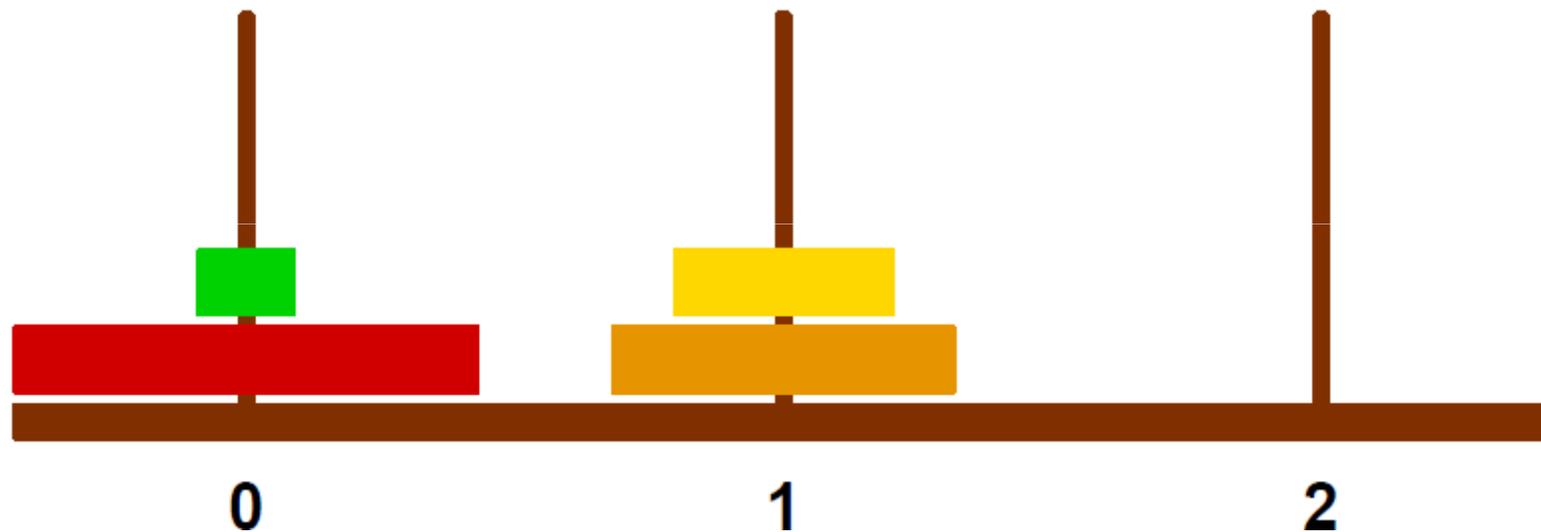


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 7:



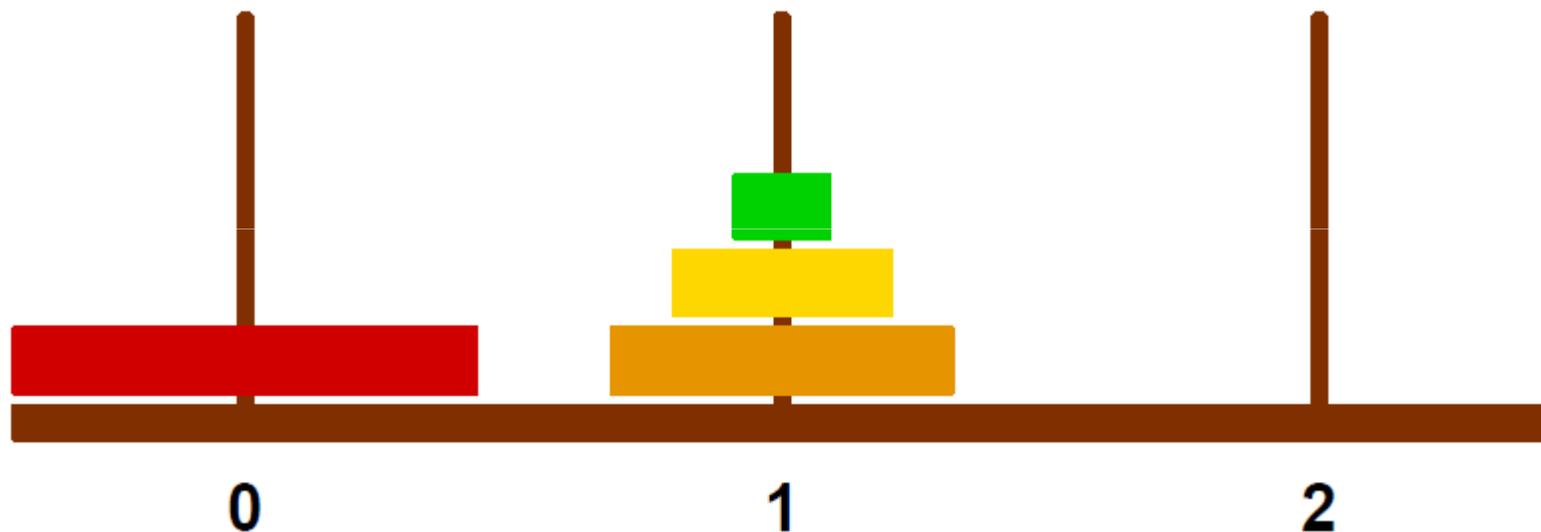


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 8:



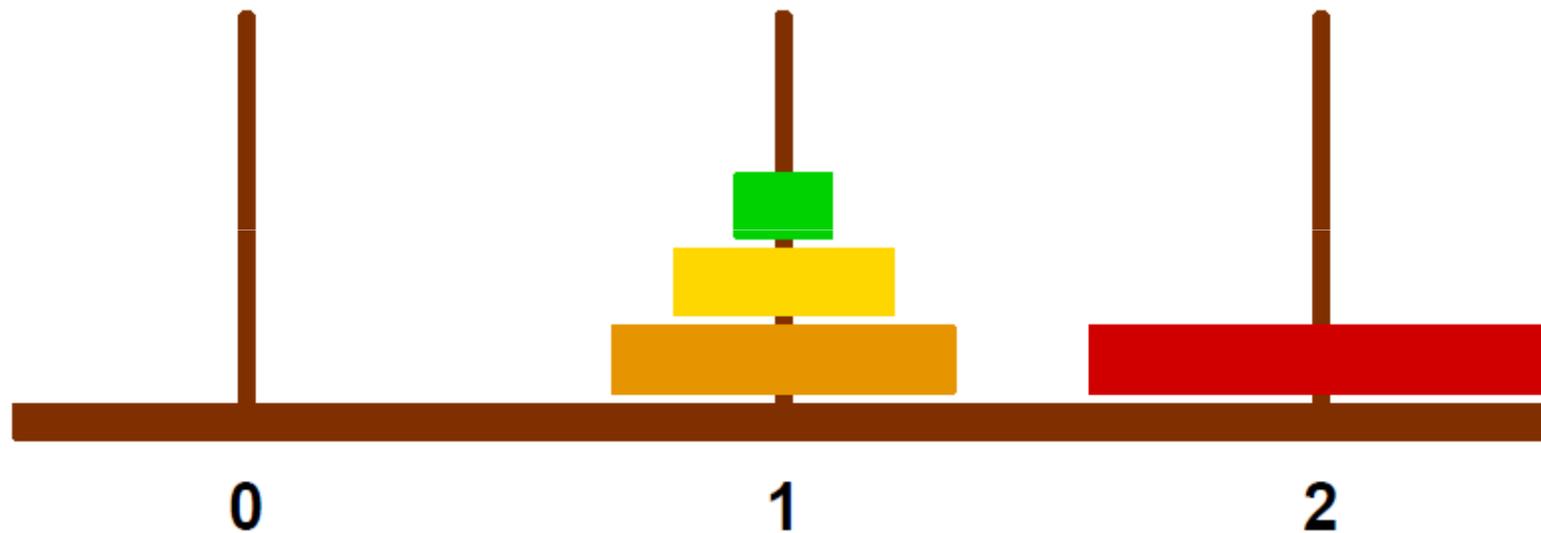


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 9:



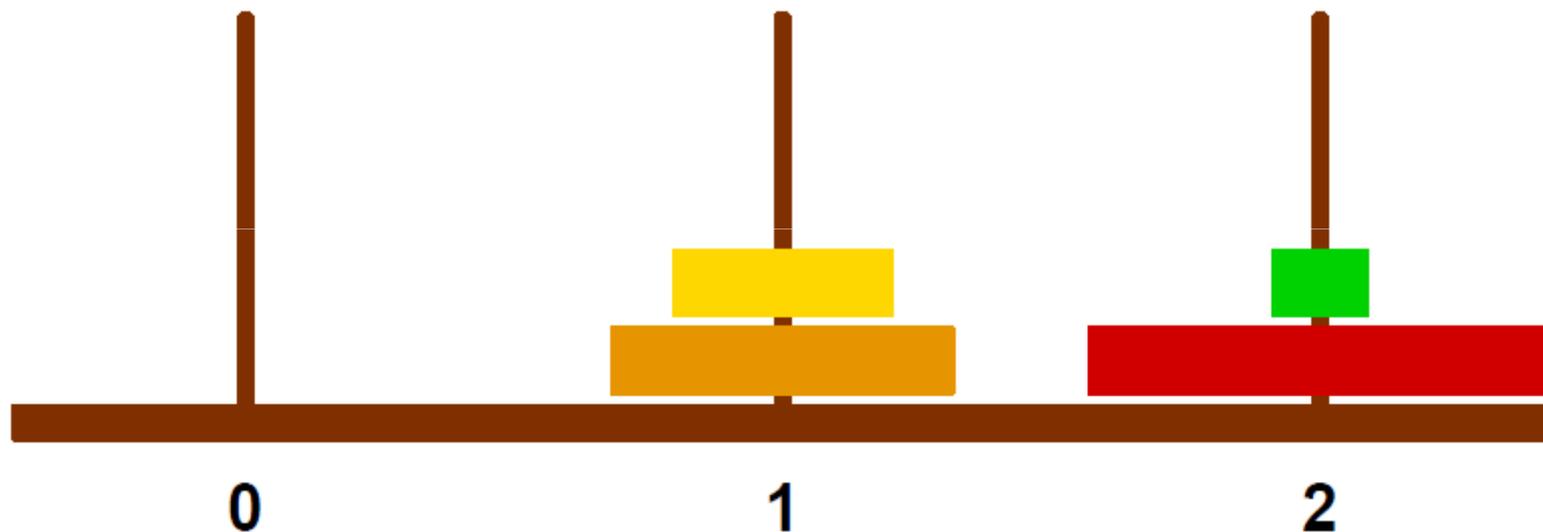


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 10:



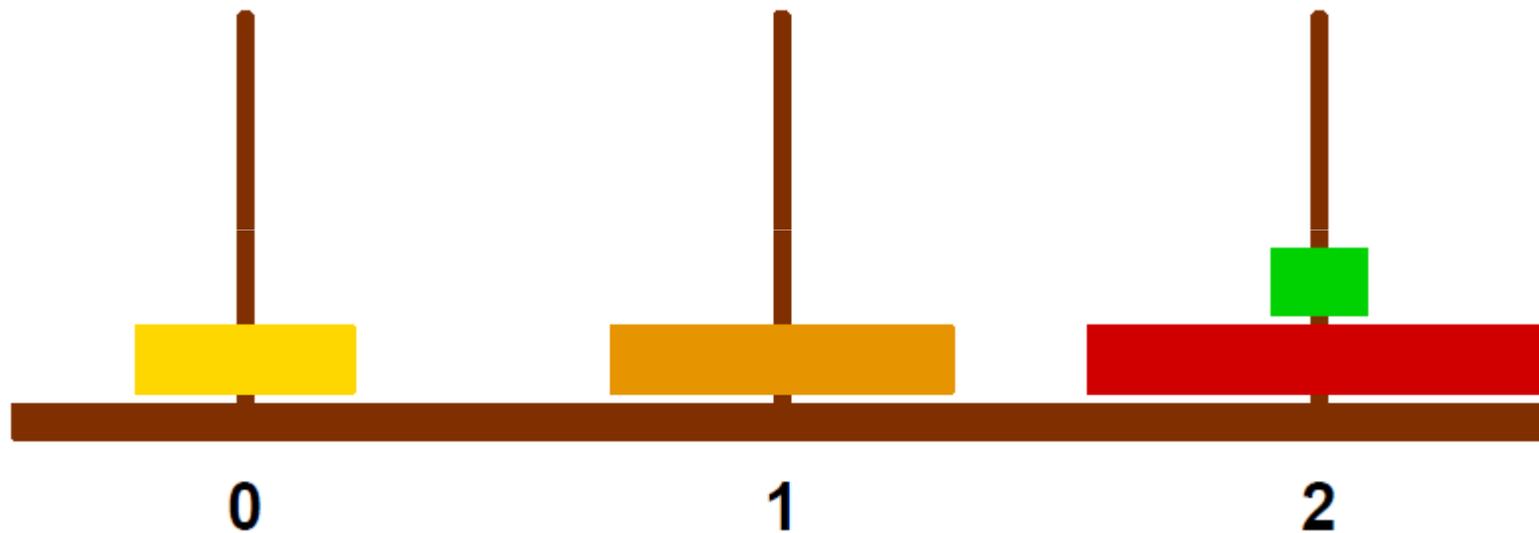


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 11:



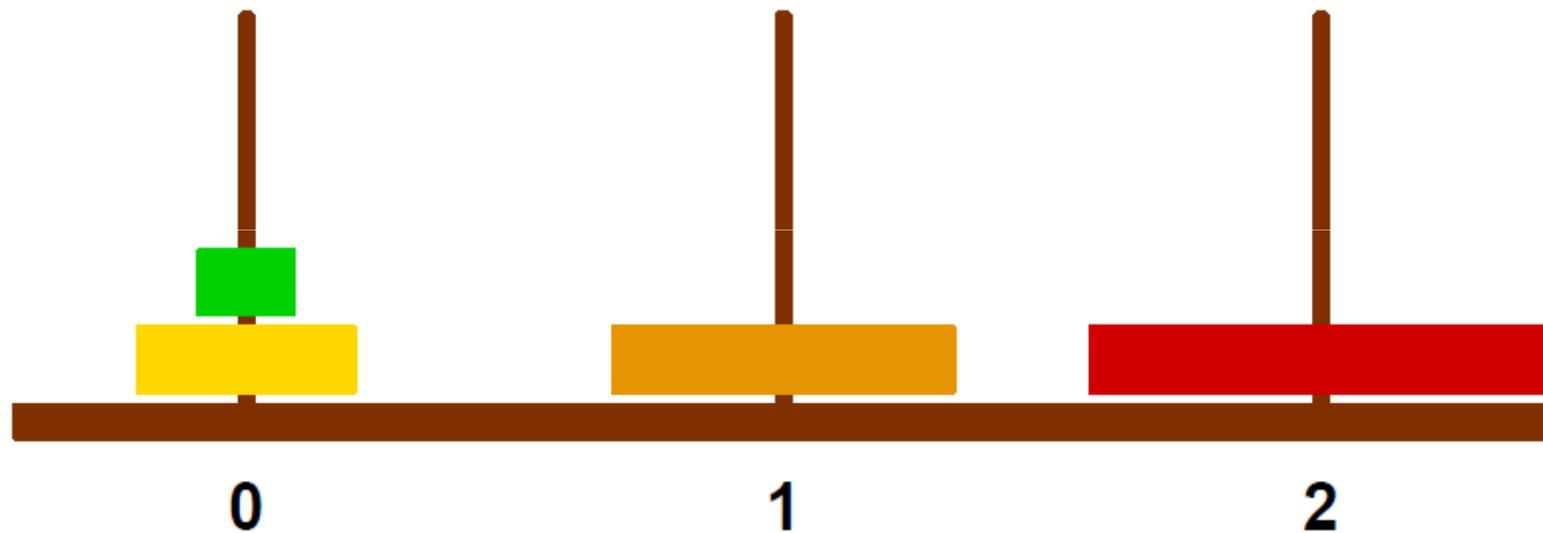


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 12:



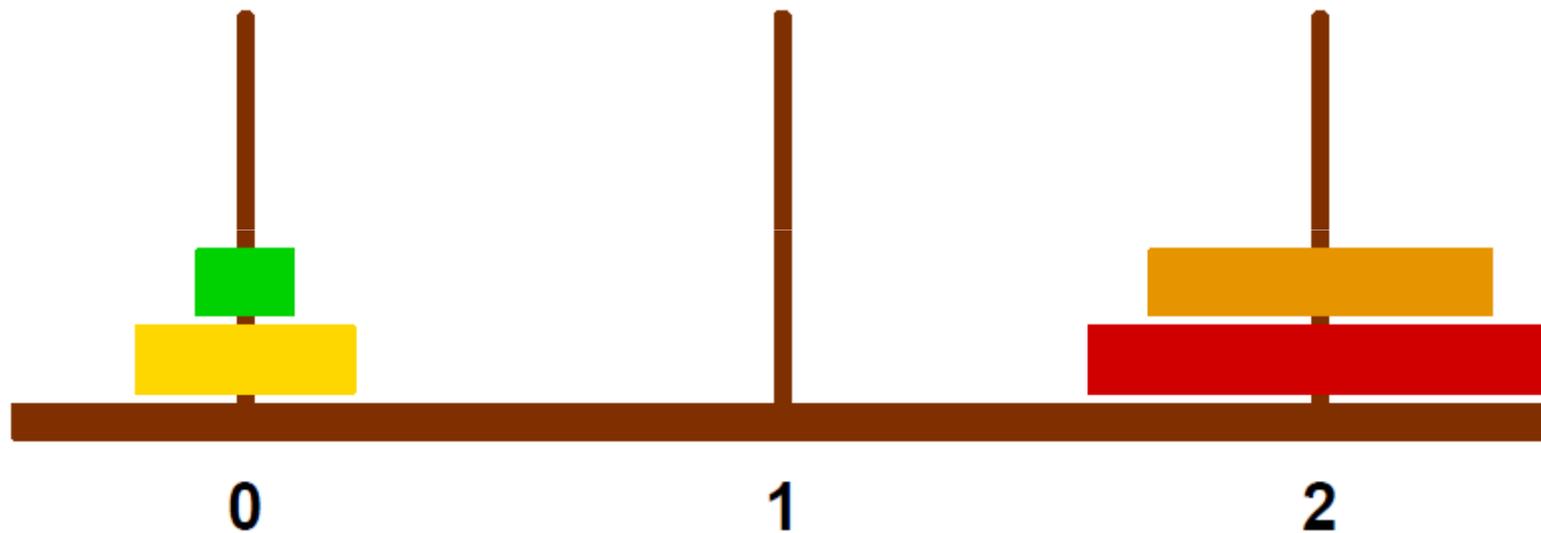


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 13:



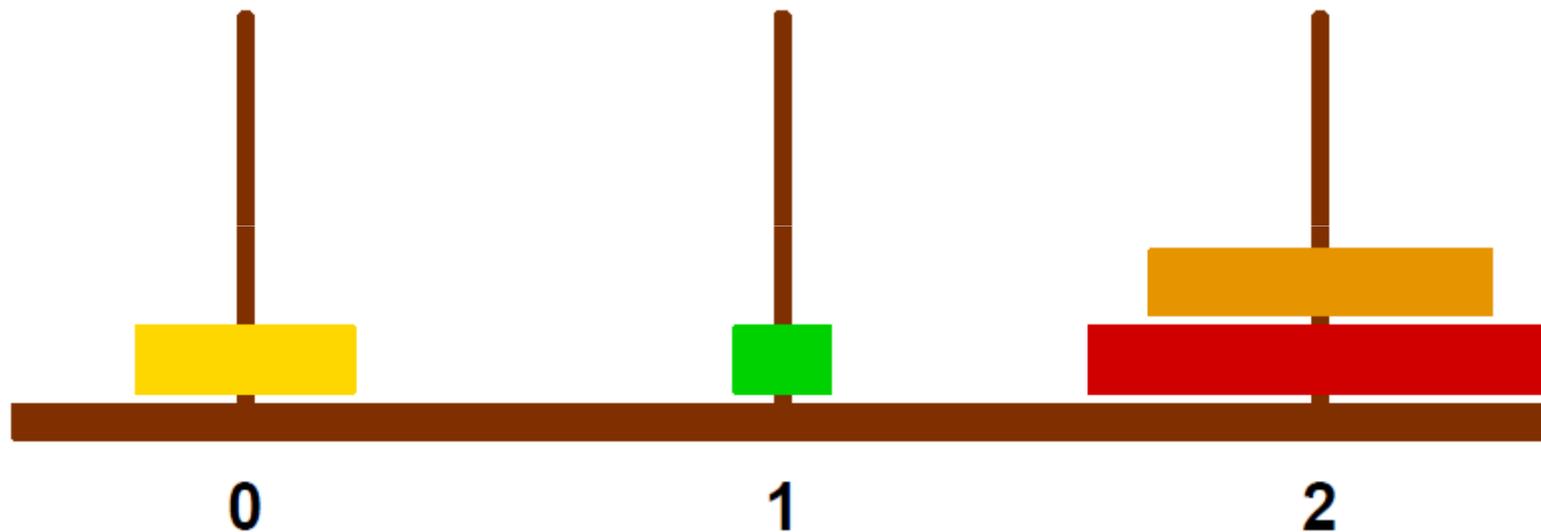


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 14:



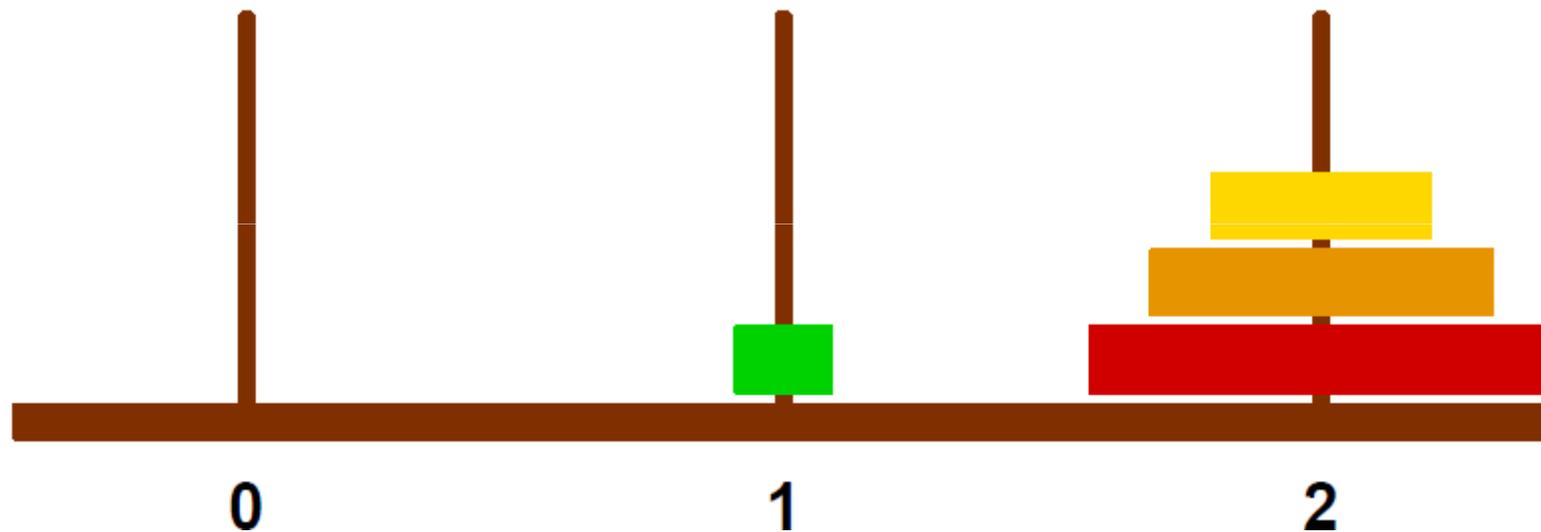


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 15:



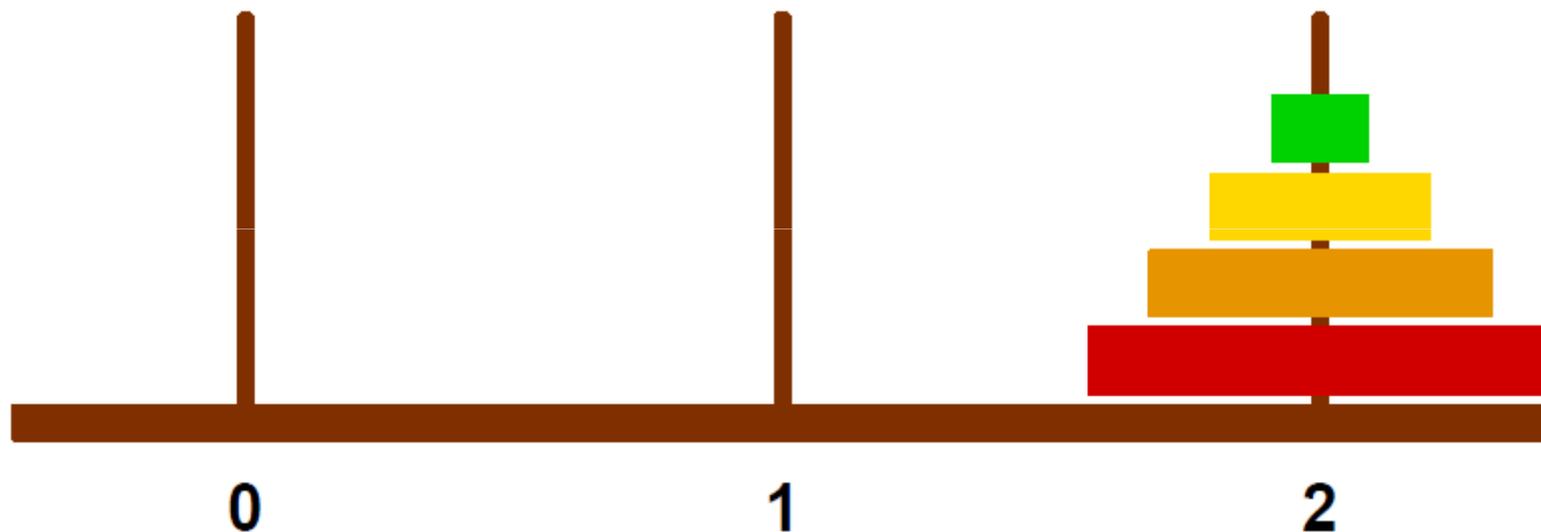


## 7.8.1 Die Türme von Hanoi ...

---

### Visualisierung der Lösung für Turmhöhe 4

Schritt 16:





## 7.8.1 Die Türme von Hanoi ...

---

### Wie lösen wir ein Problem durch Rekursion?

- Wir gehen davon aus, daß alle kleineren Probleme bereits gelöst sind
- Dann konstruieren wir aus der Lösung der kleineren Probleme eine Lösung des größeren Problems
  - oft: konstruiere aus der Lösung für die Problemgröße  $n$  eine Lösung für die Problemgröße  $n + 1$
- Zu beachten ist jetzt noch der Terminierungsfall, d.h. wir brauchen noch eine Lösung für die kleinsten Probleme
  - oft: Problemgröße 0 oder 1
- Vgl. das Beweisprinzip der vollständigen Induktion!



## 7.9 SML: Zusammenfassung

---

### Die Ausdrucksfähigkeit von SML wird bewirkt durch:

- Integrierte "Collections": Tupel, Records, Listen
- Betrachtung von Funktionen als normale Werte
  - damit: Funktionen höherer Ordnung, Currying
- Automatische Bestimmung der Typen (Typinferenz)
- Polymorphes Typsystem (parametrisierte Typen)
  - damit: typsichere generische Funktionen
- Konstruktorbasierte Datentypen
- *Pattern Matching* von Funktionsargumenten
- Rekursion (Funktionen und Datentypen)