Kapitel 2

Algorithmische Abstraktion

2.1 Algorithmen in Gofer und Haskell

Programmiersprachen lassen sich grob einteilen in

deklarative : Das WAS wird formalisiert, d.h. dies sind Programmiersprachen, deren Programme eher Spezifikationen sind.

imperativ (prozedural) : Das WIE wird formalisiert. Die Programmiersprache formalisiert genau den sequentiellen Ablauf des Algorithmus, der sich danach problemlos auf eine von-Neumann-Architektur abbilden läßt.

Diese Einteilung nach diesen Gesichtspunkten ist nicht ganz scharf, da manche Programmiersprachen Aspekte beider Eigenschaften haben. Funktionale Programmiersprachen wie Haskell sind den deklarativen P.S. zuzuordnen, während Assembler oder C den imperativen zuzuordnen sind. Sogenannte logische Programmiersprachen sind deklarativ. Die objektorientierten P.S. sind eher imperativ, beinhalten verschiedene Prinzipien, die die Formalisierung erleichtern.

Wichtige Eigenschaften **funktionaler Programmiersprachen**

Keine Seiteneffekte , d.h. keine dauerhaften Speicheränderungen, die von anderen Funktionen aus sichtbar sind. Insbesondere: Der Wert eines Ausdrucks ist nur bestimmt durch die Argumente. Mehrfacher Aufruf einer Funktion mit den gleichen Argumenten ergibt immer den gleichen Wert.
Pascal-Gegenbeispiel:

```
var count: integer;

function f (x: integer) : integer
begin count := count +1
```

```
        f := count
end;
```

Der Test $f(0) = f(0)$ ergibt "falsch" bei diesem Beispiel, da Seiteneffekte bewirkt werden.

Verzögerte Auswertung Nur die für das Resultat notwendigen Unterausdrücke werden ausgewertet.

Polymorphes Typsystem Alle Ausdrücke haben einen Typ, Funktionen können im Typ eine Typvariable haben, d.h. z.B. einen Typ $\forall a.a \to a$ haben. Das Typsystem garantiert, daß niemals Daten vom Typ her falsch interpretiert werden.

automatische Speicherverwaltung Anforderung von Speicher und Freigabe von unerreichbaren Objekten erfolgt automatisch.

Funktionen sind Datenobjekte Funktionen können wie Zahlen behandelt werden, d.h. sie können in Datenobjekten als Teil auftauchen und sind als Argument und Resultat von Funktionen zugelassen

Wir wollen die funktionale (applikative) Programmiersprache Haskell zur Modellierung von Algorithmen und Datenstrukturen und deren Analyse verwenden. Dazu werden wir den Interpreter Gofer oder den Hugs (Haskells User Gofer System) verwenden, die es erlauben, sehr schnell komplexere Algorithmen zu entwerfen, zu testen und zu analysieren. Dazu müssen wir auch etwas über diese Programmiersprache(n) wissen.

Haskell ist reich an sauberen Möglichkeiten Algorithmen und zugehörige Daten in modularer Weise zu strukturieren. Diese Konzepte kann man nach und nach erlernen, verstehen und benutzen.

Grundprinzipien sind:

man definiert Funktionen und
läßt Ausdrücke auswerten.

Vordefiniert sind (je nach Interpreter): ganze Zahlen mit $|n| < 2^{31} - 1 = 2147483647$, beliebig lange ganze Zahlen, rationale Zahlen, Fließkommazahlen und entsprechende arithmetische Operatoren. Man beachte, daß die Sprache ein komplexes Typsystem besitzt, das nur typbare Ausdrücke zuläßt. Im Moment können wir dies ignorieren, solange keine Fehler auftreten.

Beispiel 1. Definition eines Polynoms $x^2 + y^2$:

```
quadratsumme x y = x*x + y*y
```

Eingabe in den Interpreter (am besten einen File laden, der diese Definition textuell enthält)

```
Gofer?
quadratsumme 3 4              <CR>
25
```

Definition 1. Die (vereinfachte) Syntax einer Funktionsdefinition ist:

$$
\begin{array}{lcll}
\langle \text{Def} \rangle & ::= & \langle \text{Funktionsname} \rangle \langle \text{Parameter} \rangle^* = \langle \text{Ausdruck} \rangle & \\
\langle \text{Ausdruck} \rangle & ::= & \langle \text{Bezeichner} \rangle \mid \langle \text{Zahl} \rangle & \\
 & & \mid (\langle \text{Ausdruck} \rangle\ \langle \text{Ausdruck} \rangle) & \text{(Anwendung)} \\
 & & \mid (\langle \text{Ausdruck} \rangle) & \\
\langle \text{Bezeichner} \rangle & ::= & \langle \text{Funktionsname} \rangle \mid \langle \text{Datenkonstruktorname} \rangle &
\end{array}
$$

Funktionsnamen und Datenkonstruktornamen sind teilweise schon im "Prelude" vorhanden, weitere sind benutzerdefinierbar. Datenkonstrukoren werden syntaktisch von Funktionsnamen dadurch unterschieden, daß Datenkonstruktoren mit großen Buchstaben beginnen. Beispiele sind: `True, False`. Es sind verschiedene Arten von Zahlen verfügbar: kurze ganze Zahlen (*Int*), beliebig lange ganze Zahlen (Integer), Brüche (Rational), Fließkommazahlen (Float).

Da Haskell sowohl Präfix als auch Infix-Operatoren und auch Prioritäten für Operatoren hat, werden oft Klammern weggelassen, wenn das Programm eindeutig lesbar bleibt. Eine Spezialität ist das Weglassen der Klammern bei (links-)geschachtelten Anwendungen: $s_1\ s_2 \ldots s_n$ ist dasselbe wie $((\ldots(s_1\ s_2)\ s_3 \ldots)\ s_n)$.

Es ist zu beachten, daß es Kontextbedingungen gibt: die formalen Parameter müssen verschiedene Namen haben. Außerdem müssen alle undefinierte Namen im rechten Ausdruck formale Parameter sein.

Zur Trennung können außerdem noch die Klammern "{","}" und das Semikolon "; " verwendet werden. Diese Syntax werden wir schrittweise erweitern und mit Beispielen erläutern.

Beispiel 2.

```
fakultaet x  = if x == 0 then 1
               else x*(fakultaet (x-1))
```

```
> fakultaet 1              <CR>
1
> fakultaet 6              <CR>
720
> fakultaet (20::Integer)
2432902008176640000
```

2.1.1 Fallunterscheidung

Die Fallunterscheidung ist ein Ausdruck mit der Syntax:

`if` ⟨Ausdruck⟩ `then` ⟨Ausdruck⟩ `else` ⟨Ausdruck⟩

Der Ausdruck nach dem `if` muß einen Booleschen Wert ergeben. Der Versuch einer falschen Eingabe ergibt einen Typfehler (je nach Interpreter):

Beispiel 3.

```
> (if 1 then 1 else 2)    <CR>
ERROR: Type error in conditional
*** expression     : if 1 then 2 else 3
*** term           : 1
*** type           : Int
*** does not match : Bool

> (if 1 then 1 else 2)    <CR>
ERROR: Bool is not an instance of class "Num"
```

Vergleichsausdrücke können gebildet werden mit den Infixoperatoren $==, <, >, <=, >=, /=$ Achtung: (= alleine ist reserviert für Funktionsdefinitionen, == ist der Vergleichsoperator). Weitere Boolesche Ausdrücke sind Anwendungen von wahrheitswertigen Funktionen (Prädikate) auf Argumente. Diese können auch benutzerdefiniert sein.

Boolesche Ausdrücke können kombiniert werden mit not, ||, && für die logischen Funktionen nicht, oder, und. Die Konstanten für wahr und falsch sind verfügbar als `True, False`. (Achtung: erster Buchstabe groß geschrieben).

2.1.2 Problemanalyse

Wir betrachten zunächst Algorithmen, die nur einfach strukturierte Daten (Zahlen) verwenden, so daß wir uns zunächst auf den reinen Algorithmenentwurf konzentrieren.

Die Mittel zur Problemanalyse und zur Erstellung eines Algorithmus sind dann:

- Zerlegung in (einfachere) Teilprobleme
- Lösen der Teilprobleme mittels Unterfunktionen
- Zusammensetzen des Algorithmus.

Wichtig bei diesem Verfahren der *funktionalen Abstraktion* ist

was die Funktion leistet,

nicht: wie die Funktion realisiert wird.

D.h. zentrale Leitideen beim Entwurf sind die *Korrektheit*, *Modularität* neben der *Effizienz*.

2.1.3 Berechnung der Wurzel aus x mit dem Newtonschen Iterationsverfahren

Dieses Beispiel demonstriert die Zerlegung der Aufgabe, eine iterative Funktion zur Berechnung der Quadratwurzel zu definieren. Eine nichtprozedurale Beschreibung ist:

$\sqrt{x} := y$ wobei $y^2 = x$ und $y \geq 0$.

Die Berechnung von $\sqrt{x}$ mit dem Newton-Verfahren geschieht wie folgt: Ein Schätzwert s für die Wurzel $\sqrt{x}$ wird wiederholt durch $0.5(s + \frac{x}{s})$ ersetzt.

```
quadrat x = x*x
gutgenug  schaetzwert x =
           abs ((quadrat schaetzwert) - x) < 0.001


wurzel x = wurzeliter 1.0 x

wurzeliter schaetzwert x =
          if  gutgenug schaetzwert x  then schaetzwert
else wurzeliter (verbessern schaetzwert x)  x

verbessern schaetzwert x =
             mittelwert  schaetzwert   (x / schaetzwert)

mittelwert x y =  (x + y) / 2.0

> wurzel 2.0      <CR>
1.41422
```

2.1.4 Auswertung, operationale Semantik: Transformationen

Jetzt haben wir genug Beispiele, um die uns interessierenden Teile der *operationalen Semantik* von Haskell zu verstehen. Um die Betrachtung zu vereinfachen, gehen wir davon aus, daß die Infix-, Postfixnotation von Anwendungen (intern) bereits in Präfixnotation umgewandelt wurde, und daß eine Fallunterscheidung (`if` b `then` s_1 `else` s_2) in eine Anwendung (`ifthenelse` b s_1 s_2) umgewandelt wurde. Z.B. (`mittelwert` x y) entspricht den Anwendungen ((`mittelwert` x)y), und $x * y$ dem (internen) Ausdruck $((* \ x) \ y)$. [1]

Ein *Programm* ist eine Menge von Funktionsdefinitionen. Weiterhin hat man einen Ausdruck, der auszuwerten ist.

Das Prinzip der Berechnung ist:

$$\boxed{\text{Auswertung} = \text{Folge von Transformationen}}$$

D.h. Gegeben eine Menge von Funktionsdefinitionen, wird ein Ausdruck solange transformiert, bis ein Resultat erzielt ist, d.h. bis man nicht weiter transformieren kann. Als *Resultate* bezeichnen wir (vorläufig) nur Zahlen und die Booleschen Datenkonstruktoren `True` und `False`.

Die wichtigste Transformation ist die *Definitionseinsetzung (β-Reduktion)*:

> $(f \ t_1 \ldots t_n) \to (\texttt{Rumpf}_f[t_1/x_1, \ldots t_n/x_n])$
> Wenn f n formale Parameter hat und die formalen Parameter im Rumpf von f die Namen $x_1, \ldots x_n$ haben und $\texttt{Rumpf}_f$ der Rumpf der Funktionsdefinition von f ist.

Der Ausdruck $(\texttt{Rumpf}_f[t_1/x_1, \ldots t_n/x_n])$ entsteht durch (paralleles bzw. unabhängiges) Ersetzen der formalen Parameter x_i durch die jeweiligen Ausdrücke (die aktuellen Parameter) t_i. D.h. die Ausdrücke, die die aktuellen Parameter sind, werden textuell für die formalen Parameter eingesetzt. Damit dies syntaktisch akzeptabel bleibt, sollte man die aktuellen Parameter mit Klammern versehen.

Ein Spezialfall sind die Reduktionen der eingebauten arithmetischen Funktionen. Diese erwarten ihre Parameter bereits als Resultate und transformieren dann den Ausdruck wie gewohnt in eine Zahl.

Eine weitere Programmtransformation ist die *Auswertung einer Fallunterscheidung (case-Reduktion)*:

> $(\texttt{ifthenelse True } e_1 \ e_2) \to e_1$
> $(\texttt{ifthenelse False } e_1 \ e_2) \to e_2$

Wir nennen eine Transformation auch Reduktion und eine Folge von Programmtransformationen auch Reduktionsfolge (oder Auswertung).

[1] Will man diesen Ausdruck direkt syntaktisch eingeben, so ist zu beachten, daß $*$ syntaktisch in spezieller Weise behandelt wird: Klammerung verhindert das: $(((*) \ x) \ y)$. Der Ausdruck $((* \ x) \ y)$ ist auch erlaubt, bedeutet aber $y * x$.

Beispiel 4. Die Auswertung von `wurzel 2.0` ist eine Folge von Transformationen. Im folgenden sind die jeweiligen Ausdrücke angegeben.

```
wurzel 2.0

wurzeliter 1.0 2.0

if  gutgenug 1.0 2.0  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  abs ((quadrat 1.0) - 2.0) < 0.001  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  abs ((1.0 * 1.0) - 2.0) < 0.001  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  abs (1.0 - 2.0) < 0.001  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  abs (-1.0) < 0.001  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  1.0 < 0.001  then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

if  False then 1.0
else wurzeliter (verbessern 1.0 2.0)  2.0

wurzeliter (verbessern 1.0 2.0)  2.0

if  gutgenug (verbessern 1.0 2.0) 2.0  then (verbessern 1.0 2.0)
else wurzeliter (verbessern (verbessern 1.0 2.0) 2.0)  2.0

......
```

Offenbar gibt es Situationen, in denen man an verschiedenen Stellen eines Ausdrucks transformieren kann.

Beispiel 5. Für den Ausdruck $\texttt{quadrat}(4+5)$ gibt es 3 verschiedene Ersetzungsfolgen (Auswertungsreihenfolgen):

1. $\texttt{quadrat}(4+5) \to (4+5) * (4+5) \to 9 * (4+5) \to 9 * 9 \to 81$

2. `quadrat`$(4+5) \to (4+5)*(4+5) \to (4+5)*9 \to 9*9 \to 81$

3. `quadrat`$(4+5) \to$ (`quadrat` $9) \to 9*9 \to 81$

Man sieht, daß das **Ergebnis gleich ist**. Allerdings ist die ANzahl der Reduktionen verschieden.

Satz Church-Rosser I *Wenn für einen Ausdruck e zwei verschiedene Reduktionsfolgen mit den (ganzzahligen) Ergebnissen e_1 bzw. e_2 terminieren, dann sind die Ergebnisse gleich, d.h. $e_1 = e_2$.*

Damit wissen wir jetzt: Wenn man eine Menge von definierten Funktionen hat, dann spezifiziert ein Ausdruck ein eindeutiges Resultat. Möglicherweise ist das Resultat auch undefiniert, wenn man den Ausdruck nicht in ein Resultat überführen kann.

Die eingebauten Funktionen wie $*, +, -$ verlangen, daß ihre Argumente als Resultate vorliegen, deshalb ist es nicht ausreichend, wenn man nur Funktionsanwendungen als Ersetzungen durchführt. Man sagt auch, diese Funktionen sind *strikt* (implementiert).

Es gibt zwei ausgezeichnete *Reduktionsstrategien*:

Applikative Reihenfolge: Eine Funktionsanwendung darf erst dann transformiert werden, wenn die aktuellen Argumente bereits Resultate darstellen.

Normale Reihenfolge: Der am weitesten oben, und danach am weitesten links stehende transformierbare Unterausdruck wird transformiert.

Die erste Reduktionsfolge oben ist in normaler Reihenfolge, Die dritte Reduktionsfolge oben ist in applikativer Reihenfolge. Die applikative Reihenfolge erinnert an "call by value" während die normale Reihenfolge an "call by name" erinnert. Allerdings sieht man, daß es auch andere Reduktionsfolgen gibt.

Die normale Reihenfolge der Ersetzung hat zu beachten, daß es Funktionsanwendungen gibt, bei denen die Definitionsersetzung nicht durchführbar ist, da bestimmte (eingebaute) Funktionen ihre Argumente in einer ausgewerteten Form erwarten.

Die Auswertung eines Ausdrucks (per Transformation) in normaler Reihenfolge geschieht deshalb nach folgendem (rekursiven) Algorithmus:

- Suche den obersten Ausdruck der Form $(\ldots(f\ t_1)\ t_2)\ldots)\ t_n)$, so daß f nicht von der Form $(s_1\ s_2)$ ist.
- Bestimme die Anzahl m der von f erwarteten Argumente. Wenn genügend viele Argument vorhanden sind (d.h. $m \leq n$), dann betrachte den entsprechenden Unterausdruck.

– Wenn $f \equiv$ `ifthenelse`, dann ist $m = 3$. Wenn das erste Argument ausgewertet ist kann eine case-Reduktion durchgeführt werden, andernfalls wird zunächst eine Transformation für das erste Argument gestartet.

– Wenn f eine eingebaute (arithmetische, Zeichen-) Funktion ist, dann werden zunächst die m Argumente per Transformation von links nach rechts ausgewertet.

– Wenn f eine (normale) Funktion ist, dann wird Definitionseinsetzung durchgeführt.

Das Typsystem und die zugehörige Typüberprüfung von Haskell[2] sorgen dafür, daß Ausdrücke die einen arithmetischen Typ haben, entweder bei Auswertung ein entsprechendes Resultat liefern, oder daß die Auswertung nicht terminiert.

Satz Church-Rosser-II *Wenn für einen Ausdruck e eine Reduktionsfolgen mit einem Resultat e_0 terminiert, dann terminiert die normale Reihenfolge der Auswertung von e und liefert als Resultat genau e_0.*

Damit haben wir einen weiteren wichtigen Baustein: Wenn wir eine Menge von Funktionsdefinitionen haben, und ein Ausdruck läßt sich zu einem Resultat auswerten, dann kann man auch eine feste Strategie wählen, um dieses Resultat zu berechnen. D.h. jetzt haben wir eine eindeutige Berechnungsvorschrift, die aus einem Ausdruck den Wert berechnet. Damit gilt:

> Eine Menge von Funktionsdefinitionen, ein Ausdruck und die Festlegung auf eine ausführbare Auswertungsstrategie ergibt eine Berechnungsvorschrift.

Im allgemeinen wählen wir die normale Reihenfolge der Auswertung.

Die Sätze von Church-Rosser erlauben uns die Freiheit auch dann von einem Algorithmus zu sprechen, wenn die Reihenfolge der Auswertung nicht festgelegt ist, und wir uns nur dafür interessieren, daß es ein Resultat gibt.

Beispiel 6. Der Satz von Church-Rosser-II gilt nicht für die applikative Reihenfolge der Auswertung:

```
nt x = nt x
proj x y = x
```

Der Ausdruck `proj 0 (nt 1)` hat unter der applikativen Reihenfolge der Auswertung kein Resultat, denn diese Auswertung terminiert nicht. Es gibt aber eine Reduktionsreihenfolge, die zuerst die Definitionseinsetzung verwendet und dann sofort das Resultat 0 liefert.

[2] werden wir noch behandeln

Eine optimale[3] Zahl von Reduktionen bei Transformationen wird erreicht, wenn man bei der normalen Reihenfolge darauf achtet, welche Unterausdrücke kopiert wurden, und deren Transformationen parallel macht. In Implementierungen wird dies dadurch erreicht, daß keine Kopie eines Ausdrucks gemacht wird, sondern intern ein gerichteter Graph statt eines Baumes erzeugt wird. Diese Auswertungsstrategie wird von Haskell verwendet. Sie wird *verzögerte Auswertung* oder auch *lazy* genannt. Der Satz von Church-Rosser-II gilt auch noch für die Graphvariante der normalen Reihenfolge, d.h. für verzögerte Auswertung.

Legt man die einfache Syntax und die beiden Reduktionsregeln zugrunde, dann gilt:

Aussage 2. *Die normale Reihenfolge der Auswertung mit der Vermeidung von Doppelauswertung ergibt eine optimale Anzahl von Reduktionen.*

Im Beispiel `quadrat`$(4+5)$ sieht die Reduktion folgendermaßen aus:

Beispiel 7. `quadrat`$(4+5) \rightarrow (4+5)^{(1)} * (4+5)^{(1)} \rightarrow 9 * 9 \rightarrow 81$, wobei mit $(4+5)^{(1)}$ angedeutet werden soll, daß dies intern derselbe Ausdruck ist.

2.2 Auswertungsprozesse

Die Fakultätsfunktion mit der Definition

$$0! := 1$$
$$n! := n * (n-1)!$$

haben wir schon definiert:

```
fakultaet x  = if x == 1 then 1
                      else x*(fakultaet (x-1))
```

Den Prozeß der Auswertung können wir veranschaulichen, wenn wir die Folge der Transformationen betrachten, die zum Ergebnis führt. Wir nehmen dafür die normale Reihenfolge.

```
(fakultaet 6)
(* 6 (fakultaet 5))
(* 6 (* 5 (fakultaet 4)))
(* 6 (* 5 (* 4 (fakultaet 3))))
(* 6 (* 5 (* 4 (* 3 (fakultaet 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (fakultaet 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
```

[3] unter den bisherigen Voraussetzungen

```
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Dieses Verhalten eines Auswertungsprozesses nennt man auch *linear rekursiv.*

Charakteristisch ist, daß jeweils nur eine rekursive Funktionsanwendung auftritt, der Gesamtausdruck aber beliebig groß werden kann.

Eine alternative Berechnung der Fakultät erhält man wenn man folgende Ersetzung iteriert:

Produkt	$\Rightarrow$	Produkt $*$ Zähler
Zähler	$\Rightarrow$	Zähler $+ 1$

```
 fakultaet_lin n =  fakt_iter 1 1 n

fakt_iter  produkt  zaehler  max =
               if zaehler > max
               then produkt
               else (applikativ_2 fakt_iter) (zaehler * produkt)  (zaehler + 1) max
```

Die Funktion `applikativ_2` bewirkt, daß die Funktion `(applikativ_2 fakt_iter)` die gleichen Resultate liefert, aber sich so verhält als wenn man die Funktion `fakt_iter` in applikativer Reihenfolge auswertet. Ihre Definition ist im Anhang. Der Auswertungsprozeß unter Verwendung der applikativen Reihenfolge der Auswertung sieht so aus:

```
(fakultaet 6)
(fakt_iter 1 1 6)
(fakt_iter 1 2 6)
(fakt_iter 2 3 6)
(fakt_iter 6 4 6)
(fakt_iter 24 5 6)
(fakt_iter 120 6 6)
(fakt_iter 720 7 6)
720
```

Diese Art eines Prozesses nennt man *iterativ.* Charakteristisch ist, daß jeweils nur eine rekursive Funktionsanwendung auftritt, die Größe des jeweiligen Gesamtausdrucks aber beschränkt ist. Analog ist *endrekursiv* definiert: wenn die Auswertung eines Ausdrucks $(f\ a_1 \ldots a_n)$ nach einigen Auswertungen als Wert einen Ausdruck der Form $(f\ b_1 \ldots b_n)$ auswerten muß. Hierbei sollten die Argumente a_i, b_i bereits Resultate sein, d.h. völlig ausgewertet sein.

Bei der Verwendung von Haskell ist zu beachten, daß die normale Reihenfolge der Auswertung verwendet wird, die zwar höchstens soviele Reduktionsschritte wie die applikative Reihenfolge benötigt, aber in gewissen Fällen einen linear rekursiven Prozeß erzeugt statt eines iterativen Prozesses. Wir werden später noch sehen, wie man dies in Haskell eleganter und besser modelliert.

2.2.1 Baumrekursion

Beispiel 8. Berechnung der Fibonacci Zahlen :

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

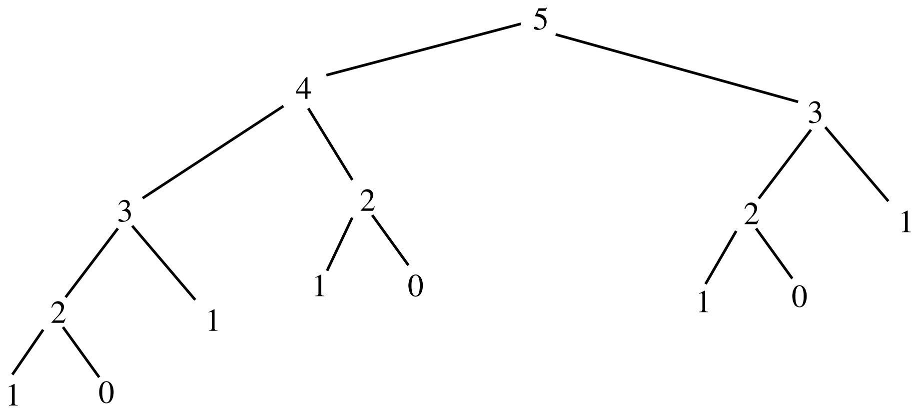
Die Fibonacci Zahlen sind definiert durch folgende rekursive Gleichung:

$$Fib(n) := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sonst} \end{cases}$$

```
fib n = if n <= 0 then 0
        else if n == 1 then 1
             else fib (n-1) + fib(n-2)
```

Der induzierte Prozeß ergibt folgende Ausdrücke

```
fib(5)
fib(4) + fib(3)
(fib(3)+fib(2))  + fib(3)
((fib(2)+fib(1))+fib(2))  + fib(3)
(((fib(1)+fib(0))+fib(1))+fib(2))  + fib(3)
(((1+1)+fib(1))+fib(2))  + fib(3)
((2+fib(1))+fib(2))  + fib(3)
((2+1)+fib(2))  + fib(3)
(3+fib(2))  + fib(3)
(3+(fib(1)+fib(0)))  + fib(3)
(3+(fib(1)+fib(0)))  + fib(3)
```



Einen solchen Prozeß nennt man auch *Baumrekursion.* Charakteristisch ist daß die Ausdrücke unbegrenzt wachsen können und daß mehrere rekursive Aufrufe einer Funktion vorkommen. Diese sollten nicht geschachtelt sein, d.h. in den Argumenten einer rekursiven Funktion sollten keine rekursiven Funktionen mehr vorkommen.

Bemerkung. Geschachtelte Rekursion
Der allgemeine Fall ist geschachtelte Rekursion, obwohl er normalerweise selten benötigt wird, denn die damit definierten Funktionen sind i.a. nicht effizient berechenbar. Ein Beispiel ist die Ackermannfunktion, die folgendermaßen definert ist:

$$
\begin{array}{lcl}
ack\ 0\ y & := & 1 \\
ack\ 1\ 0 & := & 2 \\
ack\ x\ 0 & := & x+2 \quad \text{für } x \geq 2 \\
ack\ x\ y & := & ack(ack\ (x-1)\ y)\ (y-1)
\end{array}
$$

Diese Funktion wächst sehr schnell und benötigt sehr viele Ressourcen zur Berechnung. Sie wurde konstruiert, um nachzuweisen, daß es sehr schnell wachsende Funktionen gibt, die sich nicht mit Baumrekursion berechnen lassen. Sie hat auch Anwendung in der Komplexitätstheorie.

Übungsaufgabe 3. Werte `ack` für verschiedene Argumente aus. Insbesondere werte `ack 5 2`, `ack 6 2`, `ack 7 2` aus. Unter Benutzung der Statistiken in Hugs schätze den Aufwand für `(ack 4 3)` und `(ack 5 3)`.

2.3 Analyse von Algorithmen

Wir analysieren Algorithmen hinsichtlich ihrer Qualität und ihres Ressourcenbedarfs

Beispiel 9. Die Berechnung von fib (5) ist leicht zu analysieren:

fib(3) wird 2 mal berechnet
fib(2) wird 3 mal berechnet
fib(1) wird 5 mal berechnet

Genauer: Die Analyse der Berechnung von fib(n) für $n \geq 2$ zeigt, daß fib(1) jeweils fib(n)-mal berechnet wird.

Es gilt, daß fib(n) $\approx \frac{\Phi^n}{\sqrt{5}}$ wobei $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$

D.h. der Wert von `fib` wächst exponentiell. Die Anzahl der Reduktionen ist ebenfalls exponentiell abhängig von n, d.h. die Laufzeit der Funktion `fib` ist exponentiell abhängig von n. Der Ausdruck selbst wächst nicht so schnell: Man kann die Größe linear halten in n.

2.3.1 Iterative Version von Fib

Es gibt eine einfache Idee, die Berechnung von `fib` zu beschleunigen. Eine einfache Beobachtung dazu ist, daß man zur Berechnung von fib(n) höchstens die Werte aller fib(i) benötigt für alle $1 \leq i \leq n$. D.h. die sukzessive Berechnung einer Wertetabelle ergibt bereits einen Algorithmus, der schneller ist als `fib`. Eine einfache, verbesserte Variante ist die Idee, sich jeweils nur fib($n-1$) und fib($n-2$) zu merken und daraus den nächsten Wert fib(n) zu berechnen.

Rechenvorschrift: $(a, b) \rightarrow (a + b, a)$

```
fib_lin n =  (fib_iter 1 0 n)
--   Idee:   fib_iter  FIB(i) FIB(i-1) n
fib_iter a b zaehler =  if zaehler ==  1
                        then a
                        else applikativ_3 fib_iter (a + b) a (zaehler - 1)
```

Prozeß für `(fib_iter 5)`:

```
(fib_lin 5)
(fib-iter 1 0 5)
(fib-iter 1 1 4)
(fib-iter 2 1 3)
(fib-iter 3 2 2)
(fib-iter 5 3 1)
5
```

Offenbar benötigt diese Version von fib linear viele Reduktionen in Abhängigkeit von n, und die Größe aller Ausdrücke, die während des Prozesses erzeugt werden, hat ein obere Schranke, also ist der Platzbedarf konstant (d.h. unabhängig) von n. Allerdings nur, wenn man die Größe der Darstellungen der Zahlen vernachlässigt.

Übungsaufgabe 4. Binomialkoeffizienten sind definiert durch:

$$\texttt{bk}\ i\ j := \begin{cases} 1 & \text{wenn } i = j \text{ oder } j = 0 \\ \texttt{bk}(i-1)\ j + \texttt{bk}\ (i-1)\ (j-1) & \text{wenn } 0 < j < i \end{cases}$$

Eine alternative Berechnung ist: $\texttt{bk}\ i\ j = \dfrac{i * \ldots * (i-j+1)}{j * \ldots * 1}$.

Programmiere beide Methoden und vergleiche die beiden Auswertungsprozesse.

2.3.2 Terminierung: Hält ein Algorithmus an?

Beispiel 10. Folgende Funktion ist leicht zu definieren, aber deren Terminierung für alle natürlichen Zahlen ist unbekannt. Diese Funktion ist unter dem Namen($(3n+1)$-Funktion) bekannt. Die "Collatz Vermutung" ist die Vermutung, daß diese Funktion für jede natürliche Zahl als Eingaben anhält.

```
drein x = if x == 1 then 1
             else if geradeq x then drein (x ‘div‘ 2)
                    else drein (3*x+1)

geradeq n =  (rem n 2) == 0
```

Die Terminierung dieser Funktion wurde bereits überprüft für alle Zahlen $< 2^{40}$.

Die Frage, ob eine Funktion für eine Menge von Eingaben anhält, sollte man sich beim Algorithmenentwurf stellen, denn sonst ist kein Resultat definiert. Normalerweise entwirft man Algorithmen gleich so, daß sie anhalten, da man das Problem so strukturiert hat, daß man Probleme auf Teilprobleme zurückführt.

Leider zeigt die Theorie, daß eine Programmiersprache, die von vorneherein alle nichtterminierenden Programm ausschließt, nicht ausreicht zur Formalisierung aller Berechnungsverfahren. Somit muß man damit leben, daß es Programme geben kann, die nicht anhalten.

Terminierungsnachweise führt man normalerweise mit Induktion. In einer funktionalen Sprache läßt sich das besonders einfach hinschreiben.

Beispiel 11. Beh: `fib` terminiert mit einer ganzen Zahl: Induktionsbasis ist 0 und 1: `fib`(0) und `fib`(1) terminieren und ergeben jeweils das Resultat 1.

Induktionsschritt: Für $n > 1$ gilt:

`fib`(n) reduziert zu `fib`$(n-1)+$`fib`$(n-2)$. Da $n-1$ und $n-2$ im Bereich der Induktionshypothese liegen, terminieren diese Aufrufe mit einer Zahl. Anschließende Addition terminiert ebenfalls

Beispiel 12. Die einfachen Ansätze zum Terminierungsbeweis für die $3n+1$-Funktion scheitern, da die Argumente unregelmässig größer und kleiner werden. Offenbar kann man Terminierung leicht zeigen für Zahlen der Form 2^n, aber bereits für ($drein$ 3) ist man auf Ausprobieren angewiesen.

2.3.3 Ressourcenbedarf von Algorithmen, Effizienz, Wachstumsraten und Größenordnungen

Wir untersuchen den Ressourcenbedarf von Algorithmen. Die wichtigsten Ressourcen sind:

- Zeit
- Platz

Es gibt auch weitere Ressourcen / Maße, die für Algorithmen und Programme eine Rolle spielen:

- Aufwand zur Erstellung des Programms
- organisatorischer Aufwand zur Nutzung
- Aufwand zum Portieren eines Programms in andere (Rechner-)umgebungen.
- Anzahl benötigter Prozessoren bei parallelen Algorithmen,
- Anzahl der Aufrufe bestimmter Unterprozeduren (Dienste): Datenbankzugriffe, Kommunikation: Kanäle, usw.
- Größe des Programms
- Aufwand für Verifikation / Änderung eines Programms
- Kosten innerhalb eines organisatorischen Rahmens

Alle hier erwähnten Ressourcen spielen in der praktischen Programmierarbeit eine Rolle. Da die Rechnerentwicklung ständig schnellere Prozessoren und leistungsfähigere Speicherhardware entwickelt, sind die Ressourcen Zeit /Platz nicht mehr so dominant wichtig. Ein Beispiel sind graphische Benutzeroberflächen, oder Übertragung und Bearbeitung von Bildern statt Zeichen, was einen großen Platzbedarf bedeutet. Als prominentes und aktuelles Beispiel kann man"Java" nennen, bei dem die übliche Vorgehensweise ist, daß man ein interpretierbaren File (Bytecode) von anderen Rechnern lädt und diesen dann von einem Interpreter ausführen läßt. Hier ist die Ausführungszeit des Algorithmus selbst nicht die bestimmende Größe, sondern die leichte Ausführbarkeit auf verschiedensten Rechnern steht im Vordergrund, danach kommt der Gesamtbedarf an Zeit (inklusive Übertragungszeit), und danach erst der Zeitbedarf des Algorithmus selbst.

Bemerkung. Die Optimierung (Tuning) von Programmen hat das Ziel, durch Abänderungen den Ressourcenbedarf zu reduzieren. Hier muß man warnen vor zu unvorsichtiger Optimierung. Man sollte folgendes beachten:

- Wie oft wird dieses Programm benötigt? Lohnt sich der Optimieraufwand / Kosten im Vergleich zur Gesamtlaufzeit des Programms?
- Das Programm wird normalerweise durch Optimierungen unübersichtlicher, d.h. schlechter lesbar und wartbar.
- Wenn zu erwarten ist, daß mit späteren Versionen des benutzten Compilers (Interpreters) oder eines zugrundeliegenden Systems der Ressourcenbedarf verbessert wird, dann sollte man abwarten.
- Man sollte sich darüber im klaren sein , ob Optimierungen die Portierbarkeit verschlechtern
- Die experimentelle Feststellung des Ressourcenbedarfs von Algorithmen /Programmen ist fehlerbehaftet. D.h. die experimentelle Feststellung, daß ein bestimmtes Vorgehen den Algorithmus verbessert, kann in der nächsten Version falsch sein.

Einige dieser Bemerkungen könnten auch auf Haskell zutreffen, wenn der Compiler optimierende Programmtransformationen vornimmt, die von Version zu Version verschieden gut sind.

Wir betrachten zunächst mal nur die Ressourcen Zeit und Platz und benutzen diese Meßlatte, um bestimmte Verfahren und Algorithmen zu analysieren.

Wir machen eine Vereinfachung und messen folgendes:

Zeit: Anzahl der Transformationsschritte.
Platz: Größe der Ausdrücke

Dies ist machbar für eine feste Eingabe, wir wollen aber auch Aussagen machen für alle möglichen Eingaben, die ein fester Algorithmus bearbeiten kann. Der Aufwand eines Algorithmus wird deshalb angegeben als Funktion der Größe der Eingabe. Im allgemeinen nimmt man als Größe der Eingabe die Größe der tatsächlich verwendeten Repräsentation. Für Algorithmen mit ganzzahliger positiver Eingabe ist als Bezugsgröße die Zahl selbst geeignet. Allerdings muß man dann bei der Angabe der Aufwandsfunktionen diese Bezugsgröße angeben.

Bei der Abschätzung des Zeitbedarfs durch die Anzahl der Reduktionsschritte muß man sich vergewissern, daß dies korrekt ist bzw. wie groß der Fehler ist, den man dabei macht. Es muß gelten, daß eine einzelne Reduktion in konstanter Zeit durchgeführt werden kann, wenn die Definition der Funktionen gegeben ist. Außerdem ist zu beachten, daß der Aufwand für die Reduktion gewisser eingebauter Funktionen unbekannt ist. Auch hier macht man die vereinfachende Annahme, daß diese eine obere Schranke hat. Bei Feinanalysen müsste man den genauen Zeitbedarf aller eingebauten Funktionen (zumindest relativ) kennen (Z.B. Multiplikation, Addition, Konversion von Zahlen,usw.).

Beim Zählen der Reduktionsschritte können wir entweder die verzögerte Reduktion verwenden, oder auch eine andere Strategien, die keine unnötigen oder doppelten Auswertungen macht.

Bei der Abschätzung des Platzbedarfs muß man darauf achten, daß der benötigte Platz für Zahlen nicht konstant ist, wenn man Zahlen beliebiger Größe darstellen kann. Wenn man allerdings nur mit ganzen Zahlen rechnet, die z.B. $< 2^{32}$ sind, dann kann man deren Platzbedarf als konstant annehmen. Dies gilt z.B. auch für die üblichen Darstellungen von Fließkommazahlen.

Der optimale Platzbedarf im Sinne der Größe der Ausdrücke ist nicht mit einer festen Strategie ermittelbar oder erreichbar. Strenggenommen müsste man über alle Reduktionen minimieren. Oft hat ein iterativer Prozeß den besten Platzbedarf. Wir werden zum Ermitteln des Platzbedarfs die verzögerte Reihenfolge verwenden, und evtl. Abänderungen der Reduktionsstrategie in den Definitionen der Funktionen markieren.

Wir schreiben den Ressourcenbedarf eines Algorithmus `alg` als Funktion der Eingabe von der Größe n:

$red_{\mathrm{alg}}(n)$	Anzahl der Reduktionen, die `alg` für eine Eingabe der Größe n durchführt
$Zeit_{\mathrm{alg}}(n)$	Zeitbedarf
$Platz_{\mathrm{alg}}(n)$	Platzbedarf

Beispiel 13. Der Ressourcenbedarf des Algorithmus für `fib` wird gemessen in der Größe der eingegebenen Zahl. Vereinfacht man weiter und zählt nur die Anzahl der rekursiven Aufrufe selbst, so ergibt sich: Anzahl $\leq \mathtt{fib}(n+1)$. Da die Anzahl der nicht-`fib`-Reduktionen konstant ist, ergibt sich: $red_{\mathtt{fib}}(n) \leq c * \mathtt{fib}(n+1)$, wobei c eine Konstante ist. Da $\mathtt{fib}(n)$ exponentiell wächst, $\approx 1.6^n$, gilt dies auch für den Zeitbedarf.

Misst man den Ressourcenbedarf allerdings in der Größe der Darstellung der Zahl n, so sieht man: Die Darstellung benötigt $\lceil \mathtt{log}_{10}(n) \rceil$ Dezimalstellen. D.h. mit diesem Maß wäre der Zeitbedarf $\approx 1.6^{(10^{size(inp)})}$.

Die *Komplexitätstheorie* untersucht die Eigenschaft von Problemen in Bezug auf deren Ressourcenbedarf; Im wesentlichen sind dies Platz und Zeit. D.h. es wird auch untersucht, ob Algorithmen für ein Problem optimal sind, oder ob es theoretisch besser geht oder nicht.

Die Untersuchung von fib leitet etwas in die Irre, denn normale Problemklassen verhalten sich irregulärer als diese einfachen Berechnungen: Es kann ein breite Streuung geben für die Menge aller Eingaben einer bestimmten Größe. Deshalb untersucht man folgende Gütemaße von Algorithmen:

- Ressourcenbedarf im *schlimmsten Fall* (worst-case). Was ist die obere Schranke des Ressourcenbedarfs (Zeit, Platz)?
- Ressourcenbedarf im *besten Fall* (best-case). Was ist die untere Schranke des Ressourcenbedarfs (Zeit, Platz)?
- Ressourcenbedarf im *Mittel* (average case). Was ist der Mittelwert des Ressourcenbedarfs (Zeit, Platz)? Hier ergibt sich das Problem der Bewertung der Fälle bzw. der Wahl einer geeigneten Verteilung der Eingaben.
- Ressourcenbedarf für in der Praxis vorkommende Fälle. Dies ist nicht exakt und unabhängig vom Problem zu formulieren.

Einen Algorithmus nennt man *effizient*, wenn er wenig Ressourcen benötigt, (d.h. wenig Zeit und wenig Speicher ...). Diese Definition ist nicht formal, denn ein Algorithmus soll ja eine Klasse von Problemen lösen. Oftmals sind die beiden Ziele konkurrierend, d.h. die Optimierung des Zeitbedarfs ist teilweise mit einer Verschlechterung des Platzbedarfs erkauft. Der Ressourcenbedarf eines Algorithmus ist zunächst rechnerunbhängig definiert. Nimmt man die vergangene Zeit als Maßstab, so hängt diese natürlich von der Verarbeitungsgeschwindigkeit des benutzten Systems ab.

O-Schreibweise

Die O-Schreibweise wird verwendet, um die Größenordnung von numerischen Funktionen abzuschätzen. Seien $R, f : \mathbb{N}^+ \to \mathbb{R}^+$ zwei Funktionen: Wir schreiben:

$$R(n) = O(f(n)), \quad \text{wenn es eine Konstante } K \text{ gibt, so daß} \\ R(n) \leq K * f(n) \text{ für alle genügend großen} n.$$

Man beachte, daß dies keine Gleichung ist, sondern eine Abschätzung einer Funktion nach oben, und daß diese Abschätzung nicht immer die bestmögliche sein muß.

Beispiel 14.

- Es gilt $\sqrt{n} = O(n)$, denn $\sqrt{n} \leq n$ für alle positiven ganzen Zahlen n.
- Es gilt $p(n) = O(2^n)$ für alle Polynome p.
- Wenn $f(n) = O(c * g(n))$, dann auch $f(n) = O(g(n))$.
- Wenn $c, d > 1$ und e eine Konstante, dann gilt: $f(n) = O(c^{n+e})$ gdw. $f(n) = O(c^n)$
- Wenn $c, d > 1$, dann gilt: $f(n) = O(log_c(n))$ gdw. $f(n) = O(log_d(n))$
- Wenn $c, d > 1$, und $c < d$, dann gilt: $c^n = O(d^n)$, aber nicht $d^n = O(c^n)$.

- Wenn $f(n) = O(g(n))$ und $g(n) = O(h(n)$, dann gilt auch $f(n) = O(h(n))$.
- Wenn eine Funktion f durch eine Konstante nach oben beschränkt ist, dann gilt $f(n) = O(1)$.

Im folgenden benutzen wir immer `log` zur Basis 2.

Beispiel 15. Wenn man `fib` in der Größe der eingegebenen Zahl n, mißt bzw. in der Größe der Darstellung der Zahl d, dann gilt

$$\begin{array}{lcl}
\mathrm{red}_{\mathrm{fib}}(n) & = & O(1.62^n) \\
\mathrm{platz}_{\mathrm{fib}}(n) & = & O(n) \\
\mathrm{red}_{\mathrm{fib_lin}}(n) & = & O(n) \\
\mathrm{platz}_{\mathrm{fib_lin}}(n) & = & O(\mathtt{log}(d)) \\
\mathrm{red}_{\mathrm{fib_lin}}(d) & = & O(10^d) \\
\mathrm{platz}_{\mathrm{fib_lin}}(d) & = & O(d)
\end{array}$$

Es gibt einige Komplexitäten von Algorithmen, die häufig vorkommen:

$O(1)$	konstant
$O(log(n))$	logarithmisch
$O(n)$	linear
$O(n * log(n))$	fastlinear (oder auch n-log-n)
$O(n^2)$	quadratisch
$O(n^k)$	polynomiell
$O(2^n)$	exponentiell

Beispiel 16. Vergleich der Laufzeiten von vier Algorithmen, die $\mathtt{log}(n), n, n^2, 2^n$ Sekunden benötigen.

Größe n / Algorithmus	10	100	1000
$\mathtt{log}_2(n)$	0.000003	0.0000007	0.000001
n	0.00001	0.0001	0.001
n^2	0.0001	0.01	1
2^n	0.001	10^{16} Jahre	$\sim\infty$

Beachte, daß viele Optimierungen, die ein Programm effizienter machen und die in der Praxis relevant sind, sich in der O-Notation nicht bemerkbar machen. Oft sind dies konstante Faktoren, um die ein Algorithmus schneller wird. Z.B. bewirkt die Verwendung eines schnelleren Rechners nur einen konstanten Faktor bei der tatsächlich benötigten Zeit.

Es gibt auch Optimierungen, die die Größenordnung des Zeitbedarfs eines Programm verbessern, aber die Konstante vergrößern, so daß diese Verbesserung praktisch nicht relevant ist.

Die Bestimmung der genauen Komplexität des Ressourcenbedarfs einer Problemklasse ist ein sehr schweres theoretisches Problem, das nur für wenige Problemklassen exakt genug bekannt ist. I.a. sind nur gute obere Abschätzungen bekannt, von denen man vermutet, daß diese der Realität entsprechen, während nichttriviale untere Schranken oft nur sehr schlecht ermittelt werden können.

2.3.4 Ω-Schreibweise

Im Gegensatz zur O-Notation, die Funktionen nach oben abschätzt, wird die Ω-Schreibweise verwendet, um die Größenordnung von numerischen Funktionen nach unten abzuschätzen. Seien $R, f : \mathbb{N}^+ \to \mathbb{R}^+$ zwei Funktionen: Wir schreiben:

$$R(n) = \Omega(f(n)), \quad \text{wenn es eine Konstante } K \text{gibt, so daß} \\ R(n) \geq K * f(n) \text{ für alle genügend großen } n.$$

Analog zu O gilt:

Beispiel 17.

- Es gilt $n = \Omega(\sqrt{n})$, denn $\sqrt{n} \leq n$ für alle positiven ganzen Zahlen n.
- Es gilt $\sqrt{n} = \Omega(1)$.
- Es gilt $2^n = \Omega(p(n))$ für alle Polynome p.
- Wenn $f(n) = \Omega(c * g(n))$, dann auch $f(n) = \Omega(g(n))$.
- Wenn $c, d > 1$ und e eine Konstante, dann gilt: $f(n) = \Omega(c^{n+e})$ gdw. $f(n) = \Omega(c^n)$
- Wenn $c, d > 1$, dann gilt: $f(n) = \Omega(log_c(n))$ gdw. $f(n) = \Omega(log_d(n))$
- Wenn $c, d > 1$, und $c < d$, dann gilt: $d^n = \Omega(c^n)$, aber nicht $c^n = \Omega(d^n)$.
- Wenn $f(n) = \Omega(g(n))$ und $g(n) = \Omega(h(n)$, dann gilt auch $f(n) = \Omega(h(n))$.
- Wenn eine Funktion f durch eine Konstante nach unten beschränkt ist, dann gilt $f(n) = \Omega(1)$.

Beispiel 18. Berechnung von Potenzen b^n für positive ganze Zahlen n.

Die Potenzen sind rekursiv definiert durch

$$b^n := \begin{cases} 1 & \text{falls } n = 0 \\ b * b^{n-1} & \text{sonst} \end{cases}$$

Direkte Kodierung des Algorithmus ergibt ein rekursives Programm:

```
potenz b n =  if  n == 0
              then  1
              else  b * (potenz b (n - 1))
```

Platz- und Zeitbedarf sind von der Größenordnung $O(n)$. Der Zeitbedarf ist ebenfalls von der Größenordnung $O(n)$. Dies läßt sich verbessern, indem man folgende Idee ausnutzt: statt $b^8 = b * b * b * b * b * b * b * b$ berechne

- $b^2 := b * b$
- $b^4 = b^2 * b^2$
- $b^8 = b^4 * b^4$

Als allgemeine Rechenvorschrift halten wir fest:

$$b^n := \begin{cases} 1 & \text{falls } n = 0 \\ (b^{n/2})^2 & \text{falls } n \text{ gerade und } \geq 2 \\ b * b^{n-1} & \text{falls } n \text{ ungerade} \end{cases}$$

```
potenz_log b n = if n == 0 then 1
                 else   if geradeq n
                        then quadrat (potenz_log b (n `div` 2))
                        else b * (potenz_log b (n - 1))

geradeq n =  (rem n 2) == 0
```

Der Platz- und Zeitbedarf in der Zahl n ist $O(log(n))$, z.B. für $n = 1000$ benötigt dieser Algorithmus nur 14 Multiplikationen. Beachtet man noch den extra Aufwand für die Multiplikationen, die bei beliebigen langen Zahlen (Integer) mit der Länge der Zahl steigt, so ist der wahre Aufwand höher: je nachdem wie diese Multiplikation implementiert ist. Bei normaler Implementierung: $O((log(n)^3)$,

2.3.5 Der größte gemeinsame Teiler

Idee zur Berechnung von $\mathrm{ggT}(a, b)$:

Teile a durch b gibt Rest r,
wenn $r = 0$, dann $\mathrm{ggT}(a, b) := b$
wenn $r \neq 0$, dann berechne $\mathrm{ggT}(b, r)$.

Beispiel 19. $\mathrm{ggT}(30, 12) = \mathrm{ggT}(12, 6) = 6$

```
ggt a b =     if  b == 0
              then a
              else ggt b (rem a b)
```

Es gilt:

Satz Lamé : *Wenn der Euklidische* ggt*-Algorithmus k Schritte benötigt, dann ist die kleinere Zahl der Eingabe $\geq fib(k)$.*

Daraus kann man sofort herleiten, daß dieser Algorithmus Platz- und Zeitbedarf: $O(log(n))$ hat: Wenn n die kleinere Zahl ist und der Algorithmus k Schritte benötigt, dann ist $n \geq fib(k) \approx \Phi^k$. Also ist $k = O(log(n))$

2.3.6 Test auf Primzahl-Eigenschaft

Das Problem ist: Gegeben eine Zahl n. Frage: ist n eine Primzahl?

1. Methode: finde den kleinsten Teiler:

```
--    ad-hoc test auf Prizahleigenschaft
primzahlq n =  n == (kleinster_teiler n)
kleinster_teiler n = finde_teiler n 2

finde_teiler n pruef_teiler =
     if n < (quadrat pruef_teiler)   then    n
     else   if    teiltq pruef_teiler n
            then  pruef_teiler
            else  finde_teiler n (pruef_teiler + 1)

teiltq a b =    0 ==  (rem b a)
```

   Der Zeitbedarf dieser Funktion ist von der Größenordnung $O(\sqrt{n})$.

2. Benutze kleinen Fermatschen Satz.

   Satz Fermat : *Wenn n eine Primzahl ist und $1 < a < n$, dann ist $a^n \equiv a(mod\ n)$*

```
potenzmod b e m =
   if  e == 0 then  1
```

```
  else if geradeq e
       then rem (quadrat (potenzmod b (e `div` 2) m))
                m
       else rem (b * (potenzmod b (e - 1) m))
                m
--  randomInts ist in Library random.hs

fermat_test_it n rnd =
        let  a = (rnd `mod` n)
  in  (potenzmod a n n) == a

fermat_test n  = and    (map (\rnd -> fermat_test_it n rnd)
                             (take 100 (randomInts 2 n))))
```

Dies ist ein probabilistischer Algorithmus, der mit einer sehr grossen Wahrscheinlichkeit das richtige Ergebnis liefert. Die Funktionen in dem Ausdruck `map ...` werden wir noch besprechen.

Eigenschaft: Die Antwort: `False` $\equiv$ "ist keine Primzahl" ist immer korrekt, während `True` etwas anderes bedeuten kann als "ist eine Primzahl" .

Zeitverbrauch pro Test: $O(log\ n)$.

Übungsaufgabe 5.

- Für welche Konstanten $a, b, > 1$ gilt $a^{b^n} = O(b^{a^n})$?
- Gebe eine Haskellfunktion f an, die als Argument eine Zahl hat, und für deren Anzahl der Reduktion gilt $red_f(n) = O(3^{3^n})$, aber nicht $red_f(n) = O(2^n)$.
  1. mit arithmetischen Funktionen.
  2. nur mit den eingebaute arithmetische Funktionen $+, -, ==$.

2.4 Lokale Funktionsdefinitionen, anonyme Funktion, Lambda-Ausdrücke

Man kann lokale Funktionen mit *let* definieren indem man statt eines Bezeichners einen Funktionsbezeichner mit formalen Parametern hinschreibt: Die Fakultätsfunktion läßt sich dann folgendermaßen definieren:

```
let fak x = if x <= 1 then 1 else x*(fak (x-1)) in fak 5
```

`ergibt: 120.`

Wir betrachten zunächst die einfachen Fälle und setzen dann am Ende alles wieder zusammen, so daßobiges Beispiel vollständig klar ist.

Man kann Funktionen wie ein Datenobjekt definieren: Die Syntax für einen Lambda-Ausdruck ist:

$$\backslash x_1 \ldots x_n \ -> \ \langle \text{Ausdruck} \rangle$$

In der Syntax hat obiger Lambda-Ausdruck den Status eines Ausdrucks. Damit kann man z.B. die Quadrat-funktion auf eine weitere Weise definieren:

```
quadrat = \x  -> x*x
```

Wir bezeichnen im folgenden die syntaktische Gleichheit von Ausdrücken mit $\equiv$, d.h. $s \equiv t$, wenn s und t die gleichen Ausdrücke sind (inklusive aller Namen).

Definition 6 Freie Vorkommen von Variablen in t. Für den Bindungsbereich der Variablen in einem Lambda-Ausdruck definiert man zunächst den Begriff *freie Variablen*, bzw. freie Vorkommen von Variablen. Als Variablen kann man nur Bezeichner (Namen) verwenden, die keine Konstruktoren sind (d.h. True, False, Zahlenkonstanten und Characterkonstanten sind nicht zulässig)

- x ist frei in x.
- x ist frei in $(t\ t')$, wenn x frei ist in t oder t' ist.
- x ist nicht frei in t, wenn t eine Name ist und $x \not\equiv t$ ist.
- x ist frei in `if` t_1 `then` t_2 `else` t_3, wenn x frei ist in t_1, t_2, oder t_3.
- x ist frei in $\backslash x_1 \ldots x_n \ -> \ t$, wenn $x \not\equiv x_i$ für $i = 1, \ldots n$ und x frei in t ist.

Für die freien Variablen eines Ausdrucks t schreibt man auch $FV(t)$. *Gebundene* Vorkommen von Variablen sind Vorkommen der Variablen x_i im Rumpf t des Lambda-Ausdrucks $\backslash x_1 \ldots x_n \ -> \ t$.

Beispiel 20. $FV(\backslash\ x\ -> (x * y)) = FV(x * y) \setminus \{x\} = \{x, *, y\} \setminus \{x\} = \{y, *\}$.

Die Namen der global definierten Namen (außer Konstruktoren) werden auch als freie Variablen betrachtet. Betrachtet man das ganze Programm, so sind diese dann gebunden (siehe unten Beispiel 23).

Die Transformationsregel für Lambda-Ausdrücke ist die gleiche wie für Definitionseinsetzung, allerdings muß man jetzt darauf achten, daß

1. nur die freien Vorkommen der formalen Parameter ersetzt werden (nicht die gebundenen Vorkommen).

2. freie Variablen (des einzusetzenden Ausdrucks) nicht in einen falschen Bindungsbereich geraten.

Dies kann man teilweise nur garantieren, wenn man die *Umbenennung von gebundenen Variablen* (formalen Parameter) erlaubt. Die Namen der formalen Parameter spielen keine Rolle, deshalb kann man auch andere Namen dafür verwenden, solange man keine Namenskonflikte innerhalb des Ausdrucks provoziert.

Definition 7. Umbenennung von Variablen:

$\backslash x_1 \ \ldots \ x_n \ -> e$
$\longrightarrow$
$\backslash y_1 \ \ldots \ y_n \ -> e[y_1/x_1, \ldots, y_n/x_n]$,
wenn y_i Variablennamen sind, die nicht in e vorkommen, und wenn bei der Ersetzung von x_i nur die freien Vorkommen von x_i in e ersetzt werden.

Wenn man (rekursiv) in allen Lambda-unterausdrücke eines Ausdrucks diese Variablen umbenennt, dann nennt man das Umbenennung aller gebundenen Variablen.

Beachte, daß freie Variablen eines Ausdrucks nicht umbenannt werden.

Beispiel 21. Der Ausdruck $\backslash\ x-> (\backslash y-> x * ((\backslash x-> x + y)\ 5))$ kann umbenannt werden durch Ersetzen des inneren x durch `xx`:

$\backslash\ x-> (\backslash y-> x * ((\backslash xx-> xx + y)\ 5))$.

Nicht erlaubt ist die Umbenennung von y in x. Dies würde ergeben:

$\backslash\ x-> (\backslash x-> x * ((\backslash x-> x + x)\ 5))$.

Umbenennung des äußeren x ergibt: $\backslash\ xx-> (\backslash y-> xx * ((\backslash x-> x + y)\ 5))$. Dies kann man auch testweise in Haskell eingeben; man erhält für den falschen Ausdruck auch andere Ergebnisse.

```
>   (\x -> (\y -> x*((\x -> x+y) 5))) 2 2
14

>  (\x -> (\x -> x*((\x -> x+x) 5))) 2 2
20

>  (\xx -> (\y -> xx*((\x -> x+y) 5))) 2 2
14
```

Definition 8. Transformationsregel für Anwendungen von Lambdaausdrücken auf Argumente:

$((\backslash\ x_1 \ldots x_n\ -\!>\ e)\ s_1 \ldots s_n)$
$\rightarrow$
$e'[s_1/x_1, \ldots, s_n/x_n]$,
wobei e' der Ausdruck e ist nach einer Umbenennung aller gebundenen Variablen.

Man kann auch mit weniger Umbenennungen auskommen, falls die gebundenen Variablen in e nicht in Konflikt mit freien Variablen geraten können.

Bei der normalen Auswertung werden anonyme Funktionen (d.h. Lambdaausdrücke) wie Funktionsnamen behandelt.

2.4.1 Lokale Variablen mit let

Man kann lokale Variablen mit *let* definieren.

Die Syntax für `let`-Ausdrücke ist:

let $\{\langle\text{Name}\rangle = \langle\text{Ausdruck}\rangle\}^+$ in $\langle\text{Ausdruck}\rangle$.

Da die Erklärung lokaler Namen (Bezeichner, Variablen) geschachtelt auftreten kann, braucht man Regeln, um den Geltungsbereich von lokalen Namen (Bezeichnern, Variablen) zu ermitteln.

> **Regel**: let $x = s$ in t bindet alle freien Vorkommen der Variablen x in t an den Wert s.

Definition 9. Zur Bestimmung der freien Variablen in t benötigt man jetzt eine zusätzliche Regel:

x ist frei in `let` $z = t$ in t', wenn $x \not\equiv z$ und x frei ist in t oder in t'.

Im folgenden bezeichnen wir mit $s[x]_p$ einen Ausdruck, der an einer Stelle p das freie Vorkommen von x hat

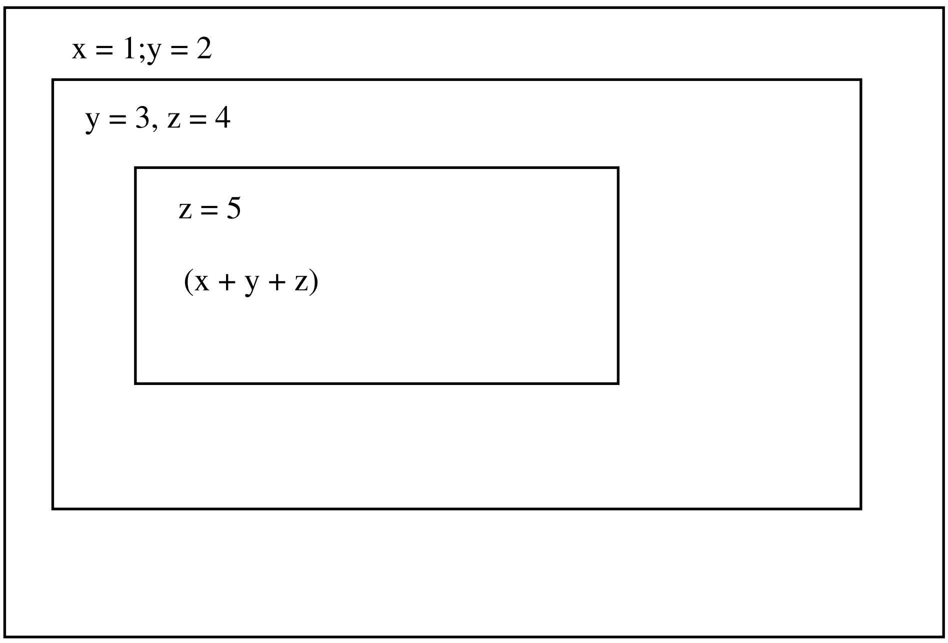
Definition 10. Die (nichtdeterministischen) Transformationsregeln für `let`-Ausdrücke sind folgendermaßen definiert:
`let` $\{x_1 = s_1; \ldots; x_n = s_n\}$ `in` $e[x_i]_p$
$\rightarrow$
`let` $\{x_1 = s_1; \ldots; x_n = s_n\}$ `in` $e'[s_i]_p$.
Hierbei ist e' der Ausdruck e nach Umbenennung der gebundenen Variablen.

Als extra Regel nehmen wir noch hinzu: `let` $\{x_1 = s_1; \ldots ; x_n = s_n\}$ `in` $e \to e$, wenn e kein freies Vorkommen der Variablen x_i hat.

Da Ausdrücke der Form ((`let ... in` ...) s) auftreten können, geben wir auch hierfür eine Regel an:
((`let` $\{x_1 = t_1; \ldots ; x_n = t_n\}$ `in` t) s)
$\to$
(`let` $\{y_1 = t_1\sigma; \ldots ; y_n = t_n\sigma\}$ `in` ($t\sigma$ s))
wobei $\sigma := [y_1/x_1, \ldots , y_n/x_n]$ und y_i neue und verschiedene Variablennamen sind.

Die normale Reihenfolge der Auswertung wird so gesteuert, daß bei Auswertung eines Ausdrucks der Form ((`let ... in ...`) s) zunächst das s in nach obiger Regel in den `let`-Ausdruck geschoben wird. Bei Auswertung eines let-Ausdrucks wird der Ausdruck nach dem `in` ausgewertet. Will man für eine Variable deren Bindung und den Wert ermitteln (z.B. weil die normale Reihenfolge dieses x auswerten will, so ist der an x gebundene Ausdruck im **innersten** Bindungsbereich zu finden, der x bindet. Diese Bindungen nennt man auch *Bindungsumgebungen* oder Blockstruktur von Programmen.

Beispiel 22. `let {x = 1;y = 2} in (let {y =3;z = 4}`
`in (let z = 5 in (x+y+z)))`



ergibt: 1 + 3+ 5 = 9

Beachte, daß in manchen anderen Programmiersprachen `let`-Ausdrücke so definiert sind, daß der Bindungsbereich von x genau der Ausdruck t ist.

Beispiel 23. Ein Programm mit den Definitionen $f_i := e_i \mid i = 1, \ldots, n$ und dem auszuwertenden Ausdruck e kann als "großes" `let` betrachtet werden:

$\texttt{let}\ \{f_1 := e_1; \ldots; f_n := e_n\}\ \texttt{in}\ e$

In dem Ausdruck e sind diese Variablennamen dann als frei zu betrachten.

Die rekursive Bindung in einem let macht nicht immer Sinn, und kann leicht zur Nichtterminierung führen.

Beispiel 24. Der Ausdruck $\texttt{let}\ x = x\ \texttt{in}\ x$ führt bei Auswertung zu Nichtterminierung: Die einzig mögliche Ersetzung erzeugt wieder denselben Ausdruck $\texttt{let}\ x = x\ \texttt{in}\ x$

Beispiel 25. Die Funktion $f(x, y) := x(1+xy)^2+y(1-y)+(1+xy)(1-y)$ läßt sich effizienter berechnen, wenn man definiert:

$$\begin{array}{lll} a & := & 1 + xy \\ b & := & 1 - y \\ f(x,y) & := & xa^2 + yb + ab \end{array}$$

```
f x y = let a = 1 + x *y
            b  = 1 - y
          in
  x * a^2 +  y* b + a*b
```

Beispiel 26. Der folgende Ausdruck hat zwei Bindungsbereiche für verschieden erklärte, aber gleich benannte Bezeichner x.

```
let x = 10 in (let x = 100 in (x+x)) + x
```

Schreibt man die Bindungsbereich ebenfalls hin, so ergibt sich:

```
let x1 = 10 in (let x2 = 100 in (x2+x2)) + x1
```

Dieser Term wertet zu 210 aus.

Die Eingabe eines Ausdrucks wie

```
let x = (x*x) in (x+x)
```

führt zu Nichtterminierung.

Beispiel 27.

```
let    y = 20*z
       x = 10+y
       z = 15
           in x
```

Oder in anderer Syntax:

```
  let   {y = 20*z;  x = 10+y ; z = 15}   in x
```

Wertet aufgrund der obigen Bindungsregel aus zu : 310.

2.4.2 Lokale Funktionen mit `let`

Die erweiterte Syntax für `let`-Ausdrücke mit lokalen Funktionsdefinitionen ist:

$$\text{let } \{\langle\text{Name}\rangle\langle\text{Variable}\rangle\ldots\langle\text{Variable}\rangle = \langle\text{Ausdruck}\rangle\}^{+} \text{ in } \langle\text{Ausdruck}\rangle.$$

Wir sehen diese lokale Definition einfach an als schöne Möglichkeit, die etwas unschönere lokale Definition als anonyme Funktion zu umgehen. Als Transformation benutzen wir:

`let` $\ldots\ f\ x_1 \ldots x_n = r$ `in` $\ldots\ s$
$\rightarrow$
`let` $\ldots\ f = \backslash x_1 \ldots x_n -> r\ \ \ldots$ `in` s

Analog kann man das für `lets` mit mehreren lokalen Variablen durchführen.

Beispiel 28. Einige Schritte der Reduktion des folgenden Beispiels:

```
(let fak x = if x <= 1 then 1 else x*(fak (x-1)) in fak)  5
->
(let fak = \x ->  if x <= 1 then 1 else x*(fak (x-1)) in fak)  5
->
let fak = \x ->  if x <= 1 then 1 else x*(fak (x-1)) in (fak 5)
->
let fak = \x ->  if x <= 1 then 1 else x*(fak (x-1)) in
                if 5 <= 1 then 1 else 5*(fak (5-1))
->
let fak = \x ->  if x <= 1 then 1 else x*(fak (x-1)) in
                if False then 1 else 5*(fak (5-1))
->
let fak = \x ->  if x <= 1 then 1 else x*(fak (x-1)) in
                5*(fak (5-1))
....
```

Man sieht: das ist im wesentlichen das gleiche, als wäre `fak` direkt im Programm als globale Funktion definiert.

2.5 Funktionen als allgemeine Methoden

Es gibt einige Algorithmen, die auf (arithmetischen) Funktionen definiert sind, wie z.B. Nullstellenbestimmung, Integrieren, Differenzieren, ..., und die wir hier besprechen wollen.

Formuliert man diese Algorithmen, so ist es bequem, wenn man die Funktionen direkt als Argumente an eine definierte Funktion übergeben kann, die die allgemeinere Methode implementiert.

Beispiel 29. Bestimmung von Nullstellen einer stetigen Funktion mit Intervallhalbierung

Idee: Sei $f(a) < 0 < f(b)$,

- wenn $f((a+b)/2) > 0$, dann suche die Nullstelle im Intervall $[a, a+b/2]$.
- wenn $f((a+b)/2) < 0$, dann suche die Nullstelle im Intervall $[(a+b)/2, b]$.

```
suche_nullstelle f a b genau =
 let m = (mittelwert a b)
   in if abs (a - b) < genau
      then m
      else let fm = f m
              in if fm > 0
                 then suche_nullstelle f a m genau
                 else if fm < 0
                      then suche_nullstelle f m b genau
                      else m

intervall_halbierung f a b genau =
 let  fa = f a
             fb = f b
          in
   if fa < 0 &&   fb > 0
           then suche_nullstelle f a b genau
   else if fa > 0 && fb < 0
                then suche_nullstelle f b a genau
                 else error ("Werte haben gleiches Vorzeichen" ++
                             (show a) ++ (show b))
```

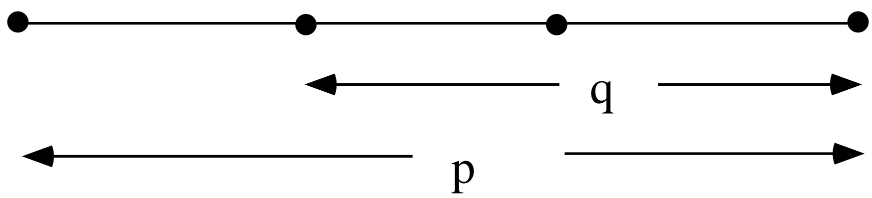
```
>   intervall_halbierung sin 2 4 0.00001

3.14159012
```

Den Aufwand der Intervallhalbierungsmethode kann man bestimmen durch die maximale Anzahl der Schritte: $\lceil log_2(T/L) \rceil$, wobei L die Länge des Intervalls und T die Genauigkeit ist. D.h der Zeitbedarf ist O(log(L/T)), während der Platzbedarf Grössenordnung $O(1)$ hat.

Beispiel 30. Die Methode des goldenen Schnittes

Bestimme das Maximum einer unimodalen Funktion in $[a\ b]$, d.h. einer stetigen Funktion, die nur ein lokales Maximum im Intervall $[a\ b]$ hat und kein Minimum.

Man sieht schnell, daß ein neuer Zwischenpunkt nicht ausreicht. Man benötigt zwei Zwischenpunkte, damit man das Maximum neu einfangen kann. Dazu ist zweimaliges Berechnen des Funktionswertes pro Iterationsschritt notwendig. Eine Verbesserung ist es, nur einmal den Funktionswert pro Iteration zu berechnen:



Die optimale Verkleinerung wird erreicht, wenn $p/q = q/(p-q)$, d.h. $p(p-q) = q^2$, bzw. $q^2 + pq - p^2 = 0$.

Die Lösung ist: $q = -0.5 * p \pm 0.5 * \sqrt{p^2 + 4p^2} = 0.5(\sqrt{5} - 1))p \approx 0.618.. * p$

2.5.1 Funktionen als Ergebnis

Es kann sehr nützlich sein, auch Funktionen als Ausgabe zu erlauben, z.B., wenn man die Nullstellenbestimmung auf zusammengesetzte Funktionen anwenden will. Das einfachste Beispiel ist die Komposition von Funktionen:

```
komp f g x = f (g x)
intervall_halbierung (sin ‘komp‘ quadrat) 0 4 00.1
```

Beispiel 31. Ein anderes Beispiel ist näherungsweises Differenzieren

$$Df(x) := \frac{f(x + dx) - f(x)}{dx}$$

```
ableitung f dx =  \ x -> ((f(x+dx)) - (f x)) / dx
```

Dies liefert als Resultat eine neue Funktion, die die Ableitung annähert.

```
>  ((ableitung (\x -> x^3) 0.001) 5)
74.9292937
```

Hiermit kann man die Newtonsche Methode zur Bestimmung der Nullstellen beliebiger (gutartiger) Funktionen implementieren.

Vorgehen: Ersetze den Schätzwert y jeweils durch den verbesserten Schätzwert $y - \frac{f(y)}{Df(y)}$.

```
newton_nst_verbessern y f =  y -  (f y) / (ableitung f  0.001 y)

newton_nst f y =
if (abs (f y)) < 0.0001
then y
else newton_nst f (newton_nst_verbessern y f)

>  newton_nst (\x -> x^2-x) 4
1.0
```