

Edmund Weitz

Mathematische Methoden der Informatik

(Skript für den Studiengang *Medieninformatik* an der *HAW Hamburg*)

17.10.2026, 18:14

Dieses Skript ist momentan noch „work in progress“ und wird regelmäßig aktualisiert. Sie sollten ab und zu nachschauen, ob es eine neue Version (siehe Datum oben) gibt. Den jeweils letzten Stand finden Sie unter der Adresse <https://weitz.de/files/mmi-skript.pdf>.

Falls Sie Fehler entdecken (auch „kleine“ Tippfehler), freue ich mich über eine E-Mail an edmund.weitz@haw-hamburg.de.

Der gesamte Text ist unter der Creative-Commons-Lizenz [BY-NC-ND 3.0 DE](https://creativecommons.org/licenses/by-nc-nd/3.0/de/) verfügbar. Eine vereinfachte Zusammenfassung dieser Lizenz findet man unter der folgenden Adresse:
<https://creativecommons.org/licenses/by-nc-nd/3.0/de/>

Inhaltsverzeichnis

Vorwort	1
I Rekursion	3
1 LISP und SCHEME	3
2 Zahlen, Symbole, Listen	4
3 Evaluation	6
4 S-Ausdrücke	9
5 Funktionen	14
6 Bedingte Anweisungen und Verzweigungen	18
7 Funktionale Programmierung	23
8 Rekursion	26
9 Die natürlichen Zahlen	31
II Der Lambda-Kalkül	39
10 Die Syntax von Lambda-Ausdrücken	39
11 Die intendierte Semantik	43
12 Beta-Reduktion	46
13 Church-Kodierung	52
III Typentheorie	63
14 Ausdrücke, Typen, Konstruktoren	63
15 Urteile und Normalformen	66
16 Funktionen	71
17 Rekursion über natürliche Zahlen	75
18 Parametrisierte und abhängige Typen	80
19 Gleichheit von Funktionen	86
20 Die Curry-Howard-Korrespondenz	89
21 Logik in PIE	99
IV Ausblick: LEAN	109
Lösungen zu ausgewählten Aufgaben	115
Index	161

Vorwort

In diesem Skript werden nacheinander drei Themen behandelt, die aufeinander aufbauen: [Rekursion](#), [der Lambda-Kalkül](#) und [Typentheorie](#), wobei Letztere in gewissem Sinne das Ziel des Semesters bildet, weil sie nach wie vor ein aktiver Gegenstand der Forschung ist und auch im Zusammenhang mit [aktuellen KI-Systemen](#) genutzt wird. Ein Beispiel für ein sowohl in der Mathematik als auch in der Informatik häufig eingesetztes System, das auf Typentheorie basiert, ist der [Beweisassistent LEAN](#), den wir uns ganz zum Schluss anschauen. Wenn Sie dieses Skript durchgearbeitet haben, werden Sie nicht sofort mit LEAN arbeiten können, aber Sie haben hoffentlich verstanden, wozu so ein System überhaupt da ist und wie es funktioniert.

Ein durchgehendes Thema wird *Minimalismus* sein. Wir werden oft nicht den einfachsten und bequemsten Weg verwenden, um ein Problem zu lösen, sondern den, für den wir mit möglichst wenigen, möglichst simplen Werkzeugen auskommen. Das machen wir deshalb, weil es um *Grundlagen* geht.

Einsatz des Computers

Obwohl man argumentieren kann, dass wir uns hier mit Mathematik beschäftigen, werden wir in allen drei Kapiteln mit dem Computer arbeiten. Im zweiten und dritten Kapitel verwenden wir spezielle Sprachen, die dezidiert für Lehrzwecke entwickelt wurden, im ersten Kapitel benutzen wir eine „richtige“ Programmiersprache, beschränken uns dabei jedoch im Sinne des angesprochenen Minimalismus auf eine sehr kleine Teilmenge des Sprachumfangs. Diese Tools bieten die Möglichkeit, neue Konzepte direkt auszuprobieren und sofortiges Feedback zu erhalten. Sie sollten diese Möglichkeit nutzen und so viel wie möglich experimentieren!

Index

Am Ende des Skripts finden Sie einen [Index](#), in dem viele Begriffe von den vorherigen Seiten aufgeführt sind, damit Sie diese nachschlagen können, falls Sie vergessen haben, wo etwas definiert oder zum ersten Mal erwähnt wurde.

Aufgaben

Im Skript finden Sie Aufgaben. Die Bearbeitung der Aufgaben ist freiwillig, aber natürlich sollten Sie sich mit ihnen beschäftigen, damit Sie eine gewisse Routine im Umgang mit den neuen Begriffen und Konzepten entwickeln. Die Aufgaben sind als *Ergänzung* zur Vorlesung gedacht und sollen beim Verständnis helfen. Selbst dann, wenn einzelne Aufgaben bereits besprochen wurden, kann es nicht schaden, sich noch einmal mit ihnen zu beschäftigen, um sicherzustellen, dass Ihnen auch wirklich alles klar ist.

Im [Anhang](#) finden Sie Lösungen für die meisten Aufgaben, aber die sollten Sie sich erst anschauen, wenn Sie selbst lange genug über den Aufgaben gegrübelt haben. (Geben Sie nicht zu früh auf!) Wenn Sie einfach nur die Lösungen lesen, lernen Sie nichts. Aber nachdem Sie selbst nachgedacht haben, sollten Sie sich die Lösungen schon anschauen. Vielleicht finden Sie dort eine andere Herangehensweise oder weitere Hinweise.

Bonusmaterial

Bei Abschnitten und Aufgaben, die mit einem blauen Stern (★) gekennzeichnet sind, handelt es sich um optionales Material, das man nicht unbedingt braucht, um den Rest des Skripts zu verstehen. Wenn Sie große Angst haben, aus Versehen etwas zu lernen, das Sie für die Prüfung nicht brauchen, dann können Sie es überspringen.¹ Aber natürlich stehen diese Sachen nicht ohne Grund im Text. Sie werden auf jeden Fall nicht dümmer, wenn Sie sich damit beschäftigen, und Ihnen wird kein bleibender Schaden entstehen.

¹Allerdings sollten Sie sich dann fragen, warum Sie überhaupt studieren ...



Rekursion

Das wesentliche Thema dieses Kapitels ist, wie man dem Titel entnehmen kann, **Rekursion**. Das dient der Vorbereitung auf die folgenden Kapitel, in denen dieses Konzept ebenfalls eine wichtige Rolle spielt. Rekursion ist aber auch unabhängig davon eine elegante und flexible Programmiertechnik, die man beherrschen sollte.

Gewissermaßen nebenbei wird es im Folgenden auch um **funktionale Programmierung** gehen. Und Sie bekommen Einblicke in eine Programmiersprache, die Sie wahrscheinlich bisher nicht kannten – aber kennen sollten, weil es immer von Vorteil ist, wenn man in der Informatik „mehrsprachig“ ist und nicht nur eine Art und Weise kennt, eine Aufgabe zu erledigen.

1. LISP und SCHEME

Wir werden in diesem Kapitel die Programmiersprache **SCHEME** verwenden, die zur „LISP-Familie“ gehört.

LISP (für *List Processor*) ist die zweitälteste **höhere Programmiersprache**, die heutzutage noch regelmäßig verwendet wird.¹ Sie entstand 1958 am **MIT**.² Im Gegensatz zu den meisten vergleichbaren Sprachen gibt es nicht nur *eine* aktuelle Version von LISP, sondern eine ganze Familie von mehr oder weniger direkten Nachfahren der „Ursprache“, die sich teilweise substantiell unterscheiden. LISP-Varianten werden z. B. in **EMACS**, **GIMP** oder **AUTOCAD** verwendet. Der LISP-Dialekt, der für größere Projekte in der Industrie eingesetzt wird, heißt **COMMON LISP**. Vergleichsweise moderne LISP-Sprachen sind **CLOJURE** (2007) und (im weiteren Sinne) **JULIA** (2012).

SCHEME ist eine LISP-Variante, die Anfang der 1970er Jahre ebenfalls am MIT entwickelt wurde. Ursprünglich im Rahmen eines Forschungsprojektes und mit einer ganz anderen Zielsetzung entstanden, hat sich **SCHEME** als sehr einflussreich für

¹Die älteste ist **FORTRAN**.

²Zum Vergleich: **PYTHON**, **JAVA** und **JAVASCRIPT** entstanden in den 1990er Jahren und sind damit über 30 Jahre „jünger“.

die Entwicklung anderer Sprachen erwiesen und ist inzwischen eine „ausgewachsene“ Hochsprache. Gleichzeitig hat die Sprache sich einen gewissen Minimalismus bewahrt, wodurch sie sich sehr gut für die Lehre eignet.

Wir verwenden den SCHEME-Dialekt **RACKET**, weil er mit einer komfortablen **Entwicklungsumgebung** namens **DRRACKET** ausgeliefert wird, die sich auch für spätere Kapitel eignen wird.³ Für das, was wir in diesem Kapitel machen, würde sich aber auch jede andere SCHEME-Variante eignen.

Achtung: Der Sinn der folgenden Abschnitte ist *nicht* das Erlernen von SCHEME! Wir werden uns absichtlich auf eine minimale Teilmenge von SCHEME beschränken.⁴ Wenn z. B. **Strings**, **Vektoren** oder **Fließkommazahlen** nicht erwähnt werden, dann heißt das nicht, dass SCHEME so etwas nicht hat, sondern nur, dass wir so tun werden, als gäbe es das nicht. Sollten Sie sich mit SCHEME bereits auskennen, so verwenden Sie bitte trotzdem nur die Elemente der Sprache, die auf den folgenden Seiten vorgestellt werden. (Zusätzlich sei darauf hingewiesen, dass RACKET in einigen Punkten bewusst von **SCHEME-Standard** abweicht. Siehe z. B. Aufgabe 8.6.)

2. Zahlen, Symbole, Listen

In **DRRACKET** arbeiten wir zunächst im (unteren) Interaktionsfenster.⁵ In der LISP-Welt spricht man auch von einem *Listener* oder einer *REPL* (für *read-eval-print-loop*): Man gibt etwas ein und erhält sofort eine Antwort.

Eine (ganze) **Zahl** (*integer*) ist eine Folge von Dezimalziffern, vor der optional eines der Zeichen + oder – steht.

Gibt man in der REPL so etwas wie 42 oder –1000 ein, so wird als Antwort einfach diese Zahl wieder ausgegeben.

Ein **Symbol** ist eine Folge von Zeichen.

Die genauen Regeln dafür, welche Zeichenfolgen zugelassen sind, will ich hier nicht wiedergeben, weil das unnötiger Ballast wäre. Grob gesagt ist SCHEME in dieser Hinsicht sehr liberal, so dass die folgenden Zeichenketten beispielsweise alle Symbole sind:

```
hamburg  
r2d2  
wurst-salat
```

³Man kann nämlich in LISP sehr gut **domänenspezifische Sprachen** erstellen und RACKET stellt dafür eine komplette Infrastruktur zur Verfügung.

⁴Und teilweise werde ich bei Erklärungen auch bewusst vereinfachen oder Dinge sagen, die formal nicht ganz korrekt sind. Es gibt **viele gute Bücher über SCHEME**. Dieses Skript ist keines von denen.

⁵Im oberen *Definitions Fenster* sollte in der ersten Zeile `#lang racket` stehen. Das geschieht normalerweise automatisch.

```
x=y  
...  
+
```

Als Faustregel kann man sich merken, dass keine Symbole zugelassen sind, die man mit Zahlen⁶ verwechseln könnte (z. B. 42, 1e3, 3/4 oder .4) oder die Leerzeichen oder einen **Hochstrich** enthalten. Das **Rautezeichen** und gerade **Anführungszeichen** sollte man ebenfalls vermeiden. Beschränkt man sich auf Buchstaben und **Bindestriche** ist man auf der sicheren Seite.⁷

Aus Gründen, die gleich noch klar werden, muss man in der REPL Symbolen einen Hochstrich voranstellen. (Der Hochstrich ist *nicht* Teil des Symbols.) Geben Sie z. B. 'HAW ein, so erhalten Sie auch 'HAW wieder zurück; das ist das Symbol HAW.

Zahlen und Symbole werden wir auch als **Atome** bezeichnen. In „unserer“ (eingeschränkten) Version von SCHEME sind das die einzigen Atome.

Wie der Name LISP bereits andeutet, sind *Listen* die wesentliche Datenstruktur. Sie können eingegeben werden als Folge von Objekten, die durch Leerzeichen (oder Zeilenumbrüche) getrennt sind. Diese Objekte nennen wir die *Elemente* der Liste. Anfang und Ende einer Liste werden durch Klammern markiert. Auch hier beginnen wir vorerst mit einem Hochstrich. Ein paar Beispiele, die Sie ausprobieren sollten:

```
'(23 42)  
'(1 2 drei)  
'()  
'(1 2 (23 42) 5)  
'( 1    2 (23 42)  
  5 )
```

Das erste Beispiel ist eine Liste, die aus zwei Zahlen besteht. Das zweite Beispiel ist eine Liste mit drei Objekten – zwei Zahlen gefolgt von einem Symbol. Beachten Sie, dass vor dem Symbol *drei* *kein* Hochstrich steht. Der Hochstrich vorne bezieht sich auf die gesamte Liste. Das dritte Beispiel zeigt die leere Liste. Das vierte Beispiel demonstriert, dass die Elemente von Listen auch Listen sein können. Das dritte Element dieser Liste ist die Liste (23 42), die anderen Elemente sind Zahlen. Nebenbei sieht man am zweiten und vierten Beispiel, dass die Elemente einer Liste – anders als z. B. die eines **Arrays** in JAVA oder C – nicht alle denselben **Datentyp** haben müssen. Das fünfte Beispiel zeigt, dass **Leerraum** keine wesentliche Rolle spielt. Es hat exakt denselben Effekt wie das vierte Beispiel.

Aufgabe 2.1. Wie viele Elemente hat die Liste (1 2 (23 42) 5) aus dem obigen Beispiel? Geben Sie die Elemente an.

⁶Damit sind an dieser Stelle auch Fließkommazahlen und Brüche gemeint.

⁷Die volle Wahrheit ist, dass eigentlich *jede* Zeichenkette als Symbol zulässig ist. Es ist bei manchen allerdings schwieriger, sie als Symbol einzugeben. Die Details erspare ich Ihnen hier.

Aufgabe 2.2. Wie viele Elemente hat die Liste (1 2 1 3 2)?

3. Evaluation

Nun klären wir, warum der Hochstrich notwendig war. Geben Sie das Folgende
+ ohne Hochstrich ein:

```
(+ 40 2)
```

Sie werden die Antwort 42 erhalten. Offensichtlich haben wir eine Funktion mit dem Namen + aufgerufen, die zwei Zahlen addieren kann.

Die Funktionsweise von SCHEME sieht vereinfacht folgendermaßen aus:

- (E1) Wenn im Folgenden von einem **Ausdruck** die Rede ist, ist damit immer eine Zeichenkette gemeint, die **syntaktisch** korrekt ein Objekt der Sprache angibt.⁸
- (E2) Jeder Ausdruck, der eingegeben wird, wird **evaluiert**. Damit ist gemeint, dass der Ausdruck in einen **Wert** (*value*) umgewandelt wird.
- (E3) Zahlen sind **selbst-evaluierend**, d. h., bei der Eingabe einer Zahl kommt – wie wir bereits gesehen haben – die Zahl selbst wieder heraus. Selbst-evaluierende Objekte werden auch **Literale** genannt.
- (E4) Andere Zeichenketten werden als **Namen** interpretiert.⁹ Gibt es ein Objekt mit diesem Namen, so ist das der zurückgegebene Wert. Anderenfalls gibt es eine Fehlermeldung.
- (E5) Ist der zu evaluierende Ausdruck eine Liste, so wird zunächst das erste Element der Liste evaluiert. Der Wert muss eine **Funktion** (in SCHEME *procedure* genannt) sein. Die weiteren Elemente der Liste werden ebenfalls evaluiert und anschließend als Argumente der Funktion übergeben.

Punkt (E4) lässt sich dadurch illustrieren, dass Sie in der REPL einfach + eingeben. Sie erhalten als Antwort so etwas wie das hier:¹⁰

```
#<procedure: +>
```

+ ist der Name der Funktion, die für uns gerade zwei Zahlen addiert hat. Geben wir den Namen ein, so erhalten wir diese Funktion zurück. (Was man damit anstellen kann, werden wir **noch sehen**.) Entscheidend ist, dass es hier nicht um das *Symbol* + ging. Das hätten wir mit ' + bekommen.¹¹

⁸Was *nicht* syntaktisch korrekt ist, lässt sich einfach erkennen: Das System meckert dann über einen **Syntaxfehler**. Was korrekt ist, lernen Sie auf den folgenden Seiten.

⁹Wir ignorieren wie gesagt andere Datentypen und gehen davon aus, dass diese nicht aus Versehen eingegeben werden.

¹⁰In SCHEME nennt man Funktionen traditionell *Prozeduren*. Im Folgenden werde ich aber in der Regel von Funktionen sprechen. Das ist auch in anderen LISP-Dialekten der übliche Sprachgebrauch.

¹¹In SCHEME sind Namen und Symbole getrennte Konzepte. In anderen Lisp-Dialekten wie z. B. COMMON LISP gibt es einen Zusammenhang: Dort ist beispielsweise das Symbol + der Name der Funktion +.

Geben Sie stattdessen HAW (ohne Hochstrich) ein, so erhalten Sie eine Fehlermeldung, weil es (noch) kein Objekt mit diesem Namen gibt.

Neben + gibt es weitere **arithmetische** Funktionen wie -, * und expt für Subtraktion, Multiplikation und Potenz. Probieren Sie diese Eingaben aus:

```
(- 100 19)
(* 3 27)
(expt 3 4)
```

Es gibt natürlich auch / für die Division, aber darauf werden wir verzichten, weil es sonst ggf. zu Ergebnissen kommen könnte, die keine ganzen Zahlen mehr sind. Wir werden stattdessen die Funktionen quotient und modulo verwenden:¹²

```
(quotient 12 3)
(quotient 14 3)
(modulo 12 3)
(modulo 14 3)
```

- * expt

quotient modulo

Aufgabe 3.1. Bis auf expt akzeptieren die arithmetischen Funktionen auch mehr als zwei Argumente. Es ist wohl klar, was bei (+ 10 5 2) und (* 10 5 2) herauskommen sollte. Aber was würden Sie bei (- 10 5 2) erwarten? Ist das offensichtlich?

Aufgabe 3.2. Fortsetzung von Aufgabe 3.1: Ein einzelnes Argument ist auch erlaubt. Was erwarten Sie als Ergebnis von (- 23)? Probieren Sie es aus!

Aufgabe 3.3. Fortsetzung von Aufgabe 3.2: Man kann auch (+) und (*) eingeben. Was sollte da jeweils herauskommen? Begründen Sie Ihre Antwort!

Punkt (E5) wird durch dieses Beispiel illustriert:

```
(* 3 (+ 2 4))
```

Die Argumente für die Funktion * sind die Zahl 3 und die Liste (+ 2 4). Diese Argumente werden ebenfalls evaluiert, und zwar, *bevor* sie der Funktion übergeben werden. 3 bleibt einfach 3, aber aus (+ 2 4) wird 6, so dass schließlich $3 \cdot 6 = 18$ berechnet wird.

Aufgabe 3.4. Wenn die Art, wie man in LISP-Sprachen Funktionen aufruft, neu für Sie ist, dann sollten Sie zur Eingewöhnung die folgende „Übersetzung“ vornehmen: Nehmen wir an, Sie hätten in JAVA (oder C oder PYTHON oder einer anderen „konventionellen“ Sprache) Funktionen fOne, fTwo und fThree, die als Argument eine, zwei und drei Zahlen erwarten und jeweils eine Zahl zurückgeben. In Ihrem Programm kommen die folgenden Zeilen vor:

¹²Das sollten Sie aus der Mathematik-Vorlesung kennen. Siehe dazu auch [die entsprechende Liste auf Wikipedia](#), die auf die Unterschiede bei negativen Argumenten eingeht.

```
fOne(fTwo(42, y))
fThree(fOne(12), fOne(fOne(12)), fTwo(42, 23))
fTwo(fThree(1, 2, fTwo(8, fOne(11))), fTwo(x, y))
```

Wie würde das in SCHEME-Syntax aussehen?

Aufgabe 3.5. Programmiersprachen wie JAVA müssen Regeln zur **Operatorrangfolge** wie "Punktrechnung vor Strichrechnung" beherrschen, damit zum Beispiel aus $3+4*5$ das Ergebnis 23 und nicht etwa 35 wird. Warum ist das in SCHEME und anderen LISP-Dialekten nicht nötig?

Bei der Beschreibung der Funktionsweise von SCHEME fehlte noch ein wichtiger Punkt, der hiermit nachgereicht wird.

(E6) Neben Funktionen gibt es auch **Spezialoperatoren**, bei deren Ausführung *nicht* alle Argumente evaluiert werden.¹³

quote Einer dieser Spezialoperatoren heißt `quote`. Er akzeptiert ein Argument und gibt dieses einfach *ohne Evaluation* zurück:

```
(quote wurst)
(quote (+ 40 2))
```

Das erste Beispiel liefert das Symbol `wurst` und versucht nicht, ein Objekt mit dem Namen `wurst` zu finden. Das zweite Beispiel liefert die Liste `(+ 40 2)` und versucht nicht, die Funktion `+` mit den Argumenten 40 und 2 aufzurufen.

Der rätselhafte Hochstrich ist schlicht und einfach eine Abkürzung für `quote`: Statt `(quote <whatever>)` kann man einfach `'<whatever>` (ohne Klammern) schreiben. Die beiden Beispiele von oben hätte man genauso auch als

```
'wurst
'(+ 40 2)
```

eingeben können.

define Ein weiterer Spezialoperator ist `define`. Dieser Operator bekommt zwei Argumente, von denen nur das zweite ausgewertet wird. Das Ergebnis der Auswertung erhält dann als Namen das erste (unausgewertete) Argument.¹⁴ Hier ein Beispiel:

```
(define five (+ 2 3))
```

Danach können Sie in der REPL `five` eingeben und erhalten als Antwort die Zahl 5. Das Argument `(+ 2 3)` wurde also ausgewertet, der Name `five` nicht. Denn sonst hätte es eine Fehlermeldung gegeben.

¹³Die offizielle Terminologie ist in SCHEME eine andere, aber ich hatte ja bereits davor gewarnt, dass es hier manchmal etwas ungenau zugehen wird.

¹⁴`define` kann noch mehr. Aber das ignorieren wir.

Beachten Sie, dass Spezialoperatoren in gewissem Sinne gefährlich sind: Der Aufruf eines solchen Operators ist **syntaktisch** nicht von dem einer Funktion zu unterscheiden, obwohl die Evaluationsregeln andere sind. Man muss also anhand des Namens erkennen, wann man es mit einem Spezialoperator zu tun hat.¹⁵ Zum Glück gibt es nicht so viele.

Aufgabe 3.6. Was muss man eingeben, um der Liste, die aus den drei Symbolen H, A und W besteht, den Namen HAW zu geben?

Aufgabe 3.7. Was müssen Sie machen, damit die Funktion für das Addieren von Zahlen auch unter dem Namen `plus` erreichbar ist?

4. S-Ausdrücke

In LISP gibt es traditionell das Konzept eines **Paares**, das dem des **geordneten Paares** aus der Mathematik entspricht. Die Funktion `cons` (für *construct*) akzeptiert zwei Argumente und gibt als Ergebnis ein Paar zurück, das diese beiden Argumente als erste und zweite Komponente hat. (Ein Paar wird deshalb auch manchmal als *cons-Zelle* oder einfach als *cons* bezeichnet.) Paare werden wie Listen mit zwei Elementen dargestellt, allerdings mit einem Punkt zwischen den beiden Komponenten. Zum Beispiel liefert

`cons`

```
(cons 23 'quark)
```

das Paar `(23 . quark)`, das Sie übrigens auch direkt als

```
'(23 . quark)
```

hätten eingeben können. (Beachten Sie den Hochstrich und die Leerzeichen links und rechts vom Punkt.)

Damit man auf die erste bzw. zweite Komponente eines Paares zugreifen kann, gibt es die Funktionen `car` und `cdr` (ausgesprochen so ähnlich wie „koder“).¹⁶ Im Beispiel

`car` `cdr`

```
(define pair-1 (cons 23 42))
(car pair-1)
(cdr pair-1)
```

¹⁵Speziell in RACKET kann man sich zur Not auch folgendermaßen behelfen: Gibt man den Namen eine Funktion wie `+` ein, so erhält man, wie wir oben gesehen haben, die Funktion als Objekt zurück. Gibt man den Namen eines Spezialoperators wie `define` ein, so antwortet das System mit einer Fehlermeldung.

¹⁶Aus nostalgischen Gründen werden diese Namen seit über 60 Jahren beibehalten, obwohl sie sich ursprünglich auf die spezielle Architektur des Rechners **IBM 704** bezogen, auf dem LISP entwickelt wurde (und übrigens auch FORTRAN). `car` und `cdr` standen damals für „contents of the address/decrement part of register number“.

liefert der zweite Ausdruck als Ergebnis 23 und der dritte 42. Im LISP-Jargon nennt man die Komponenten auch den *car* bzw. den *cdr* des Paares.

Selbstverständlich können neben Zahlen und Symbolen auch andere Objekte Komponenten von Paaren sein, insbesondere Paare selbst. Probieren Sie z. B. das hier aus und überlegen Sie sich jeweils vorher, was herauskommen sollte:¹⁷

```
(define pair-2 (cons
  (cons 3 4)
  (cons 'foo
    (cons 'bar 0))))

(car pair-2)
(cdr (car pair-2))
(car (cdr (cdr pair-2)))
```

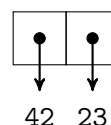
Probieren Sie auch andere Kombinationen von *car* und *cdr* aus. Da man so etwas in LISP häufig macht, gibt es übrigens Abkürzungen. Man hätte die letzten beiden Ausdrücke auch als

```
(cdar pair-2)
(caddr pair-2)
```

schreiben können. Jede Funktion der Form *c?r*, bei der für das Fragezeichen bis zu vier Zeichen (natürlich nur *a* oder *d*) eingesetzt werden, existiert.

Aufgabe 4.1. Bauen Sie sich ähnliche Beispiele wie *pair-2* (oder noch komplexere) und experimentieren Sie mit *car*, *cdr* und *cons*, bis Sie sich sicher im Umgang mit diesen Funktionen fühlen.

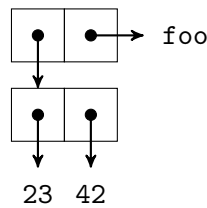
Für das Verständnis kann es hilfreich sein, sich Paare grafisch auf eine Art vorzustellen, die ihre (fiktive) Repräsentation im Computerspeicher darstellt. Das Paar (42 . 23) würde z. B. so aussehen:



Man stelle sich jedes Quadrat als eine Speicherzelle vor und die Pfeile als **Zeiger**.¹⁸ Das Paar ((23 . 42) . foo) sähe so aus:

¹⁷Beachten Sie die Einrückung. Die ist zwar – anders als z. B. in PYTHON – nicht zwingend erforderlich, hilft aber beim Erkennen der Struktur komplexer Ausdrücke ungemein. Ein guter LISP-Editor macht das für Sie automatisch.

¹⁸Dabei ist es irrelevant ob „kleine“ Objekte wie Zahlen evtl. direkt in der Speicherzelle residieren statt indirekt über einen Zeiger referenziert zu werden.



Paare eignen sich als flexible Bausteine für verschiedene Datenstrukturen, insbesondere für die bereits erwähnten Listen. Dafür geht man folgendermaßen vor: Zunächst gibt eine spezielle Liste, die wir bereits gesehen haben, nämlich die *leere* Liste (). Die kann man sich in diesem Zusammenhang als „Zeiger ins Nichts“ vorstellen.¹⁹ Verwendet man die leere Liste als cdr eines Paares, so wird das als Liste mit einem Element interpretiert, wobei dieses Element der car des Paares ist:

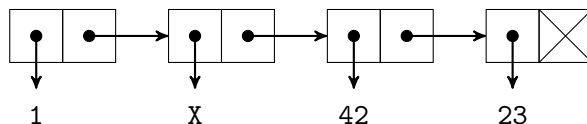
```
(cons 23 '())
```

Wenn Sie das ausprobieren, werden Sie sehen, dass nun eine andere Schreibweise verwendet wird. Statt als (23 . ()) wird das Ergebnis als (23) dargestellt.

Und so kann man an den Anfang einer solchen Liste mittels cons weitere Elemente „anflanschen“:

```
(cons 42 (cons 23 '()))
(cons 'X (cons 42 (cons 23 '())))
(cons 1 (cons 'X (cons 42 (cons 23 '()))))
```

Das Ergebnis des letzten Ausdrucks ist die Liste (1 X 42 23):



Dabei wurde in der grafischen Darstellung die leere Liste als durchkreuztes Quadrat dargestellt. Wie Sie aus der Informatik-Vorlesung wissen dürften, handelt es sich bei dem, was wir hier erzeugt haben, um eine Datenstruktur, die man *einfach verkettete Liste* nennt.

Zum Glück muss man Listen nicht so aufwendig erzeugen. Erstens wissen wir schon, dass wir Listen mithilfe von quote direkt eingeben können, und zweitens gibt es eine Funktion list, die beliebig viele Argumente akzeptiert und diese zu einer Liste zusammenfügt. Die Liste von eben hätte man also auch auf diese beiden Arten zusammenfügen können:²⁰

list

¹⁹In einer Sprache wie C wäre das so etwas wie der Zeiger NULL und in vielen LISP-Dialekten, allerdings nicht in SCHEME, heißt dieses Objekt auch NIL.

²⁰Es gibt einen feinen Unterschied zwischen beiden Varianten, der aber nur für fortgeschrittene SCHEME-Programmierung relevant ist.

```
'(1 X 42 23)
(list 1 'X 42 23)
```

Aufgabe 4.2. Überlegen Sie, was hier jeweils herauskommen sollte. Überprüfen Sie es dann in DRRACKET.

```
(define my-list
  (list 23 'X 42))
(car my-list)
(cadr my-list)
(cons 'Y my-list)
```

Aufgabe 4.3. Mit welcher SCHEME-Funktion kann man auf das dritte Element einer Liste zugreifen?

Aufgabe 4.4. Kann man einfach (list) eingeben? Was sollte herauskommen? Probieren Sie es aus!

Aufgabe 4.5. Stellen Sie die beiden folgenden Listen grafisch dar und überlegen Sie sich, wie die REPL sie ausgeben wird, bevor Sie sich das anzeigen lassen.

```
(list 'X (cons 'A 'B) 42 23)
(list 'X 42 (list 'A 'B) 23)
```

Aufgabe 4.6. Man kann in SCHEME

```
'(1 2 3 . 4)
```

eingeben, ohne dass das System meckert. Wie kann man wohl das gleiche Ergebnis mithilfe von verschachtelten cons-Aufrufen erreichen?

Aufgabe 4.7. Was würden Sie bei der Eingabe

```
'(1 2 . 3 4)
```

erwarten? Probieren Sie es aus!

Wir führen noch zwei Begriffe ein, wobei es sich beim zweiten nur um eine Präzisierung eines bereits verwendeten Wortes handelt:

Atome sind **S-Ausdrücke** und die leere Liste () ist ebenfalls einer. Sind X und Y S-Ausdrücke, dann ist das Paar (cons X Y) ebenfalls einer. () ist eine **Liste**. Ist L eine Liste und X ein S-Ausdruck, dann ist (cons X L) ebenfalls eine Liste. Nur etwas, was durch endliche viele Anwendungen der obigen Regeln entstanden ist, ist ein S-Ausdruck bzw. eine Liste.

Zunächst ein paar Bemerkungen zur historischen Einordnung. Das englische Wort *S-expression* (für *symbolic expression*, oft abgekürzt als *sexp*) wurde bereits im „Ur-LISP“ verwendet. Im Prinzip meint man damit heute immer noch dasselbe.²¹ Allerdings hat sich die Bedeutung des Begriffs *Atom* geändert, weil moderne LISP-Dialekte wesentlich mehr Datentypen als früher haben. Wir tun in diesem Skript so, als gäbe es außer der leeren Liste, ganzen Zahlen und Symbolen keine anderen primitiven Objekte²² und als wären Paare die einzigen zusammengesetzten Objekte. In unserer vereinfachten Variante von SCHEME gibt es also *nur* S-Ausdrücke!

Was eine *Liste* ist, wird hier wie gesagt nur zur Präzisierung definiert. Diese Definition entspricht dem, was man in der LISP-Welt *proper list* nennen würde, passt aber beispielsweise *nicht* zum Datentyp LIST in COMMON LISP.²³

In beiden Fällen handelt es sich übrigens um sogenannte **rekursive Definitionen**.²⁴ Die laufen immer nach dem Prinzip ab, dass man zunächst die einfachsten Fälle definiert und darauf aufbauend dann komplexere, wobei man Letzteres beliebig fortsetzen kann. Die Zahlen 17 und 19 sowie das Symbol FZ sind beispielsweise S-Ausdrücke. Daher ist auch das Paar (17 . FZ) ein S-Ausdruck. Und daraus folgt, dass auch (19 . (17 . FZ)) einer ist. Und so weiter.

Eine Einschränkung im Sinne des letzten Satzes der obigen Definition wird oft nicht explizit ausgesprochen, ist aber immer so gemeint. Das hat zur Konsequenz, dass man immer ein Verfahren angeben kann, mit dem sich überprüfen lässt, ob ein Objekt *X* einer rekursiven Definition entspricht. Der **Algorithmus** für S-Ausdrücke würde z. B. so aussehen:

- (i) Ist *X* die leere Liste? Dann ist *X* ein S-Ausdruck.
- (ii) Ist *X* ein Paar? Dann ist *X* ein S-Ausdruck, *wenn* car und cdr des Paares S-Ausdrücke sind. Und das kann man jeweils mit demselben Algorithmus überprüfen.
- (iii) Anderenfalls ist *X* kein S-Ausdruck.

Wie Sie bereits an einfachen Beispielen gesehen haben, sind die Anweisungen, die wir in der REPL eingeben und die dann von SCHEME evaluiert werden, ebenfalls S-Ausdrücke. Wenn man vorher in einer anderen Programmiersprache programmiert hat, ist das gewöhnungsbedürftig. Man hört oft Klagen über „die vielen Klammern“.²⁵

Das ist jedoch der entscheidende Grund für die Langlebigkeit von LISP, der diese Sprachfamilie gleichzeitig von allen anderen Programmiersprachen unterscheidet:

²¹Übrigens wird WEBASSEMBLY auch durch S-Ausdrücke dargestellt.

²²Der Vollständigkeit halber sollten wir eigentlich noch **Funktionen** zu den Atomen hinzufügen, aber für die späteren Beispiele wird das keine Rolle spielen.

²³(listp '(3 . 4)) führt in COMMON LISP zur Antwort T (für *true*). In SCHEME hingegen liefert (listp '(3 . 4)) die Antwort #f (für *false*).

²⁴So etwas kommt z. B. auch im Modul *Mathematik 1* bei der Definition von Formeln der **Aussagenlogik** oder im Modul *Theoretische Informatik* bei der Definition **regulärer Ausdrücke** vor.

²⁵Obwohl sich komischerweise kaum jemand darüber beschwert, wie viele Semikola und geschweifte Klammern man in JAVA oder C tippen muss.

LISP-Programme sind S-Ausdrücke und liegen damit als Datenstruktur vor, für die die Sprache optimiert ist. Während in typischen Programmiersprachen der **Compiler** oder **Interpreter** den **Quellcode** in einen **Syntaxbaum** übersetzt, auf den das Programm dann keinen Zugriff mehr hat, *ist* ein LISP-Programm im Prinzip bereits ein Syntaxbaum. Diese Eigenschaft – dass die Programme selbst Datenstrukturen der Sprache sind – nennt man *Homoikonizität* und in vollem Umfang haben sie nur LISP-Dialekte. Homoikonizität ist eine wichtige Voraussetzung für **Metaprogrammierung**.²⁶

5. Funktionen

Wie haben schon ein paar Funktionen wie `+`, `car` oder `list` kennengelernt. Ein Unterschied zu vielen „typischen“ Programmiersprachen, den wir bereits am Rande beobachten konnten, wollen wir noch etwas vertiefen: In SCHEME sind Funktionen sogenannte *First-Class-Objekte*, was bedeutet, dass man sie genauso wie Zahlen oder Symbole behandeln kann: Sie können Argumente oder Rückgabewerte anderer Funktionen sein oder in **Datenstrukturen** wie Listen und Paaren gespeichert werden. Probieren Sie das hier aus:

```
(define my-list (list 1 + * 4))
(second my-list)
((third my-list) 6 7)
```

Insbesondere das, was in der letzten Zeile geschieht, wäre in vielen anderen Sprachen unmöglich.²⁷ Aber in SCHEME kann an der Stelle, an der eine Funktion erwartet wird, nach (E5) irgendein Ausdruck stehen, solange sein *Wert* eine Funktion ist.

In einer sinnvollen Programmiersprache sollte man jedoch auch eigene Funktionen definieren können. In SCHEME macht man das mit dem Spezialoperator `lambda`.²⁸ Hier das erste Beispiel

```
(lambda (x) (+ x 1))
```

Das ist eine (zugegebenermaßen noch nicht besonders interessante) Funktion, die die Zahl eins zu einer anderen Zahl addiert. Hinter `lambda` folgt in Klammern der Name des *Arguments*. Der muss nicht `x` lauten, sondern man kann irgendeine Zeichenkette verwenden, die sich auch als Symbol eignen würde. Und dann folgt als *Rumpf* (*body*) der Funktion ein Ausdruck, in dem typischerweise dieser Name vorkommt, und der beschreibt, was die Funktion macht.

²⁶Ein Beispiel dazu finden Sie in der Lösung von Aufgabe 7.2.

²⁷Selbst in manchen anderen LISP-Dialekten. Beispielsweise hätte man in COMMON LISP `my-list` durch `(list 1 #' + #' * 4)` definiert. Die letzte Zeile hätte `(funcall (second my-list) 6 7)` lauten müssen.

²⁸Warum der so heißt, erfahren Sie im [nächsten Kapitel](#). Man kann übrigens statt der Zeichenkette `lambda` in DRACKET auch [das Unicode-Symbol λ](#) verwenden und das auch [direkt eingeben](#).

Wenn Sie den obigen Ausdruck im Interaktionsfenster eingeben, erhalten Sie als Antwort `#<procedure>`: Damit wird angezeigt, dass Sie eine Funktion erzeugt haben. Und zwar eine sogenannte *anonyme Funktion*, weil sie keinen Namen hat. Trotzdem kann man mit solchen Objekten bereits etwas anfangen. Hier zwei Beispiele:

```
((lambda (x) (+ x 1)) 41)
(define some-pair (cons 'wurst (lambda (x) (+ x 1))))
((cdr some-pair) 22)
```

Aber typischerweise will man Funktionen Namen geben. Wir wissen bereits, wie das geht: Mit `define` kann man Objekten wie Zahlen oder Symbolen Namen geben und in SCHEME sind wie gesagt Funktionen ganz normale Objekte.²⁹

```
(define add1
  (lambda (x) (+ x 1)))
(add1 99)
```

Übrigens würde man solche Definitionen typischerweise im Definitionsfenster oben eingeben – darum heißt es so. Drückt man anschließend den Button *Run* (oder alternativ die Tastenkombination `Ctrl-r`), so steht danach der definierte Name im Interaktionsfenster zur Verfügung.

Aufgabe 5.1. Definieren Sie eine Funktion `square`, die eine Zahl quadriert, und probieren Sie sie aus.

Aufgabe 5.2. Definieren Sie eine Funktion, die die [gaußsche Summenformel](#) implementiert. Das Argument soll also eine Zahl³⁰ n und der Ausgabewert die Summe der Zahlen von 1 bis n sein.

Wer schon einmal programmiert hat, sollte durch die bisherigen Beispiele im Prinzip verstanden haben, wie Funktionen in SCHEME arbeiten. Wir wollen an dieser Stelle aber doch noch einmal genauer darauf eingehen, was „unter der Haube“ geschieht, weil das eines der Themen der Vorlesung ist. Wir machen das anhand des `lambda`-Ausdrucks, den wir `add1` genannt haben.

- Ein Aufruf wie `(add1 (* 2 5))` ist zunächst einmal nur eine Abkürzung für `((lambda (x) (+ x 1)) (* 2 5))`. Das haben wir durch die Verwendung von `define` erreicht.
- Wir wissen bereits, dass bei einem Aufruf wie `(add1 (* 2 5))` zuerst der Ausdruck `(* 2 5)` ausgewertet wird. Denselben Effekt hätte man also in diesem Fall mit `(add1 10)` erzielt.
- Nun wird *temporär* dem Namen `x` der Wert 10 zugeordnet.
- Dann wird der Rumpf `(+ x 1)` der Funktion ausgewertet. Im Prinzip passiert also das, was auch nach der Eingabe von

²⁹Man darf abkürzend `(define (add1 x) (+ x 1))` schreiben, aber wir werden das aus didaktischen Gründen absichtlich nicht machen.

³⁰Es reicht, wenn die Funktion für [natürliche Zahlen](#) funktioniert.

```
(define x (* 2 5))
(+ x 1)
```

geschehen würde. Der entscheidende Unterschied ist allerdings, dass dieser Zusammenhang zwischen dem Namen `x` und dem Wert 10 nach der Ausführung der Funktion wieder „vergessen“ wird.

- Insbesondere „verdeckt“ die temporäre Zuordnung von 10 zu `x` – die man **Variablenbindung** nennt – den „eigentlichen“ Wert von `x`, falls es einen gab. Überlegen Sie sich erst, was im Folgenden herauskommen muss, und probieren Sie es dann aus:

```
(define x 41)
(+ x 1)
((lambda (x) (+ x 1)) 99)
(+ x 1)
```

Durch diese Erklärung sollte hoffentlich klar geworden sein, dass die beiden folgenden Funktionen aus der Sicht von SCHEME *identisch* sind.

```
(lambda (x) (+ x 1))
(lambda (y) (+ y 1))
```

Sie sehen zwar im [Quelltext](#) unterschiedlich aus, sind aber als Funktionsobjekte nicht unterscheidbar.

Aufgabe 5.3. Überlegen Sie sich, was hier herauskommen muss. Probieren Sie es aus und erklären Sie dann im Detail, was bei der Auswertung dieses Ausdrucks passiert.

```
((lambda (x) (+ x 1))
 ((lambda (x) (+ x 2)) 3))
```

Aufgabe 5.4. Sie möchten eine Funktion schreiben, die um eine Liste ein weiteres Klammerpaar „wickelt“. Aus `(1 2 3)` soll also beispielsweise die einelementige Liste `((1 2 3))` werden. Ihr Versuch sieht so aus:

```
(define strange
  (lambda (list)
    (list list)))
```

Warum klappt das nicht? Wo ist der Denkfehler?

Aufgabe 5.5. In Aufgabe 5.4 bekommen Sie Fehlermeldungen, wenn Sie beispielsweise `(strange 3)` oder `(strange ' (1 3))` eingeben. Fallen Ihnen Argumente für `strange` ein, die *nicht* zu einer Fehlermeldung führen?

Die Sache mit der Bindung ist allerdings noch ein bisschen interessanter. Das soll das folgende Beispiel verdeutlichen. Probieren Sie es am besten zuerst aus

und versuchen Sie dann, sich selbst eine Erklärung zurechtzulegen, bevor Sie weiterlesen.

```
(define create-adder
  (lambda (n)
    (lambda (x) (+ x n))))

(define add3 (create-adder 3))
(define add5 (create-adder 5))

(add3 5)
(add5 3)
```

Die Funktion `create-adder` gibt als Wert eine Funktion zurück. Wird sie als `(add3 5)` aufgerufen, so ist während der Ausführung die Variable `n` an den Wert 3 gebunden. In der erzeugten Funktion kommt `n` vor und diese Funktion „merkt“ sich gewissermaßen die Bindung, die zum Zeitpunkt ihrer „Geburt“ bestand. Diesen Mechanismus nennt man **Funktionsabschluss** (*closure*). Unter den Namen `add3` und `add5` sind also zwei *verschiedene* Funktionen gespeichert. Sie beruhen zwar beide auf demselben Code `(+ x n)`, haben aber unterschiedliche Bindungen für die Variable `n`.

Man kann mit `lambda` noch mehr als das machen, was bisher gezeigt wurde. Ich möchte an dieser Stelle aber nur noch auf eine Sache eingehen: Es ist möglich, mehr als ein Argument anzugeben. Das sieht dann z. B. so aus:

```
(define sum-from-to
  (lambda (x y)
    (- (gauss y)
       (gauss (- x 1)))))

(sum-from-to 3 6)
```

Diese Funktion berechnet mithilfe der Funktion `gauss` aus Aufgabe 5.2 die Summe der Zahlen von x bis y , wobei sie sich darauf verlässt, dass $1 \leq x \leq y$ gilt. `(sum-from-to 3 6)` berechnet also z. B. $3 + 4 + 5 + 6 = 18$.

Dass das geht, wird Sie nicht wundern, wenn Sie überhaupt schon einmal programmiert haben. Interessanter ist jedoch, dass Funktionsabschlüsse uns erlauben, auf so etwas im Prinzip zu verzichten. Wenn wir mit einer etwas umständlichen Schreibweise leben können, dann können wir Funktionen mit mehreren Argumenten durch solche mit nur einem Argument simulieren:

```
(define sum-from-to-2
  (lambda (x)
    (lambda (y)
      (- (gauss y)
         (gauss (- x 1))))))
```

```
(sum-from-to-2 3)
((sum-from-to-2 3) 6)
```

Unter diesem Blickwinkel ist eine Funktion, die mehr als ein Argument akzeptiert, nur "syntaktischer Zucker". Derselben Idee, die man auch *Currying* nennt, werden wir in den Kapiteln II und III noch öfter begegnen.

Aufgabe 5.6. Schreiben Sie eine Funktion, die aus einer Zahl n eine Liste macht, die aus drei Elementen besteht: dem Vorgänger $n - 1$, der Zahl n selbst und dem Nachfolger $n + 1$. Das Argument 8 soll also beispielsweise zum Rückgabewert $(7\ 8\ 9)$ führen.

Aufgabe 5.7. Schreiben Sie eine Funktion, die Funktionen wie die in Aufgabe 5.6 erzeugt, bei der man jedoch die „Schrittweite“ wählen kann. Während bei $(7\ 8\ 9)$ die Schrittweite 1 ist, ist z. B. bei $(5\ 8\ 11)$ die Schrittweite 3.

Aufgabe 5.8. Wie Sie sich hoffentlich noch erinnern, haben *Funktionen* in der Mathematik immer nur ein Argument. „Mehrstellige“ Funktionen werden dort mithilfe von *Tupeln* realisiert: So etwas wie $f(a, b)$ ist lediglich eine Kurzschreibweise für $f((a, b))$. Schreiben Sie `sum-from-to` in diesem Sinne so um, dass die Funktion nur ein Argument erhält, das ein Paar (also eine `cons`-Zelle) ist.

Aufgabe 5.9. Wenn Konzepte wie Funktionen als *First-Class-Objekte* und *Funktionsabschlüsse* neu für Sie sind, dann war dieser Abschnitt eventuell etwas schwer zu verdauen. Lesen Sie ihn daher gegebenenfalls noch einmal sorgfältig durch und fragen Sie sich, ob Sie alles verstanden haben und es jemandem erklären könnten!

Da wir nach und nach weitere Funktionen und Spezialoperatoren kennenlernen werden, wiederhole ich noch einmal einen guten Rat vom Anfang:

Verwenden Sie bitte in den Programmen auf den folgenden Seiten – insbesondere in den Aufgaben – nur Dinge, die bis dahin im Skript bereits eingeführt wurden. Sollten Sie `SCHEME` bereits kennen, werden Ihnen manche Dinge unnötig umständlich vorkommen. Ich weiß, dass es auch einfacher geht, aber es gibt *didaktische* Gründe, es so wie hier zu machen.

6. Bedingte Anweisungen und Verzweigungen

Was Sie vielleicht schon vermisst haben, sind *bedingte Anweisungen und Verzweigungen* – also die Möglichkeit, Teile einer Funktion nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist. Dafür benötigen wir noch einen Datentyp für *Wahrheitswerte*.³¹ Der heißt in `SCHEME` – wie in vielen Programmiersprachen – *boolean*. Es gibt nur zwei Objekte dieses Typ, nämlich `#t` und `#f`. Das steht natürlich für *true* und *false*, also wahr und falsch.³²

³¹Für diesen Abschnitt ist es hilfreich, wenn Sie sich an das erinnern, was Sie in der Mathematik-Vorlesung über *Aussagenlogik* gelernt haben.

³²Und man darf alternativ auch `#true` bzw. `#false` schreiben.

Im Folgenden sehen Sie ein paar Funktionen, die Wahrheitswerte zurückgeben – sogenannte *Prädikate*:³³

```
(= 3 (- 5 2))
(< 12 (- 13 1))
(eqv? 'wurst 'quark)
(eqv? 4 'vier)
(eqv? (* 2 2) (+ 2 2))
(symbol? 'wurst)
(integer? 'wurst)
(null? 'wurst)
```

= und < sind zwei Beispiele für Funktionen, die Zahlen als Argumente erwarten. Es wird getestet, ob die beiden Argumente gleich sind bzw. ob das erste Argument kleiner als das zweite ist. Es gibt auch noch >, <= und >=, bei denen wohl klar sein dürfte, was ihr Job ist.

eqv? (für *equivalent*) testet ebenfalls Gleichheit, akzeptiert aber nicht nur Zahlen. Diese Funktion ist für unsere Zwecke besonders gut geeignet, weil wir sie für die Objekte verwenden können, die wir *Atome* genannt haben.³⁴ Die drei Funktionen am Ende testen, ob ihr Argument ein Symbol bzw. eine ganze Zahl bzw. die leere Liste ist.

Aufgabe 6.1. Schreiben Sie eine Funktion `zero?`, die testet, ob ihr Argument die Zahl null ist.³⁵

Aufgabe 6.2. Schreiben Sie ein Prädikat `even?`, das testet, ob eine Zahl *gerade* ist.

Aufgabe 6.3. Funktionen wie = und < erlauben mehr als zwei Argumente. Welches Verhalten würden Sie in diesem Fall erwarten? Probieren Sie aus, ob Ihre Vermutung richtig ist.

Die typischen *Junktoren* gibt es natürlich auch. Was in JAVA `&&`, `||` und `!` heißt, heißt in SCHEME `and`, `or` und `not`:

```
(and (< 3 7) (= 3 7))
(or (< 3 7) (= 3 7))
(not (= 3 7))
```

Aufgabe 6.4. Schreiben Sie mithilfe von `not` und Aufgabe 6.2 ein Prädikat `odd?`, das testet, ob sein Argument ungerade ist.

³³Man erkennt auch die nützliche Konvention, dass die Namen von Prädikaten mit einem Fragezeichen enden, wenn sie nicht nur aus mathematischen Symbolen bestehen.

³⁴Im Prinzip hätten wir auch einen Buchstaben sparen und `eq?` verwenden können. Bevor Sie auf die Idee kommen, `eqv?` oder `eq?` für andere Objekte als Symbole und Zahlen zu verwenden, sollten Sie jedoch lieber die *SCHEME-Dokumentation* studieren! Siehe z. B. auch Aufgabe 9.17 und die Lösung von Aufgabe 8.6.

³⁵SCHEME hat bereits eine Funktion mit diesem Namen. Es ist allerdings nicht verboten, diese durch Ihre eigene Schöpfung zu ersetzen. Auf den folgenden Seiten werden wir häufig Funktionen selbst programmieren, die es schon gibt.

Aufgabe 6.5. Schreiben Sie ein Prädikat `atom?`, das testet, ob sein Argument ein Atom im Sinne dieses Skripts ist.

Aufgabe 6.6. Auch `and` und `or` akzeptieren eine beliebige Anzahl von Argumenten. (Bei `not` ergibt das natürlich keinen Sinn.) Es sollte wohl klar sein, welches Verhalten bei mehr als zwei Argumenten zu erwarten ist. Aber was meinen Sie, was bei `(and)` bzw. `(or)` herauskommen sollte? Probieren Sie es aus. Können Sie begründen, warum es so und nicht anders sein muss?

`if` Nun können wir endlich Verzweigungen einführen. Die einfachste Form der Verzweigung ist `if`. Dieser Spezialoperator erwartet drei Argumente. Ist das erste Argument wahr, dann wird das zweite Argument ausgewertet und zurückgegeben, anderenfalls wird das dritte Argument ausgewertet und zurückgegeben. Den Abstand $|a - b|$ zweier Zahlen a und b auf der Zahlengeraden könnte man damit so definieren:

```
(define dist
  (lambda (a b)
    (if (> a b)
        (- a b)
        (- b a))))
```

`abs` **Aufgabe 6.7.** Schreiben Sie eine Funktion, die den Absolutbetrag ihres Argumentes zurückgibt.

`max` **Aufgabe 6.8.** Schreiben Sie eine Funktion, die das Maximum zweier Zahlen berechnet.

`sub1` **Aufgabe 6.9.** Unsere erste Funktion mit einem Namen hieß `add1`. Schreiben Sie eine Funktion `sub1`, die eins von einer Zahl subtrahiert. Das soll sie allerdings nur für positive Zahlen machen. Andere Zahlen sollen unverändert zurückgegeben werden.

Aufgabe 6.10. Ein Schritt in der Collatz-Folge sieht folgendermaßen aus: Ist n gerade, dann ist die nächste Zahl $n/2$; ist n ungerade, dann ist die nächste Zahl $3n + 1$. Schreiben Sie eine Funktion, die die nächste Zahl ausgibt, wenn ihre Eingabe n ist.

Aufgabe 6.11. Schreiben Sie eine Funktion, die als Argumente zwei Zahlen m und n erhält und die als Antwort eines von drei Symbolen ausgibt: `zero`, wenn mindestens eine der beiden Zahlen null ist, `same`, wenn die beiden Zahlen dasselbe Vorzeichen haben, und `different` sonst.

Aufgabe 6.12. Warum ist `if` ein Spezialoperator? Warum ist das sinnvoll?

Es gibt hier einen wichtigen Aspekt, den man anfangs leicht übersehen kann. Um den deutlich zu machen, übersetzen wir die Funktion `dist` mehr oder weniger wortwörtlich in JAVA:

```
public class Test {
    int dist (int a, int b) {
        if (a > b)
```

```

    a - b;
  else
    b - a;
}

```

Wenn Sie versuchen, das zu kompilieren, erhalten Sie die Fehlermeldung, dass $a - b$ und $b - a$ keine Anweisungen (*statements*) seien. Man kann das reparieren, indem man durch das Hinzufügen von `return` aus den beiden Ausdrücken Anweisungen macht:

```

public class Test {
  int dist (int a, int b) {
    if (a > b)
      return a - b;
    else
      return b - a;
  }
}

```

Das ist jedoch **semantisch** nicht identisch mit dem SCHEME-Code, weil `return` die komplette Funktion verlässt und nicht einfach einen Wert „nach außen“ weitergibt. Alternativ kann man den sogenannten **ternären** Operator verwenden:

```

public class Test {
  int dist (int a, int b) {
    return a > b ? a - b : b - a;
  }
}

```

Dass es diese spezielle Konstruktion überhaupt gibt bzw. dass sie in SCHEME nicht benötigt wird, demonstriert den entscheidenden Unterschied. In SCHEME wird ein Ausdruck, der mit `if` anfängt, im Wesentlichen genauso wie einer behandelt, der mit `+` oder `list` anfängt. *Jeder* Ausdruck gibt einen Wert zurück.³⁶

Ein weiteres Beispiel, das diese Eigenschaft besonders deutlich macht, ist das folgende: Nehmen wir an, Sie haben für eine bestimmte Situation zwei verschiedene Funktionen `f-even` und `f-odd` geschrieben, wobei die eine nur gerade Zahlen bearbeitet und die andere nur ungerade. Die „richtige“ der beiden Funktionen soll mit dem Argument `x` aufgerufen und das Ergebnis an die Funktion `g` übergeben werden. Sie könnten

```

(g (if (even? x) (f-even x) (f-odd x)))

```

schreiben, was schon ziemlich kurz und knapp ist. Es geht aber sogar noch etwas prägnanter:

³⁶Wieder eine von diesen Ungenauigkeiten, die nach Ansicht des Autors didaktisch sinnvoll sind. Siehe Aufgabe 6.15.

```
(g ((if (even? x) f-even f-odd) x))
```

In dieser Version wählt der `if`-Ausdruck eine der beiden Funktionen aus und diese wird dann auf das Argument `x` angewendet. So etwas geht natürlich nur, wenn Funktionen [First-Class-Objekte](#) sind.

Aufgabe 6.13. Die Junktoren `and` und `or` könnten theoretisch normale Funktionen sein, sind aber [Spezialoperatoren](#). Fällt Ihnen ein sinnvoller Grund dafür ein? (Tipp: Es liegt daran, dass Sie sich so verhalten wie `&&` und `||` in JAVA.)

Aufgabe 6.14. In SCHEME gilt übrigens die häufig praktische Regel, dass alles außer `#f` wahr ist.³⁷ Insbesondere gibt `or` das erste wahre Argument zurück, wenn es eines gibt, und `and` gibt das letzte Argument zurück, wenn alle Argumente wahr sind (und es mindestens ein Argument gibt).

Was muss hier jeweils herauskommen? Überlegen Sie jeweils erst und geben Sie die Ausdrücke dann zur Probe in DRACKET ein.

```
(or (< 7 6) #f (* 7 6) (= 7 6))
(and (> 7 6) #t (* 7 6))
(not (* 7 6))
```

★**Aufgabe 6.15.** In Fußnote 36 ging es um eine Aussage, die absichtlich vereinfacht formuliert ist. Wenn Sie wissen wollen, wie es wirklich ist, lesen Sie die Lösung dieser Aufgabe.

`if` kennt nur zwei Ausgänge: wahr und falsch. Oft hat man es beim Programmieren jedoch mit drei oder mehr Alternativen zu tun, die man der Reihe nach durchprobiert. Ein Beispiel dafür ist Aufgabe 6.11. Dafür hat SCHEME den Spezialoperator `cond`. Die Lösung der besagten Aufgabe hätte man damit übersichtlicher folgendermaßen schreiben können.

```
(define which
  (lambda (m n)
    (cond ((= (* m n) 0) 'zero)
          ((> (* m n) 0) 'same)
          (else 'different))))
```

Syntax und Semantik von `cond` sind einfach erklärt: Jedes Argument ist eine Liste mit zwei Elementen. Der Reihe nach wird in jeder dieser Listen das erste Element – der *Test* – ausgewertet. Ist es wahr, dann gibt der `cond`-Ausdruck das zweite Element ausgewertet zurück und ignoriert die restlichen Listen. Ist der Test nicht wahr, kommt die nächste Liste dran. Die letzte Liste muss mit `else` anfangen.

³⁷Eine Warnung für die, die schon mit anderen LISP-Dialekten wie z. B. COMMON LISP gearbeitet haben: In SCHEME ist die leere Liste *nicht* dasselbe wie `#f` (und daher wahr) und `nil` ist ein ganz normales Symbol ohne besondere Bedeutung.

Ihr zweites Element wird dann ausgewertet und zurückgegeben, wenn alle Tests vorher `#f` ergaben.³⁸

Aufgabe 6.16. Schreiben Sie mithilfe von `cond` eine Funktion `distinguish`, die erkennen kann, ob ihr Argument eine gerade oder ungerade Zahl, ein Symbol oder die leere Liste ist, und die als Antwort ein entsprechendes Symbol ausgibt. Sie soll sich also beispielsweise so verhalten:

```
> (distinguish 42)
'even
> (distinguish 43)
'odd
> (distinguish 'wurst)
'symbol
> (distinguish (list))
'empty-list
> (distinguish (list 42))
'no-idea
```

In der letzten Zeile sieht man, was die Funktion ausgeben soll, wenn sie es mit einem Argument zu tun hat, das nicht in der obigen Aufzählung vorkam.

7. Funktionale Programmierung

Bevor wir zum [eigentlichen Thema dieses Kapitels](#) kommen, halte ich einen kurzen Einschub über [funktionale Programmierung](#) für angebracht. Einen wesentlichen Aspekt dieses [Paradigmas](#) haben wir bereits kennengelernt: dass nämlich Funktionen [First-Class-Objekte](#) sind. Ein weiteres zentrales Konzept folgt nun.

Um das zu illustrieren, schauen wir uns das folgende JAVA-Beispiel an:

```
public class Test {
    static int counter = 0;
    static int square1 (int n) {
        return n*n;
    }
    static int square2 (int n) {
        counter++;
        return n*n;
    }
    public static void main(String[] args) {
        for (int i = 2; i < 6; i++) {
            System.out.printf("The square root of %d is %d.\n",
                              i, i%2 == 0 ? square1(i) : square2(i));
            System.out.printf("Counter is now %d.\n", counter);
        }
    }
}
```

³⁸`cond` kann noch wesentlich mehr, aber wir werden diesen Operator immer so wie hier beschreiben verwenden.

```
}
}
```

Die **Methoden** `square1` und `square2` sind beide im Prinzip Funktionen, die das Quadrat einer ganzen Zahl berechnen und zurückgeben. `square2` macht jedoch „nebenbei“ noch etwas anderes – bei jedem Aufruf wird ein (aus der Sicht der Funktion externer) Zähler hochgezählt. Das nennt man einen **Seiteneffekt**. Wenn man rein funktional programmiert, dann darf es so etwas nicht geben. Man kann sich das am besten folgendermaßen in vereinfachter Form merken.

Ein Programm ist dann **rein funktional**, wenn alle seine Funktionen die Eigenschaft haben, dass man jederzeit den Funktionsaufruf durch sein Ergebnis ersetzen könnte, ohne dass sich am Programmablauf etwas ändern würde.

Im obigen Beispiel kann man z. B. statt `square1(4)` auch einfach 16 schreiben, ohne dass sich etwas ändert. Man kann aber nicht `square2(3)` durch 9 ersetzen, weil das Auswirkungen auf den Wert von `counter` hätte.

Sprachen aus der LISP-Familie unterstützen traditionell diesen Programmierstil, erzwingen ihn aber nicht. Das gilt auch für JAVASCRIPT und (mit Einschränkungen) für PYTHON, um zwei populäre Sprachen zu nennen. JAVA und C++ erhielten in neueren Versionen entsprechende Erweiterungen, die allerdings nur mit gewissen syntaktischen Verrenkungen genutzt werden können. Die bekannteste rein funktionale Sprache ist HASKELL. Die in der **Einleitung** erwähnte Sprache LEAN ist ebenfalls funktional.

Funktionale Programmierung ist keine Marotte von Menschen, die sich das Leben absichtlich schwer machen wollen. Sie macht es vielmehr wesentlich leichter, Programme zu untersuchen und die **Korrektheit** derselben zu verifizieren. Wenn man ein Spiel oder einen Webshop für Flip-Flops programmiert, ist das vielleicht nicht so wichtig. Aber wenn man z. B. Software schreibt, die ein bemanntes Raumschiff steuern soll, dann möchte man sich schon sicher sein, dass das Programm genau das tut, was es tun soll.

Funktionale Programmierung muss nicht zwangsläufig bedeuten, dass es keine Variablen gibt, die geändert werden dürfen, oder dass jede Funktion nur aus einem Ausdruck bestehen darf. Die folgende JAVA-Methode, die die Summe der natürlichen Zahlen von 1 bis n berechnet, ist z. B. aus der Sicht der funktionalen Programmierung OK:³⁹

```
static int sum (int n) {
    int res = 0;
    for (int i = 1; i <= n; i++)
        res += i;
}
```

³⁹Ansonsten ist sie natürlich nicht OK, siehe Aufgabe 5.2.

```
    return res;
}
```

sum verwendet zwar die Variablen res und i und ändert deren Werte, aber diese Zustandsänderungen bleiben für jemanden, der die Funktion aufruft, unsichtbar, weil sie nur *lokal* wirken. Steht im Code irgendwo `sum(100)`, so könnte man das durch 5050 ersetzen, ohne das Verhalten des Programms zu ändern.⁴⁰

Aber *wenn* eine Funktion wie im Beispiel sum aus mehreren Ausdrücken besteht, die nacheinander abgearbeitet werden, dann kann es sich dabei nicht nur um Aufrufe von „reinen Funktionen“ handeln. Nehmen wir dafür die allererste SCHEME-Funktion aus diesem Skript und fügen etwas hinzu:⁴¹

```
(lambda (x)
  (something x)      ;; neuer Ausdruck
  (+ x 1))
```

Das ist legaler SCHEME-Code, aber die zweite Zeile ist sinnlos, wenn die Funktion something keinen Seiteneffekt hat ...

Der Vollständigkeit halber sei noch gezeigt, wie man dem obigen JAVA-Code in Scheme relativ nahe kommen könnte, obwohl die große Mehrheit der SCHEME-Programmierer das so nicht machen würde:

```
(define sum
  (lambda (n)
    (let ((res 0))
      (do ((i 0 (+ i 1)))
          ((> i n))
        (set! res (+ res i)))
      res)))
```

Erwähnenswert ist hier zunächst einmal der `let`-Ausdruck, mit dem man ein oder mehrere Variablen **deklarieren** und initialisieren kann – im konkreten Fall ist die Variable `res` und ihr Anfangswert ist null. Diese Variablen existieren nur innerhalb des `let`-Ausdrucks und werden danach wieder „vergessen“. ⁴² `do` ist – ähnlich wie `for` in JAVA – dafür da, **Schleifen** zu programmieren.⁴³

Am interessantesten ist der `set!`-Ausdruck, mit dem eine **Zuweisung** realisiert wird. `set!` kann offensichtlich keine Funktion im Sinne der funktionalen Programmierung sein, denn durch den Aufruf `(set! res ...)` wird der außerhalb der Funktion sichtbare Wert der Variablen `res` geändert.⁴⁴

⁴⁰Damit ist selbstverständlich nicht der Zeitaufwand gemeint.

⁴¹Nebenbei sieht man hier, dass in den meisten LISP-Sprachen das **Semikolon** ein **Kommentar-Zeichen** ist.

⁴²Ein **Funktionsabschluss** kann sie jedoch gewissermaßen „mitnehmen“.

⁴³`let` wäre hier gar nicht nötig gewesen, siehe Aufgabe 7.3.

⁴⁴Deswegen auch das Ausrufezeichen im Namen. Eine SCHEME-Konvention für sogenanntes *destruktives* Verhalten.

Aufgabe 7.1. `set!` kann nicht einmal eine *Funktion* sein. Warum?

Aufgabe 7.2. Obwohl so etwas wie `let` für größere SCHEME-Programme ganz praktisch sein kann, könnte man theoretisch darauf auch verzichten. Haben Sie eine Idee, wie man `let` durch `lambda` ersetzen könnte?

★**Aufgabe 7.3.** Schauen Sie im [SCHEME-Standard](#) nach, wie `do` arbeitet, und schreiben Sie die obige Definition von `sum` so um, dass `res` innerhalb von `do` deklariert, initialisiert und zurückgegeben wird. (`let` wird dann nicht mehr benötigt.)

Noch eine Anmerkung zum Schluss: Würden wir im JAVA-Beispiel vom Anfang des Abschnitts die Zeile `counter++;` durch

```
System.out.println("Hello World!");
```

ersetzen, so wäre `square2` nach wie vor nicht funktional, denn es gäbe immer noch einen Unterschied zwischen der Zahl 9 und dem Aufruf `square2(3)`.

Wenn man wirklich rein funktional programmieren will, muss man sich überlegen, wie so etwas wie die Ausgabe eines Ergebnisses auf dem Bildschirm überhaupt realisiert werden kann, weil das auf den ersten Blick wie ein Seiteneffekt aussieht. Schauen Sie bei Interesse vielleicht einmal nach, wie [HASKELL](#) damit umgeht.

Aufgabe 7.4. Für die gerade erwähnte Ausgabe ist die einfachste SCHEME-Funktion `write`. Probieren Sie z.B. `(write 42)` oder `(write 'HAW)` im Interaktionsfenster aus. Erklären Sie dann ausführlich, was der wesentliche Unterschied zwischen den folgenden beiden Funktionen ist.

```
(define foo1
  (lambda (x) (+ x 42)))

(define foo2
  (lambda (x) (write (+ x 42))))
```

Nach dieser kleinen Abschweifung halten wir für die folgenden Abschnitte fest:

Alle SCHEME-Funktionen in diesem Skript sollen rein funktional arbeiten. Sie werden immer durch einen `lambda`-Ausdruck definiert, dessen Rumpf aus nur *einem* Ausdruck besteht. Die in diesem Abschnitt exemplarisch vorgestellten Operatoren `set!` und `do` werden im Folgenden *nicht* verwendet. `let` darf verwendet werden, wenn dadurch verhindert wird, dass derselbe Ausdruck mehrfach evaluiert wird.

8. Rekursion

[Rekursion](#) ist sowohl in der Mathematik als auch in der Informatik ein allgegenwärtiges Thema. Trotzdem haben erfahrungsgemäß viele Studierende damit auch

noch in höheren Semestern Schwierigkeiten. Deshalb werden wir das hier noch einmal sehr ausführlich behandeln.

Wir werden im Folgenden wie schon im letzten Abschnitt Funktionen schreiben, die es in SCHEME bereits gibt. Außerdem werden wir uns keine großen Gedanken darüber machen, ob unsere Lösung möglichst effizient sind.⁴⁵ Das Ziel ist lediglich, Routine im Umgang mit Rekursion zu erwerben.

Wir fangen mit Beispielen zu Listen an, weil die sich aufgrund der Art und Weise, wie sie **definiert wurden**, besonders gut für Rekursion eignen. Hier ist eine rekursive Methode, die Länge einer Liste (also die Anzahl ihrer Elemente) zu berechnen:⁴⁶

length

```
(define length
  (lambda (list)
    (cond ((null? list) 0)
          (else (+ 1 (length (cdr list))))))

(length '(1 2 (23 42) 5))
```

Aufgabe 8.1. Wieso kommt bei dem Beispiel 4 heraus? Sollte das nicht 5 sein?

An diesen paar Zeilen sieht man bereits alle wesentlichen Elemente einer rekursiven Funktion:

- (i) Es gibt eine Fallunterscheidung mit mindestens zwei Fällen.
- (ii) Einer der Fälle muss die *einfachste* Version des Argumentes behandeln. Dieser Fall muss zuerst behandelt werden und die Funktion darf sich dafür *nicht* selbst aufrufen.
- (iii) In anderen Fällen wird das Argument vereinfacht und *dieselbe* Funktion wird auf das vereinfachte Argument angewandt.
- (iv) Der Prozess des Vereinfachens muss dazu führen, dass nach endlich vielen Vereinfachungen garantiert die einfachste Version aus (ii) erreicht wird.

Bei Listen ist der einfachste Fall so gut wie immer die leere Liste. Und nach **Definition** ist jede Liste L , die *nicht* die leere Liste ist, von der Form $(\text{cons } X \ L')$, wobei L' ebenfalls eine Liste ist. L' ist offensichtlich eine *einfachere* Liste als L . Der typische Prozess des Vereinfachens bei Listen ist also der Übergang von L zu $(\text{cdr } L)$. Damit ist auch Bedingung (iv) erfüllt, denn $(\text{cdr } L)$ ist kürzer als L und wenn wir oft genug den cdr einer Liste nehmen, muss irgendwann die leere Liste – der einfachste Fall – herauskommen.

(ii) und (iv) sorgen gemeinsam dafür, dass es nicht zu einer **Endlosschleife** kommt, obwohl die Funktion sich selbst aufruft. Bei jedem Aufruf wird das Argument einfacher und nach endlich vielen Schritten muss der einfachste Fall eingetreten

⁴⁵Für die fortgeschrittenen Leser sei darauf hingewiesen, dass Rekursion nicht notwendig langsamer als **Iteration** sein muss. Insbesondere werden in SCHEME alle **endrekursiven** Funktionen vom Compiler in Iterationen übersetzt.

⁴⁶Man hätte hier statt `cond` auch `if` nehmen können. Ich werde aber in diesem Kapitel im Sinne eines einheitlichen Stils immer `cond` verwenden.

sein. Weil dieser von der Funktion zuerst betrachtet wird und dies nicht zu einem erneuten Selbstaufruf führt, endet der Prozess.

Für das, was man eigentlich vorhat, muss man sich dann natürlich noch überlegen, was jeweils herauskommen soll. Im konkreten Beispiel `length` ist die Länge der einfachsten Liste offensichtlich null. Das ist der erste Fall im `cond`-Ausdruck und entspricht (ii). Haben wir es nicht mit der leeren Liste zu tun, so hat unsere Liste L die Form $(\text{cons } X \ L')$. Damit hat sie ein Element mehr als L' , nämlich X . Kennen wir die Länge n von L' so kennen wir auch die Länge von L : die ist dann $n + 1$.

Hier ein weiteres Beispiel, das berechnet, wie viele Elemente einer Liste (ganze) Zahlen sind.

```
(define num-ints
  (lambda (list)
    (cond ((null? list) 0)
          ((integer? (car list))
           (+ 1 (num-ints (cdr list))))
          (else (num-ints (cdr list)))))

(num-ints '(1 2 drei 4 sieben))
```

Der wesentliche Unterschied hier ist, dass es mehr als zwei Fälle gibt. In zwei davon ruft die Funktion sich selbst auf. Neu ist hier auch, dass wir nicht nur den `cdr`, sondern auch den `car` der Liste benötigen, denn der Typ dieses Elements entscheidet ja darüber, ob wir den „Zähler“ um eins erhöhen müssen.⁴⁷

Aufgabe 8.2. Schreiben Sie `num-ints` so um, dass der `cond`-Ausdruck nur zwei Fälle hat, und versuchen Sie den zweiten Fall möglich elegant und kurz auszudrücken.

Aufgabe 8.3. Schreiben Sie eine rekursive Funktion, die testet, ob alle Elemente einer Liste Atome sind. Die Funktion sollte z. B. folgendermaßen reagieren:

```
> (list-of-atoms? '(1 2 drei 4))
#t
> (list-of-atoms? '(1 (2) drei 4))
#f
> (list-of-atoms? '())
#t
```

Sie dürfen natürlich das Prädikat `atom?` aus Aufgabe 6.5 verwenden.

Aufgabe 8.4. Schreiben Sie eine Funktion, die als Argument eine Liste von Zahlen erwartet und die die Summe dieser Zahlen zurückgibt. (Hinweis: Was bei der leeren Liste herauskommen sollte, wurde bereits in Aufgabe 3.3 besprochen.)

Aufgabe 8.5. Machen Sie dasselbe wie in Aufgabe 8.4 für Produkte.

⁴⁷Der Witz an der Rekursion ist natürlich, dass es gar keinen „Zähler“ gibt. Jedenfalls keinen, der im Quelltext zu sehen ist.

Funktionen, die Listen rekursiv durchlaufen, können auch Listen zurückgeben. Als einfachsten Fall schauen wir uns zunächst eine Funktion an, die eine Liste lediglich kopiert:

```
(define copy
  (lambda (list)
    (cond ((null? list) '())
          (else (cons (car list)
                       (copy (cdr list)))))))

(copy '(42 23 99))
```

★**Aufgabe 8.6.** Haben Sie sich gefragt, warum man `copy` nicht einfach in einer Zeile schreiben kann?

```
(define naive-copy
  (lambda (list) list))
```

Überlegen Sie sich, was wohl der Unterschied sein könnte, und lesen Sie sich dann die Lösung durch.

Nun eine marginal interessantere Version, die in einer Liste von Zahlen jeweils das Vorzeichen ändert. Der Unterschied ist offensichtlich nicht sehr groß.

```
(define invert
  (lambda (list)
    (cond ((null? list) '())
          (else (cons (- (car list))
                       (invert (cdr list)))))))

(invert '(23 0 -42 -1))
```

Aufgabe 8.7. Schreiben Sie `invert` so um, dass Nullen entfernt werden. Mit dem Argument `(23 0 -42 -1)` soll ich also z. B. `(-23 42 1)` ergeben.

Aufgabe 8.8. Schreiben Sie eine Funktion `remove-all`, die zwei Argumente erwartet, von denen das erste ein [Atom](#) und das zweite eine Liste von Atomen ist. Der Aufruf `(remove-all  $x$   $L$ )` soll die Liste zurückgeben, die durch das Entfernen von x aus L entsteht. Die Funktion soll also so funktionieren:

```
> (remove-all 'three '(one two three four five))
'(one two four five)
> (remove-all 'three '(one two four five))
'(one two four five)
> (remove-all 23 '(11 23 42 23 17))
'(11 42 17)
```

Aufgabe 8.9. Ändern Sie den Code aus der Lösung von `remove-all` so um, dass daraus eine Funktion `remove-first` wird, die das zu entfernende Atom nur einmal entfernt.

```
> (remove-first 23 '(11 23 42 23 17))
'(11 42 23 17)
```

Aufgabe 8.10. Schreiben Sie ein rekursives Prädikat `member?`, das testet, ob ein Atom Element einer Liste ist. (Das Prädikat erhält also zwei Argumente: das Atom und die Liste.) Denken Sie dabei an Aufgabe 2.1!

Aufgabe 8.11. Schreiben Sie eine rekursive Funktion `unique`, die eine Liste von Atomen als Argument erhält und die aus dieser Liste alle Dopplungen entfernt.

```
> (unique '(1 2 3 5))
'(1 2 3 5)
> (unique '(1 3 2 1 3 5 5))
'(2 1 3 5)
```

Im Prinzip macht die Funktion also aus der Liste eine **Menge**. (Hinweis: `member?` aus Aufgabe 8.10 könnte hilfreich sein.)

Aufgabe 8.12. Schreiben Sie eine rekursive Funktion `union`, die zwei Listen von Atomen als Argumente erhält und die die **Vereinigung** der beiden ermittelt. Sie können dabei davon ausgehen, dass die beiden Listen „Mengen“ im Sinne von Aufgabe 8.11 sind.

```
> (union '(1 2) '(3 4))
'(3 4 1 2)
> (union '(1 2) '(2 3))
'(3 1 2)
```

Da es hier um Mengen geht, spielt die Reihenfolge keine Rolle.

Aufgabe 8.13. Eine typische Funktion, die im Zusammenhang mit Listen häufig Verwendung findet, heißt (in vielen Programmiersprachen, nicht nur in LISP) `map`. `map` erwartet als Argumente eine Funktion und eine Liste, wendet die Funktion nacheinander auf jedes Element der Liste an und gibt die Liste der Rückgabewerte aus.

```
> (map add1 '(8 9 10))
'(9 10 11)
> (map (lambda (x) (expt 2 x)) '(1 2 3 4))
'(2 4 8 16)
> (map - '(23 0 -42 -1))
'(-23 0 42 1)
```

Schreiben Sie `map`. (Tipp: Die letzte Zeile im obigen Beispiel macht dasselbe wie `invert`. Fangen Sie mit den Code von `invert` an und ändern Sie ihn entsprechend.)

`pair?` Für die letzte Aufgabe benötigen wir noch das Standard-Prädikat `pair?`, das **Paare** erkennen kann.

```

> (pair? 'wurst)
#f
> (pair? '(wurst . quark))
#t
> (pair? '(wurst quark))
#t
> (pair? '())
#f

```

Aufgabe 8.14. Schreiben Sie ein Prädikat `list?`, das testet, ob ein Objekt eine Liste gemäß der [Definition](#) auf Seite 12 ist. Sie können dabei die Konvention verwenden, dass (zumindest in diesem Skript) *alle* Objekte S-Ausdrücke sind.

`list?`

Durch diese Aufgabe sollte hoffentlich auch noch einmal klar geworden, was auf Seite 13 mit dem Begriff *rekursive Definition* war und wie man testen kann, ob ein Objekt so eine Definition erfüllt.

9. Die natürlichen Zahlen

Aus mathematischer Sicht gibt es etwas, das sich noch direkter für Rekursion anbietet als Listen, nämlich die [natürlichen Zahlen](#).⁴⁸ Listen baut man sukzessive auf, indem man mit der leeren Liste anfängt und nach und nach durch `cons` weitere Elemente hinzufügt. Natürliche Zahlen erhält man, indem man mit der Null anfängt, und so oft eins addiert (z. B. mit `add1`), bis man die gewünschte Zahl hat. Für die Rekursion muss man „umgekehrt“ denken: Während man mit `cons` Listen baut, „vereinfacht“ man Liste mit `cdr`; (`cdr L`) ist gewissermaßen der „Vorgänger“ der Liste L . Die Gegenüberstellung zu den natürlichen Zahlen sieht dann so aus:

	Listen	$\mathbb{N}$
einfachster Fall	<code>'()</code>	0
Test dafür	<code>null?</code>	<code>zero?</code>
„Vorgänger“	<code>cdr</code>	<code>sub1</code>

Aufbauend auf diesen einfachen „Bausteinen“ kann man erstaunlicherweise die gesamte [Arithmetik](#) der natürlichen Zahlen aufbauen.⁴⁹ Das wird gleichzeitig eine Vorbereitung für Kapitel III sein. Für diesen Abschnitt setzen wir daher die folgenden Regeln fest.

Wenn in diesem Abschnitt von *Zahlen* die Rede ist, sind immer natürliche Zahlen gemeint. Negative Zahlen kommen also nicht vor.

Bis Aufgabe 9.12 werden die in SCHEME bereits vorhandenen arithmetischen Funktionen (siehe Seite 7) und Vergleiche (siehe Seite 19) *nicht* verwendet.

⁴⁸Wie in der [Mathevorlesung](#) ist auch hier die Null eine natürliche Zahl.

⁴⁹Auf den Grund dafür gehen wir in Abschnitt 13 noch einmal ein.

Zugelassen sind nur `add1`, `sub1` und `zero?` sowie Funktionen, die in diesem Abschnitt bereits definiert wurden.

Zusätzlich ist zu beachten, dass wir die Funktionen `zero?`, `add1` und `sub1` verwenden müssen, die RACKET zur Verfügung stellt. Das Folgende wird *nicht* mit den gleichnamigen Funktionen klappen, die in vorherigen Abschnitten definiert wurden.

Fangen wir mit der Addition an. Um den Ausdruck $a + b$ für natürliche Zahlen sinnvoll zu definieren, werden wir Rekursion über das zweite Argument b verwenden. Der einfachste Fall ist dann $b = 0$, also $a + 0$, und da muss natürlich a herauskommen. Ist b *nicht* null, dann gibt es einen „Vorgänger“ b' , d. h. es gilt $b = b' + 1$. Damit hat man dann

$$a + b = a + (b' + 1) = (a + b') + 1, \quad (9.1)$$

was mit anderen Worten bedeutet, dass $a + b$ der „Nachfolger“ der einfacheren Addition $a + b'$ ist. In SCHEME sieht das so aus:

```
(define +
  (lambda (a b)
    (cond ((zero? b) a)
          (else (add1 (+ a (sub1 b)))))))
```

Hier ist zu beachten, dass mit dem Zeichen `+` in der letzten Zeile der rekursive Aufruf „unserer“ Funktion `+` gemeint ist und nicht etwa die Standardfunktion mit diesem Namen. (Sie können das überprüfen, indem Sie `+` an *beiden* Stellen durch einen anderen Name wie z. B. `my+` ersetzen.)

Aufgabe 9.1. Überprüfen Sie, dass diese Version von `+` für natürliche Zahlen korrekt arbeitet. Testen Sie in solchen Fällen immer auch Spezialfälle wie $0 + 0$. Stellen Sie sicher, dass Sie die Funktionsweise von `+` verstanden haben!

Aufgabe 9.2. Warum würde die obige Definition von `+` mit „unserer“ Version von `add1` nicht funktionieren?

Multiplikation lässt sich natürlich ebenfalls rekursiv definieren. Das Pendant zu (9.1) sieht so aus:

$$a \cdot b = a \cdot (b' + 1) = a \cdot b' + a$$

Dabei ist $a \cdot b'$ der einfachere Term für die Rekursion. Zu dem wird a addiert und Addition haben wir ja zum Glück bereits. Damit sollte klar sein, wie die folgende Aufgabe zu lösen ist.

Aufgabe 9.3. Definieren Sie rekursiv eine Funktion zur Multiplikation zweier natürlicher Zahlen. Sie dürfen die oben definierte Funktion `+` verwenden.

Es bietet sich an, als Nächstes den Vergleich `<` zu definieren. Hier gehen wir zur Klärung der Frage, ob a kleiner als b ist, folgendermaßen vor:

- (i) Ist b null, dann gilt $a < b$ garantiert *nicht*.
- (ii) Ist b nicht null und a null, dann gilt $a < b$ *immer*.
- (iii) Sind a und b beide nicht null, dann sind $a - 1$ und $b - 1$ auch natürliche Zahlen und $a < b$ gilt genau dann, wenn $a - 1 < b - 1$ gilt.

Wir führen also eine Rekursion durch, die parallel beide Argumente vereinfacht.

```
(define <
  (lambda (a b)
    (cond ((zero? b) #f)
          ((zero? a) #t)
          (else (< (sub1 a) (sub1 b))))))
```

Aufgabe 9.4. Ähnlich wie $<$ kann man auch $=$ definieren. Machen Sie das!

Aufgabe 9.5. Man kann $=$ aber auch mithilfe von $<$ und **Junktoren** und ohne Rekursion definieren. Wie würde das aussehen?

Aufgabe 9.6. Wenn man erst einmal $<$ und $=$ hat, kann man $<=$, $>$ und $>=$ ganz schnell und ohne Rekursion definieren. Machen Sie das bitte.

Im Prinzip können wir auch die Subtraktion ähnlich wie (9.1) erledigen:

$$a - b = a - (b' + 1) = (a - b') - 1 \quad (9.2)$$

Es gibt jedoch ein Problem: Würden wir es so machen, so könnten sich negative Zahlen ergeben. Eine „Subtraktion“ nur für natürliche Zahlen muss (analog zu Aufgabe 6.9) sinnvollerweise so definiert sein, dass $a - b$ null ergibt, wenn b größer als a ist. Daher definieren wir das folgendermaßen.

```
(define -
  (lambda (a b)
    (cond ((zero? b) a)
          ((< a b) 0)
          (else (sub1 (- a (sub1 b))))))
```

Aufgabe 9.7. Die Definition von $-$ war problematisch, weil wir die RACKET-Version von `sub1` verwenden müssen, die beim Aufruf `(sub1 0)` nicht 0, sondern -1 liefert. Definieren Sie „unser“ `sub1` als `my-sub1` mithilfe von `sub1` und definieren Sie dann eine einfachere Version von $-$, die `my-sub1` verwendet.

Aufgabe 9.8. Die in Abschnitt 3 erwähnte Funktion `quotient` können Sie nun auch (für natürliche Zahlen) definieren. Der „Trick“ ist dabei, zu „zählen“, wie oft der **Divisor** vom **Dividenten** abgezogen werden kann. („Rekursives Zählen“ haben Sie in Abschnitt 8 bereits mehrfach gesehen. Gleich die erste dort definierte Funktion `length` macht das beispielsweise.)

Aufgabe 9.9. Was passiert, wenn das zweite Argument in der Lösung von Aufgabe 9.8 die Zahl null ist?

Aufgabe 9.10. Jetzt sollte `modulo` einfach sein, oder? (Wenn Sie ehrgeizig sind, überlegen Sie sich sogar zwei Varianten – eine rekursive und eine ohne Rekursion, die `quotient` verwendet.)

Aufgabe 9.11. `even?` haben wir bereits in Aufgabe 6.2 definiert und diese Definition würde mit den Funktionen aus diesem Abschnitt harmonisieren. Man kann `even?` aber auch rekursiv und nur mit `zero?` und `sub1` definieren. Probieren Sie es aus!

Es folgen noch ein paar weitere Aufgaben zum Einüben von Rekursion, bei denen das Niveau nach und nach ein bisschen angehoben wird. Ab hier sollten Sie alle Definitionen aus diesem Abschnitt vorsichtshalber wieder entfernen. Sie sind zwar korrekt, aber insbesondere bei größeren Zahlen sehr ineffizient. Es ist besser, die „Originalen“ von RACKET zu verwenden.

Entscheidend ist, dass man immer auf einer Menge von Objekten arbeitet, die Rekursion überhaupt erlaubt.⁵⁰ Die für uns relevanten Fälle sind Listen und natürliche Zahlen.

Aufgabe 9.12. Schreiben Sie eine rekursive Funktion, die die dezimale **Quersumme** ihres Arguments berechnet. (Tipp: Die letzte Ziffer einer Zahl n erhält man durch $(\text{modulo } n \ 10)$. Wie erhält man die restlichen Ziffern und was macht man mit denen?)

Aufgabe 9.13. Programmieren Sie rekursiv die in Abschnitt 3 erwähnte Funktion `expt` für natürliche Zahlen. Dafür sollten Sie sich überlegen, was a^0 ist und welcher Zusammenhang zwischen a^b und a^{b-1} besteht. Sie dürfen natürlich `*` verwenden.

Aufgabe 9.14. Schreiben Sie eine Funktion, die rekursiv die **Fakultät** berechnet.

Die folgende Funktion `prime?` verwendet eine Technik, die manchmal hilfreich sein kann: Die Rekursion wird in eine (gewissermaßen „unsichtbare“) Hilfsfunktion ausgelagert, weil sie einen oder mehrere zusätzliche Parameter benötigt.⁵¹

Wollen wir testen, ob eine Zahl n eine **Primzahl** ist, ohne **fortgeschrittene Techniken** einzusetzen, so müssen wir n sukzessive durch alle Zahlen ab 2 bis inklusive $\lfloor \sqrt{n} \rfloor$ teilen. Das können wir rekursiv machen, müssen aber irgendwie „Buch führen“ darüber, welche potentiellen Teiler bereits ausprobiert wurden. Eine Möglichkeit, das mit den uns zur Verfügung stehenden Mitteln zu realisieren, sieht so aus:

```
(define prime?-helper
  (lambda (n k)
    (cond ((> (* k k) n) #t)
          ((zero? (modulo n k)) #f)
          (else (prime?-helper n (add1 k))))))

(define prime?
```

⁵⁰Ganz allgemein ist Rekursion immer dann möglich, wenn man es mit einer sogenannten *wohl-fundierten Relation* zu tun hat.

⁵¹Auf die Gefahr hin, mich zu wiederholen: Es gibt natürlich diverse Möglichkeiten, diese Aufgabe eleganter zu lösen. Das ist aber nicht das, worum es hier geht.

```
(lambda (n)
  (and (> n 1) (prime?-helper n 2))))
```

Dabei ist `prime?-helper` die Funktion, die die eigentliche Arbeit verrichtet. Es wird getestet, ob n durch k teilbar ist, und wenn dem nicht so ist, wird der Test stattdessen mit $k + 1$ statt k durchgeführt. `prime?` ist quasi das "Frontend", damit der Benutzer dieser Funktion nicht den zweiten Parameter eingeben muss, der aus seiner Sicht ein Implementationsdetail ist, das er nicht kennen muss.

Abgesehen von der Aufteilung in zwei Funktionen unterscheidet sich diese Rekursion auch darin von den bisherigen, dass der sich verändernde Wert (nämlich k) größer statt kleiner wird. Wichtig ist jedoch bei rekursiver Programmierung immer, dass es eine Abbruchbedingung gibt, die zuerst geprüft wird und die garantiert irgendwann erfüllt ist. Im konkreten Fall ist es die Bedingung $k^2 > n$ die äquivalent zu $k > \sqrt{n}$ ist.

Aufgabe 9.15. Verwenden Sie die Technik der Hilfsfunktion, um eine Funktion mit dem Namen `position` zu programmieren, die als Argument ein Atom und eine Liste bekommt und die die Position angibt, an der das Atom zum ersten Mal in der Liste auftritt. Ist das Atom kein Element der Liste, soll `#f` zurückgegeben werden.

Aufgabe 9.16. Wir definieren die *Tiefe* einer Liste rekursiv wie folgt: Eine Liste, die außer Atomen keine anderen Elemente hat, hat die Tiefe 0. Anderenfalls ist ihre Tiefe um 1 größer als die maximale Tiefe der Elemente, die Listen sind. Das würde z. B. so aussehen:

```
> (depth '(a b c))
0
> (depth '(a (b c) d))
1
> (depth '(a (b c) d (e f)))
1
> (depth '(a (b (c d)) (e f)))
2
> (depth '(a (b (c d)) (e (f (g (h))))))
4
```

Schreiben Sie `depth`. Die wesentliche Idee ist in diesem Fall, dass diese Funktion sich selbst aufruft, wenn sie auf ein Element trifft, das eine Liste ist. Sie braucht eine Hilfsfunktion, die sich die aktuell größte Tiefe „merkt“. Verwenden Sie ggf. `max` aus Aufgabe 6.8.

Aufgabe 9.17. Programmieren Sie ein Prädikat `tree-equal?`, das überprüft, ob zwei S-Ausdrücke denselben Inhalt haben, indem es wie die Funktion `depth` aus Aufgabe 9.16 rekursiv in die Listenstruktur „hinabsteigt“.

```
> (tree-equal? '(1 2 (3 (4 5) 6) (7) 8)
               '(1 2 (3 (4 5) 6) (7) 8))
#t
```

```
> (tree-equal? '(1 2 (3 (4 5) 6) (7) 8)
                '(1 2 (3 (4 5 6)) (7) 8))
#f
```

Beachten Sie, dass `eqv?` in beiden Fällen die Antwort `#f` gegeben hätte.

Aufgabe 9.18. Programmieren Sie eine Funktion `subst`, die als Argumente ein Atom und zwei S-Ausdrücke erwartet. Ein Aufruf der Form `(subst new old target)` soll als Ergebnis einen S-Ausdruck liefern, in dem das Atom `old` überall durch `new` ersetzt wurde. Dabei ist mit *überall* gemeint, dass die Funktion wie `depth` aus Aufgabe 9.16 rekursiv in die Listenstruktur „hinabsteigt“. Es soll also so funktionieren:

```
> (subst 23 42 '(3 42 (12 (42 18) 42) (42) 11))
'(3 23 (12 (23 18) 23) (23) 11)
```

(Hinweis: Hier bietet sich `map` aus Aufgabe 8.13 an.)

Wenn `f` eine rekursive Funktion ist, die natürliche Zahlen als Argumente erhält, dann programmieren wir typischerweise so, dass wir z. B. beim Berechnen von `(f 13)` davon ausgehen, dass `(f 12)` „schon bekannt“ ist. Man kann aber auch mehr als einen Schritt „nach hinten“ schauen. Warum nicht auch auf `(f 11)` zugreifen oder sogar auf alle Werte von `(f 0)` bis `(f 12)`?

Ein bekanntes Beispiel ist die *Fibonacci-Folge*, die folgendermaßen definiert ist: Die beiden Startwerte sind $f_0 = 0$ und $f_1 = 1$. Jeder weitere Wert wird rekursiv als Summe der beiden vorherigen definiert, also beispielsweise f_6 als $f_5 + f_4$ und allgemein $f_{n+2} = f_{n+1} + f_n$ für alle natürlichen Zahlen n . Der Anfang der Folge sieht also so aus:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Aufgabe 9.19. Programmieren Sie die Fibonacci-Folge rekursiv. Ihre Funktion soll also f_n zurückgeben, wenn ihr Argument n ist. Beachten Sie, dass es in diesem Fall *zwei* Abbruchbedingungen ($n = 0$ und $n = 1$) gibt.

Aufgabe 9.20. Schreiben Sie mithilfe von `prime?` eine rekursive Funktion mit dem Namen `next-prime`, die die erste Primzahl findet, die mindestens so groß wie ihr Argument ist. Sie soll also so funktionieren:

```
> (next-prime 23)
23
> (next-prime 24)
29
```

Aufgabe 9.21. Schreiben Sie mithilfe von `prime?` und Aufgabe 9.20 eine rekursive Funktion `nth-prime`, die die n -te Primzahl findet, wenn ihr Argument n ist.

```
> (nth-prime 1)
2
```

```
> (nth-prime 10)
29
```

Den Fall $n = 0$ können Sie ignorieren.

★**Aufgabe 9.22.** Wir definieren ähnlich wie im klassischen Beweis des [Satzes von Euklid](#) eine Folge, bei der p_{n+1} für $n \in \mathbb{N}$ der kleinste [Primteiler](#) von $1 + p_0 \cdot p_1 \cdot \dots \cdot p_n$ sein soll. Als Startwert nehmen wir $p_0 = 2$. Die ersten sechs Werte der Folge sind 2, 3, 7, 43, 13 und 53. Programmieren Sie eine rekursive Funktion, die p_n bei der Eingabe n berechnet. (Sie benötigen wahrscheinlich mehrere Hilfsfunktionen.)

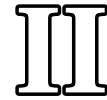
★**Aufgabe 9.23.** Eine bekannte Funktion aus der [theoretischen Informatik](#) ist die [Ackermannfunktion](#), die man beispielsweise durch

$$a(n, m) = \begin{cases} m + 1 & n = 0 \\ a(n - 1, 1) & n > 0 \wedge m = 0 \\ a(n - 1, a(n, m - 1)) & \text{sonst} \end{cases}$$

für alle $m, n \in \mathbb{N}$ definieren kann. Programmieren Sie a als rekursive SCHEME-Funktion. Die Rekursion verläuft hier über beide Argumente und im dritten Fall quasi „doppelt“. Können Sie begründen, warum sie trotzdem für jedes Paar von Argumenten garantiert endet?

(Achtung: Der „Witz“ an dieser Funktion ist ihr extrem starkes Wachstum. Wenn Ihre Funktion für $a(3, 11)$ sehr lange braucht und für $a(4, 1)$ scheinbar in einer [Endlosschleife](#) steckt, ist das eher ein gutes Zeichen.)

Aufgabe 9.24. Nehmen Sie sich einige Funktionen aus diesem Kapitel vor und schreiben Sie sie im [iterativen Stil](#) (also ohne Rekursion) in einer „konventionellen“ Sprache wie JAVA oder PYTHON.



Der Lambda-Kalkül

Der **Lambda-Kalkül** wurde 1936 von dem amerikanischen Mathematiker **Alonzo Church** entwickelt. Ursprünglich ging es darum, das sogenannte **Entscheidungsproblem** der **mathematischen Logik** zu lösen. Der Kalkül spielte in der Folge jedoch auch eine wichtige Rolle in der theoretischen Informatik und prägte viele später entstandene Programmiersprachen. Insbesondere war **die erste LISP-Version** in gewissem Sinne ein direkter Abkömmling des Lambda-Kalküls. Auch für die Typentheorie, um die es **im nächsten Kapitel** geht, spielt der Kalkül eine grundlegende Rolle.

Falls Sie das Modul **Theoretische Informatik** bereits absolviert haben: Mit dem Lambda-Kalkül kann man alle Berechnungen ausdrücken, die man mit **Turingmaschinen** ausdrücken kann. Der Kalkül ist in diesem Sinne also **turingvollständig**.

Bei der Arbeit mit dem Lambda-Kalkül muss man sich vergegenwärtigen, dass er zu einer Zeit entstanden ist, als es noch keine Computer gab. Man arbeitete also mit Zettel und Stift. Obwohl wir ab und zu einen Computer als Unterstützung verwenden werden (siehe Aufgabe 11.1), sollten Sie in der Lage sein, die wesentlichen Argumentationen auch ohne elektronische Hilfe nachzuvollziehen.

Die SCHEME-Programmieraufgaben in diesem Kapitel sind übrigens als **didaktisches** Hilfsmittel vorgesehen. Nach meiner Erfahrung versteht man neue Konzepte wesentlich besser, wenn man sie selbst programmiert hat.

10. Die Syntax von Lambda-Ausdrücken

Wir legen zunächst lediglich die **Syntax** von sogenannten *Lambda-Ausdrücken* fest, ohne etwas über deren **Semantik** zu sagen. Wir reden also in diesem Abschnitt noch nicht darüber, wofür diese Ausdrücke stehen bzw. wie sie verwendet werden sollen.

Als Grundbaustein benötigen wir *Variablen*.

Es gibt eine unendliche Menge von **Variablen**: x_1 , x_2 , x_3 und so weiter.

Darauf aufbauend folgt nun die eigentliche Definition:

Wir definieren **rekursiv**, was ein **Lambda-Ausdruck** bzw. **λ -Ausdruck** ist:

- (L1) Variablen sind λ -Ausdrücke.
- (L2) Sind A und B λ -Ausdrücke, dann ist auch $(A B)$ ein λ -Ausdruck. Wir nennen das auch die **Applikation** oder **Anwendung** von A auf B .
- (L3) Ist A ein λ -Ausdruck und x_k eine Variable, dann ist auch $(\lambda x_k. A)$ ein λ -Ausdruck.¹ Wir nennen das auch die **Abstraktion** von A bezüglich x_k .

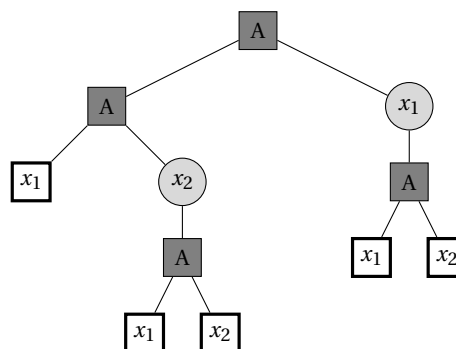
Damit sind z. B. x_3 und x_{42} nach (L1) λ -Ausdrücke. Nach (L2) ist daher auch $(x_3 x_{42})$ ein λ -Ausdruck. Erneute Anwendung von (L2) liefert unter anderem, dass auch $(x_3 (x_3 x_{42}))$ und $((x_3 x_{42}) (x_3 x_{42}))$ λ -Ausdrücke sind. Nach (L3) folgt, dass z. B. auch $(\lambda x_3. x_{42})$ und $(\lambda x_3. (x_3 x_{42}))$ λ -Ausdrücke sind. Und so weiter.

Das ist wieder eine **rekursive Definition** wie auf Seite 13. Beachten Sie, dass darin auch die unausgesprochene Festlegung enthalten ist, dass etwas, was *nicht* durch (endlich viele) Anwendungen der genannten Regeln entstanden ist, nicht dazugehört. Beispielsweise ist (x_1) *kein* λ -Ausdruck denn Klammern kommen nur in den Regel (L2) und (L3) vor. Wären die angewandt worden, dann stünden zwischen den Klammern entweder zwei Ausdrücke oder das Symbol λ müsste dort auftauchen.

Aufgabe 10.1. Welche der folgenden Zeichenketten sind λ -Ausdrücke? Begründen Sie Ihre Antworten.

$\lambda x_1. x_1$	$(\lambda x_1 x_2. x_1)$	$(\lambda. x_1)$	$(\lambda x_1. x_1 x_1)$
$x_1 x_2$	$(x_1 x_2 x_3)$	$(x_1 (x_2 x_3))$	$(x_1 (x_2) x_3)$
$(\lambda x_1. (x_1 x_2))$	$(\lambda x_1. (\lambda x_1. x_1))$	$((\lambda x_1. (\lambda x_1. x_1)) x_2)$	$(x_2 (\lambda x_1. x_1))$

Es ist häufig hilfreich, sich die Struktur eines λ -Ausdrucks grafisch klarzumachen. Dafür eignen sich **Bäume**.



In der obigen Grafik wird $((x_1 (\lambda x_2. (x_1 x_2))) (\lambda x_1. (x_1 x_2)))$ dargestellt. **A** steht für eine Applikation und hat immer genau zwei Kinder. **x_k** steht für einen Abstraktion bezüglich x_k und hat immer genau ein Kind. **x_m** steht für die Variable x_m und ist immer ein Blatt des Baums.

¹Man beachte den Punkt hinter der Variablen.

Weil das besser lesbar ist, werden wir im Folgenden häufig beliebige **lateinische** Kleinbuchstaben für Variablen verwenden. Wir schreiben also eher x , y und z statt x_1 bis x_3 .

Aufgabe 10.2. Wir definieren λ -Ausdrücke noch einmal in etwas veränderter Form, damit wir sie in SCHEME möglichst einfach untersuchen können:²

(LS0) Alle Symbole außer `lambda` sind Variablen.

(LS1) Variablen sind λ -Ausdrücke.

(LS2) Sind A und B λ -Ausdrücke, dann ist auch die Liste $(A\ B)$ ein λ -Ausdruck.

(LS3) Ist A ein λ -Ausdruck und x eine Variable, dann ist die Liste $(\text{lambda } (x)\ A)$ ebenfalls ein λ -Ausdruck.

Schreiben sie ein rekursives SCHEME-Prädikat, das solche λ -Ausdrücke erkennt.

```
> (lambda-expr? 'foo)
#t
> (lambda-expr? '(lambda (x) y))
#t
> (lambda-expr? '(lambda (x y) y))
#f
> (lambda-expr? '(lambda () y))
#f
> (lambda-expr? '(lambda (x) y e))
#f
> (lambda-expr? '(lambda (x) (y e)))
#t
> (lambda-expr? '(lambda x))
#f
> (lambda-expr? '(lambda x))
#t
> (lambda-expr? '((x y)))
#f
> (lambda-expr? '((x y) z))
#t
```

In der Mathematik ist man gerne faul und fasst sich kurz, um unnötige Schreibarbeit zu vermeiden. So entstehen Regel wie "**Punktrechnung vor Strichrechnung**". Das führen wir nun auch für λ -Ausdrücke ein:

Bei einem λ -Ausdruck darf das äußerste Klammerpaar grundsätzlich weggelassen werden. Mehrere aufeinanderfolgende λ -Ausdrücke werden **linksassoziativ** als verschachtelte Applikationen interpretiert. Applikation hat eine höhere **Priorität** als Abstraktion. Klammern hinter dem Punkt einer Abstraktion der Form $\lambda x.A$ können weggelassen werden, wenn sie lediglich äußere Klammern des λ -Ausdrucks A gewesen wären.

²Diese Definition gilt nur für die Aufgaben, in denen wir etwas mit SCHEME machen.

Nach der ersten Regel ist z. B. $\lambda x.(x y)$ eine Kurzschreibweise für $(\lambda x.(x y))$. Nach der zweiten Regel können wir $x y z$ statt $(x y) z$ schreiben, was nach der ersten Regel wiederum eine Abkürzung für $((x y) z)$ ist. Nach der dritten Regel können wir $\lambda x.y z$ als Kurzschreibweise für $\lambda x.(y z)$ verwenden. Nach der vierten Regel ist $\lambda x.\lambda y.z$ eine Abkürzung für $\lambda x.(\lambda y.z)$.

Beachten Sie jedoch, dass man bei $x (y z)$ *nicht* die Klammern weglassen kann, weil das dann (siehe letzter Absatz) als $(x y) z$ interpretiert würde. Ebenso müssen die Klammern bei $(\lambda x.y) z$ erhalten bleiben, denn sonst würde das wie $\lambda x.(y z)$ behandelt werden.

Bei solchen Konventionen muss allerdings klar sein, dass wir hier über *Abkürzungen* sprechen. *Gemeint* ist damit trotzdem immer ein formal korrekter λ -Ausdruck im Sinne der Definition am Anfang des Abschnitts! Außerdem sollten Sie wie bei allen Klammerregeln im Zweifelsfall lieber „zu viele“ Klammern verwenden, wenn Sie sich nicht sicher sind.

Aufgabe 10.3. Welche der Zeichenketten aus Aufgabe 10.1, die eigentlich keine λ -Ausdrücke sind, wären nach dieser neuen Konvention erlaubte Abkürzungen für λ -Ausdrücke und wie würden diese Ausdrücke in voller Länge aussehen?

Aufgabe 10.4. Geben Sie für die folgenden Ausdrücke jeweils die ausführliche Form mit allen Klammern an, wobei Sie das äußerste Klammerpaar weglassen können.

$$\lambda x.y x z \quad \lambda x.\lambda y.\lambda z.w \quad (\lambda x.y)(\lambda x.y z) w \quad (\lambda x.x) x y$$

Aufgabe 10.5. Auch für die λ -Ausdrücke aus Aufgabe 10.2 wollen wir Abkürzungen einführen. Das geht allerdings nicht ganz so „radikal“ wie eben, weil es sich weiterhin um *S-Ausdrücke* handeln muss – wir müssen also zumindest die äußeren Klammern beibehalten. Außerdem tasten wir die Klammern um die Abstraktionsvariable ebenfalls nicht an, damit es nach wie vor wie eine lambda-Funktion in SCHEME aussieht. Wir legen zwei Regeln fest:

Erstens wird ein Ausdruck der Form³ $(\text{lambda } (x) A_1 A_2 \dots)$ als Abkürzung für $(\text{lambda } (x) (A_1 A_2 \dots))$ aufgefasst, wobei evtl. noch die zweite Regel auf den Teil hinter (x) angewendet wird. Zweitens wird eine Liste der Form $(A_1 A_2 A_3 \dots)$ mit mindestens drei Einträgen als Abkürzung für die Liste $((A_1 A_2) A_3 \dots)$ interpretiert, wobei dieselbe Regel auf diese Liste ggf. erneut (rekursiv) angewendet werden kann.

Schreiben Sie eine Funktion, die solche abgekürzten Ausdrücke in die Form übersetzt, die von `lambda-expr?` akzeptiert wird. Ausdrücke, die nicht als Abkürzung erkannt werden, sollen einfach unverändert zurückgegeben werden.

```
> (expand 'x)
'x
> (expand '(x y))
'(x y)
> (expand '(lambda (x) y))
'(lambda (x) y)
```

³Das ist so gemeint, dass x eine Variable ist und dahinter noch mindestens zwei Elemente folgen.

```

> (expand '(lambda (x) (y z)))
'(lambda (x) (y z))
> (expand '(lambda (x) y z))
'(lambda (x) (y z))
> (expand '(x y z))
'((x y) z)
> (expand '(x (y z)))
'(x (y z))
> (expand '(x y z w))
'(((x y) z) w)
> (expand '(lambda (x) (y z w)))
'(lambda (x) ((y z) w))
> (expand '(lambda (x) y z w))
'(lambda (x) ((y z) w))
> (expand '(lambda (x) y (z w)))
'(lambda (x) (y (z w)))

```

11. Die intendierte Semantik

In der Mathematik gibt man die Ausführungsvorschrift einer [Funktion](#) häufig in einer Form wie $x \mapsto x + 2$ an. x ist dabei ein Platzhalter für das Argument und $x + 2$ beschreibt den zugehörigen Funktionswert. Das ist natürlich so gemeint, dass man beispielsweise bei der Berechnung des Funktionswertes für das Argument 40 im Term $x + 2$ die Variable x durch den konkreten Wert 40 ersetzt.

In SCHEME würde diese Funktion die Form

```
(lambda (x) (+ x 2))
```

haben und das Einsetzen des Arguments 40 würde so aussehen:

```
((lambda (x) (+ x 2)) 40)
```

Der Lambda-Kalkül (der das „Vorbild“ von LISP und damit von SCHEME war) macht das im Prinzip genauso. Abgesehen davon, dass Symbole wie $+$, 2 und 40 nicht zum Kalkül gehören, würde man die obige Funktion als $\lambda x.(x + 2)$ schreiben und deren Anwendung (Applikation) auf 40 als $(\lambda x.(x + 2)) 40$. Das intendierte Resultat dieser Anwendung wäre der Ausdruck $(40 + 2)$, den man dadurch erhält, dass man in $(x + 2)$ die Variable x durch 40 ersetzt.

Das ist die wesentliche Idee, die allerdings wegen bestimmter Sonderfälle nicht ganz so einfach zu formulieren ist. Wir fangen damit aber erst einmal an, bevor wir die Sache im folgenden Abschnitt präzisieren.

Eine Applikation der Form $(\lambda x.A) B$ wird **β -Redex** (für *reducible expression*) genannt. *Nur für diesen Abschnitt (!)* legen wir *provisorisch* fest, dass ein β -

Reduktionsschritt das Ersetzen so eines β -Redexes durch den λ -Ausdruck sein soll, der entsteht, indem man in A jedes x durch B ersetzt.

Im Folgenden wird der Präfix „ β -“ meistens weggelassen. Außerdem verwenden wir auch Formulierungen wie „wird reduziert“, womit die Durchführung eines Reduktionsschrittes gemeint ist.

Ein paar Beispiele:

- $\lambda x.y$ beschreibt das, was man in der Mathematik eine **konstante Funktion** nennt: Es kommt immer y heraus, weil x rechts vom Punkt gar nicht vorkommt, also auch nicht ersetzt werden kann. Der Redex $(\lambda x.y) x$ wird somit zu y reduziert, die Redexe $(\lambda x.y) y$ und $(\lambda x.y) w$ aber auch.
- Auch ein Teilausdruck eines λ -Ausdruck kann ein Redex sein, der reduziert wird. Ein Beispiel wäre, dass aus $x ((\lambda x.y) x)$ der Ausdruck $x y$ wird.
- $\lambda x.x$ beschreibt die **Identität**, bei der als Funktionswert einfach immer das Argument herauskommt. Der Redex $(\lambda x.x) y$ wird zu y , $(\lambda x.x) z$ wird zu z .
- $\lambda x.x x$ – oder ausführlicher $\lambda x.(x x)$ – „verdoppelt“ das Argument. Aus $(\lambda x.x x) w$ wird z. B. $w w$.
- $\lambda x.x y x$ macht dasselbe wie der vorherige Ausdruck, fügt aber noch ein y in der Mitte hinzu. Aus $(\lambda x.x y x) w$ wird $w y w$, aus $(\lambda x.x y x) y$ wird $y y y$.

Aufgabe 11.1. Unter <https://weitz.de/files/lambda.zip> finden Sie ein ZIP-Archiv, das eine Erweiterung für DRRACKET enthält, mit der Sie λ -Ausdrücke wie in den obigen Beispielen ausprobieren können. Sie sollten das herunterladen, die Anleitung durchlesen und die Software dann installieren. Als Eingewöhnung können Sie vielleicht die gerade gezeigten Beispiele durchgehen.

Dieses Tool wird im Folgenden als DRLAMBDA bezeichnet.

Aufgabe 11.2. Was kommt bei der Reduktion der folgenden drei Applikationen jeweils heraus?

$$(\lambda x.x x) w w \qquad (\lambda x.x x)(w w) \qquad \lambda x.x x w w$$

Fassen wir die intendierte Semantik der λ -Ausdrücke noch einmal zusammen:

- Ist f eine Funktion und A ein Objekt, das man in die Funktion einsetzen kann, so schreibt man in der Mathematik für den Funktionswert $f(A)$. Im Lambda-Kalkül schreibt man stattdessen $(f A)$ oder kurz einfach $f A$.
- Eine Funktion die x auf A abbildet (wobei A in der Regel ein von x abhängender **Term** ist), schreibt man in der Mathematik $x \mapsto A$. Im Lambda-Kalkül schreibt man dafür $(\lambda x.A)$.

Der Unterschied zur Mathematik ist, dass man dort für A „interessante“ Terme wie $\sin x$ oder x^2 einsetzen kann, während wir im Lambda-Kalkül nur „langweilige“ Zeichenketten wie $x y x$ zur Verfügung haben. Wie daraus trotzdem mehr werden kann, klärt sich erst in Abschnitt 13. Bis dahin müssen wir noch ein bisschen Arbeit leisten.

Wenn Sie einen arithmetischen Ausdruck wie $3 \cdot (4 + 5)$ berechnen, dann gehen Sie in der Regel Schritt für Schritt vor. Im Beispiel würden Sie erst die Summe $4 + 5$ berechnen und anschließend das Ergebnis 9 mit 3 multiplizieren. Sie sind fertig, wenn alle Rechenzeichen verschwunden sind.

So soll auch die Umwandlung umfangreicherer λ -Ausdrücke ablaufen: Solange es Redexe gibt, kann weiter reduziert werden. (Und wir werden dann auch die Anwendung mehrerer Reduktionsschritte nacheinander als **Reduktion** bezeichnen.) Dafür ebenfalls zwei Beispiele:

- $(\lambda x.(x\ y))\ ((\lambda z.(z\ z\ z))\ w)$:

Man kann zuerst den innere Redex $(\lambda z.(z\ z\ z))\ w$ zu $(w\ w\ w)$ reduzieren und erhält $(\lambda x.(x\ y))\ (w\ w\ w)$, woraus dann $(w\ w\ w)\ y$ wird. (Die Klammern im letzten Ausdruck sind überflüssig.) Man kann auch erst „außen“ reduzieren, wodurch man $((\lambda z.(z\ z\ z))\ w)\ y$ bekommt, was nach einer weiteren Reduktion zum selben Ergebnis führt.

- $(\lambda x.(\lambda y.(y\ x)))\ w\ v$:

Hier gibt es nur eine Möglichkeit, mit der Reduktion zu beginnen: Im Ausdruck $(\lambda y.(y\ x))$ wird x durch w ersetzt. Das liefert insgesamt $(\lambda y.(y\ w))\ v$. Und daraus wird schlussendlich $v\ w$.

Man kann das als eine Funktion betrachten, die im Sinne der **funktionalen Programmierung** eine Funktion als Rückgabewert liefert. Oder als eine Version des auf Seite 17 angesprochenen *Currying*, durch die wir gewissermaßen eine Applikation mit zwei Variablen erhalten.

Wie wir am ersten der beiden Beispiele gesehen haben, hat man manchmal mehrere Redexe zur Auswahl. Wir legen daher zusätzlich fest:

Wenn ein λ -Ausdruck mehrere Redexe enthält, bedeutet **β -Reduktion in Standardreihenfolge**,⁴ dass als erstes der Redex reduziert wird, der von allen Redexen, die nicht in anderen Redexen enthalten sind, am weitesten links steht. Danach wird dieselbe Regel auf das Resultat angewendet.

Mit anderen Worten kommt immer zuerst der Redex dran, dessen λ am weitesten links steht. **DRLAMBDA** führt β -Reduktion in Standardreihenfolge durch, wenn Sie einen λ -Ausdruck eingeben. (Bei komplexeren Ausdrücken kann die Reduktion auch schrittweise angezeigt werden. Das wird in der Dokumentation von **DRLAMBDA** erklärt.)

Aufgabe 11.3. Überlegen Sie sich bei den folgenden Ausdrücken jeweils, zu was sie reduziert werden, und überprüfen Sie Ihr Ergebnis dann in **DRRACKET**.

- (i) $(\lambda x.\lambda y.z)\ w\ w$
- (ii) $(\lambda x.x\ x)\ ((\lambda w.w\ w)\ v)$
- (iii) $(\lambda x.\lambda y.x)\ v\ w$

Aufgabe 11.4. Was passiert, wenn man $(\lambda x.(x\ x))\ (\lambda x.(x\ x))$ reduziert?

Ω

⁴In Fachartikeln meistens *standard* oder *normal order* genannt.

Aufgabe 11.5. Schreiben Sie eine SCHEME-Funktion, die für λ -Ausdrücke wie in Aufgabe 10.2 eine β -Reduktion in Standardreihenfolge durchführt. (Das ist zugegebenermaßen etwas aufwendiger, aber der Lerneffekt dürfte entsprechend groß sein. Die Funktion `subst` aus Aufgabe 9.18 könnte übrigens hilfreich sein.)

12. Beta-Reduktion

Bevor wir uns klarmachen, warum die im letzten Abschnitt provisorisch eingeführte β -Reduktion noch präzisiert werden muss, sollte der wesentliche Unterschied zwischen SCHEME und dem Lambda-Kalkül verdeutlicht werden, weil manche Dinge oberflächlich ähnlich aussehen, es aber nicht sind.

Die prinzipielle Funktionsweise von SCHEME wurde in Abschnitt 3 erklärt: S-Ausdrücke werden *evaluiert*. Zu diesem Evaluationsprozess gehört ein abstraktes Maschinenmodell, in dem es u. a. Variablenbindungen und zeitliche Abläufe („erst die Argumente evaluieren“) gibt und in dem aus Ausdrücken *Werte* gemacht werden. Aus dem Ausdruck

$$((\text{lambda } (x) x) (\text{lambda } (y) y)) \quad (12.1)$$

wird z. B. eine *Prozedur*, DRACKET zeigt das als `#<procedure>` an. Diese Prozedur ist, wie wir in Abschnitt 5 gelernt haben, ein vollwertiges Objekt der Sprache, aber sie ist *nicht* das gleiche wie der S-Ausdruck `(lambda (y) y)`, den wir in SCHEME durch die Eingabe von

$$(\text{quote } (\text{lambda } (y) y))$$

erhalten würden. Der S-Ausdruck ist gewissermaßen der *Quellcode*, aus dem der *Compiler* bei der Auswertung die Prozedur gemacht hat.

Die β -Reduktion im Lambda-Kalkül macht hingegen aus λ -Ausdrücken wieder λ -Ausdrücke. Es gibt im Lambda-Kalkül keine „Werte“, sondern nur Ausdrücke. Betrachtet man das Pendant $(\lambda x.x) (\lambda y.y)$ zu (12.1),⁵ so ist das Ergebnis nach Reduktion $(\lambda y.y)$. Das ist immer noch ein λ -Ausdruck.

Schauen wir uns anhand eines simplen SCHEME-Beispiels an, was dieser Unterschied für Folgen nach sich ziehen kann.

```
(define f
  (lambda (x) (list x x)))
(define F
  (lambda (x) f))
```

`f` ist eine Funktion, die eine Liste zurückgibt, in der ihr Argument zweimal vorkommt. `F` ist eine Funktion, die ihr Argument ignoriert und einfach immer `f` zurückgibt. Daher wird bei der Eingabe `((F ' (42 42)) 23)` natürlich das Argument `(42 42)` ignoriert werden und der Wert wird `(23 23)` sein.

⁵Oder sogar wortwörtlich denselben Ausdruck wie in (12.1), auf den man *normal-reduction* aus Aufgabe 11.5 anwenden könnte.

In SCHEME kann man das alles bekanntlich auch in einer Zeile erledigen:

```
((lambda (x) (lambda (x) (list x x))) '(42 42)) 23)
```

Und selbstverständlich kommt da auch (23 23) heraus – *obwohl* beide Funktionen denselben Namen x für ihr Argument verwenden. Dass das kein Problem ist, liegt am erwähnten Evaluationsmodell von SCHEME, das bei jedem Funktionsaufruf temporär eine neue Variablenbindung generiert, die eventuell vorhandene überdeckt.⁶

Wenn wir jedoch nach unserer provisorischen Definition den Ausdruck⁷

$$(\lambda x.(\lambda x.(x x)))(v v) w \quad (12.2)$$

reduzieren, so erhalten wir $(\lambda(v v).((v v)(v v))) w$, was nicht einmal mehr ein λ -Ausdruck ist. Wir müssen daher unsere Definition aus dem letzten Abschnitt anpassen. Allerdings wird die folgende immer noch nicht die endgültige sein.

Wir setzen *provisorisch* fest, dass ein **β -Reduktionsschritt** das Ersetzen eines β -Redexes $(\lambda x.A) B$ durch den λ -Ausdruck sein soll, der entsteht, indem man in A jedes x durch B ersetzt – wobei allerdings die Teilausdrücke von A ausgenommen sind, die Abstraktionen bezüglich ebendieser Variablen x sind.

$(\lambda x.(\lambda x.(x x)))(v v)$ würde also nach dieser Regel zu $\lambda x.(x x)$ reduziert werden. Oder, um es noch deutlicher zu machen: Der Ausdruck $(\lambda x.x (\lambda x.x x))(v v)$ würde zu $v v (\lambda x.x x)$ reduziert werden und (12.2) zu $w w$.

Aufgabe 12.1. Prüfen Sie das mit **DR λ BDA** nach. Dort wird natürlich die *richtige* β -Reduktion durchgeführt.

Aufgabe 12.2. Modifizieren Sie den Code aus der Lösung von Aufgabe 11.5 (oder Ihren eigenen), um diese neue Regelung umzusetzen. Zum Ausprobieren können Sie zu `test-cases` den Fall

```
(((((lambda (x) (lambda (x) (x x))) (v v)) w) (w w))
```

hinzufügen.

Das verbleibende Problem kann man wieder mit einem SCHEME-Beispiel illustrieren. Wie beim *Currying* (siehe Seite 17) bauen wir uns aus zwei einstelligen Funktionen etwas zusammen, das sich wie eine zweistellige Funktion verhält.

⁶Im konkreten Fall gibt F die Funktion f als **Funktionsabschluss** zurück, in dem die Variable x an die Liste (42 42) gebunden ist. (Evtl. auch nicht, weil x ignoriert wird. Aber das ist eine Optimierung, die für diese Diskussion irrelevant ist.) Wenn f später aufgerufen wird, wird jedoch eine neue Bindung erzeugt, die x an 23 bindet. Die „alte“ Bindung an (42 42) wird dadurch überdeckt.

⁷Hier spielen $(v v)$ und w die Rollen von (42 42) und 32. Außerdem wurden „überflüssige“ Klammern gesetzt, um den Vergleich mit dem SCHEME-Ausdruck zu verdeutlichen.

```
(define x 'x)
(define y 'y)
(((lambda (x) (lambda (y) (list y x)))) y) x)
```

Diese Funktion vertauscht offenbar die Reihenfolge ihre Argumente und in der Tat erhalten wir als Resultat $(x\ y)$.

Wenn wir jedoch $(\lambda x.\lambda y.(y\ x))\ y\ x$ nach unserer aktuellen Regel reduzieren,⁸ so erhalten wir $x\ x$ und nicht wie erhofft $x\ y$. Der Grund wird deutlich, wenn wir nur $(\lambda x.\lambda y.(y\ x))\ y$ reduzieren. Da kommt nämlich $\lambda y.(y\ y)$ heraus. x wurde durch y ersetzt, aber y ist nicht irgendeine Variable, sondern es handelt sich um die Variable der Applikation, innerhalb derer die Ersetzung stattfand. Man nennt das *variable capture*, was man wohl am besten mit **Variablenfang** übersetzen kann.

Um auch dieses Problem zu beheben, müssen wir etwas mehr Arbeit leisten. Zunächst klären wir ein Konzept, das im Prinzip schon aus der Mathematik bekannt sein sollte. Die Idee ist, dass die Variable x innerhalb eines Ausdrucks wie $\lambda x.(x\ y)$ durch das λ *gebunden* ist. Sie spielt einerseits eine spezielle Rolle, ist aber andererseits austauschbar.

Beispiele, die Sie schon kennen sollten, sind

$$\{n \in \mathbb{N} : n^2 < 1000\}, \quad \sum_{k=1}^{100} k \quad \text{und} \quad \int_2^4 x^2 dx.$$

Im ersten ist n durch die Mengenschreibweise gebunden, im zweiten k durch die Summenschreibweise und im dritten x durch die Integralschreibweise. Würden wir die Menge als $\{m \in \mathbb{N} : m^2 < 1000\}$ schreiben, so würde sich *nichts* ändern – wir erhielten *dieselbe* Menge. Und analog könnten wir auch in der Summe oder im Integral andere Variablen verwenden.

Unser Ziel ist also, dass z. B. in $x\ (\lambda x.x)$ x das erste und vierte x *nicht* gebunden sind (das werden wir *frei* nennen), während die anderen beiden gebunden sind, weil sie im „Einflussbereich“ der Applikation stehen. Das präzisieren wir nun.

$F[A] \quad B[A]$

Wir ordnen rekursiv jedem λ -Ausdruck A Mengen $F[A]$ und $B[A]$ von Variablen zu. Die Elemente der ersten Menge nennen wir **frei** (*free*) in A , die der zweiten **gebunden** (*bound*) in A .

- (i) Für Variablen x setzen wir $F[x] = \{x\}$ und $B[x] = \emptyset$.
- (ii) Für Applikationen $A\ C$ setzen wir $F[A\ C] = F[A] \cup F[C]$ und ebenso $B[A\ C] = B[A] \cup B[C]$.
- (iii) Für Abstraktionen $\lambda x.A$ setzen wir $F[\lambda x.A] = F[A] \setminus \{x\}$ und schließlich noch $B[\lambda x.A] = B[A] \cup \{x\}$.

Damit haben wir beispielsweise

$$F[x\ y] = \{x, y\},$$

$$B[x\ y] = \emptyset,$$

⁸Oder auch nach der alten, das macht hier keinen Unterschied.

$$\begin{aligned}
 F[\lambda x.(x\ y)] &= \{y\}, & B[\lambda x.(x\ y)] &= \{x\}, \\
 F[z\ (\lambda x.(x\ y))] &= \{y, z\}, & \text{und } B[z\ (\lambda x.(x\ y))] &= \{x\}.
 \end{aligned}$$

Wann eine Variable frei oder gebunden ist, kann man sich anhand von Bäumen wie auf Seite 40 klarmachen: Wenn man den Weg einer Variable bis zur Wurzel des Baumes verfolgt und dabei auf einen Kreis trifft, in dem der Name dieser Variablen steht, dann ist sie gebunden, anderenfalls ist sie frei. In der angesprochenen Grafik ist z. B. x_1 an den ersten beiden Stellen frei, an der dritten jedoch nicht.

Aufgabe 12.3. Geben Sie jeweils die freien und gebundenen Variablen für die drei λ -Ausdrücke $x\ z\ (\lambda x.(x\ y))$, $z\ (\lambda y.(\lambda x.(x\ y)))$ und $(\lambda y.(x\ y))\ (\lambda x.(x\ y))$ an.

Aufgabe 12.4. Schreiben Sie rekursive Funktionen `free-vars` und `bound-vars`, die als Eingabe einen λ -Ausdruck im Sinne von Aufgabe 10.2 erhalten und die die Menge der freien bzw. gebundenen Variablen dieses Ausdrucks als Liste ausgeben. (Es bietet sich an, Funktionen aus den Aufgaben dieses Kapitels zu verwenden. Auch in Abschnitt 8 findet man diverse hilfreiche Funktionen für diesen Zweck.)

Dass gebundene Namen im Prinzip – wie im obigen Mengenbeispiel – austauschbar sind, gilt natürlich auch in der Informatik. In SCHEME sind die Funktionen

```
(lambda (x) (+ x 2))  und  (lambda (y) (+ y 2))
```

in ihrer Wirkung nicht unterscheidbar und wir wollen, dass auch im Lambda-Kalkül kein Unterschied besteht zwischen $\lambda x.(x\ x)$ und $\lambda y.(y\ y)$, wenn diese Applikationen als Teil eines Redex reduziert werden.

Auch hier muss man jedoch aufpassen:⁹

- (i) In $\lambda x.(x\ y)$ darf man die gebundene Variable x durch irgendeine andere ersetzen und erhält damit z. B. den gleichwertigen Ausdruck $\lambda z.(z\ y)$. Aber man darf als neue Variable *nicht* y nehmen, denn $\lambda y.(y\ y)$ hat offenbar eine andere Semantik. Der Grund ist, dass y in $\lambda x.(x\ y)$ frei ist, in $\lambda y.(y\ y)$ jedoch nicht.
- (ii) In $\lambda x.\lambda y.(y\ x)$ darf man x auch nicht durch y ersetzen. Der Grund ist jedoch ein anderer. Hier war die alte Abstraktionsvariable x im Rumpf $\lambda y.(y\ x)$ frei, während die sie ersetzende Variable y im neuen Rumpf $\lambda y.(y\ y)$ gebunden wäre.

So etwas kann man durch saubere Definitionen vermeiden. Dafür legen wir zunächst fest, wie genau *Ersetzung* zu funktionieren hat.

Für Variablen x und λ -Ausdrücke A und B definieren wir rekursiv, was mit $A[x \leftarrow B]$ gemeint sein soll:

(R1) $x[x \leftarrow B]$ ist B .

$A[x \leftarrow B]$

⁹Und das gleiche Prinzip gilt nicht nur für Lambda-Ausdrücke, sondern überall dort, wo Variablen gebunden werden.

- (R2) $y[x \leftarrow B]$ ist y , wenn y eine andere Variable als x ist.
 (R3) $(A_1 A_2)[x \leftarrow B]$ ist $(A_1[x \leftarrow B] A_2[x \leftarrow B])$.
 (R4) $(\lambda x A)[x \leftarrow B]$ ist $(\lambda x A)$.
 (R5) $(\lambda y A)[x \leftarrow B]$ ist $(\lambda y A[x \leftarrow B])$, wenn $y \notin F[B] \cup \{x\}$ gilt.
 (R6) $(\lambda y A)[x \leftarrow B]$ ist $(\lambda y_1 A[y \leftarrow y_1])[x \leftarrow B]$, wenn $y \in F[B] \setminus \{x\}$ gilt.
 Dabei muss y_1 eine Variable sein, die in $F[A] \cup F[B]$ *nicht* vorkommt.

Damit können wir endlich endgültig festlegen, was genau wir mit einem β -Reduktionsschritt meinen.

Ein **β -Reduktionsschritt** ist das Ersetzen eines β -Redexes $(\lambda x.A) B$ durch den λ -Ausdruck $A[x \leftarrow B]$.

Nun können wir zum problematischen Ausdruck $(\lambda x.\lambda y.(y x)) y$ zurückkehren. Nach der endgültigen Definition ist das $(\lambda y.(y x))[x \leftarrow y]$. Weil y frei in y ist, muss Regel (R6) angewandt werden. Wir führen also zunächst eine *neue* Variable w ein und bilden dann $(\lambda w.(y x)[y \leftarrow w])[x \leftarrow y]$. Im inneren Teil haben wir

$$(y x)[y \leftarrow w] \xrightarrow{(R3)} (y[y \leftarrow w] x[y \leftarrow w]) \xrightarrow{(R1)} (w x[y \leftarrow w]) \xrightarrow{(R2)} (w x)$$

und damit insgesamt $(\lambda w.(w x))[x \leftarrow y]$, was nach (R5) zu $(\lambda w.(w x)[x \leftarrow y])$ und schließlich wie erhofft zu $\lambda w.(w y)$ wird.

Aufgabe 12.5. Geben Sie $(\lambda x.\lambda y.(y x)) y$ in **DRLAMBDA** ein. Sie werden sehen, dass dort genau das passiert, was eben beschrieben wurde; die neue Variable hat lediglich einen anderen Namen als w .

Aufgabe 12.6. Bauen Sie den Code aus Aufgabe 12.2 so um, dass β -Reduktionsschritte nach der endgültigen Definition vorgenommen werden. Zum Ausprobieren können Sie zu `test-cases` diese beiden Fälle hinzufügen:

```
(((((lambda (x) (lambda (y) (y x))) y) x) (x y))
 ((lambda (x) (lambda (y) (y (x (lambda (y) y))))) y)
 (lambda (y1) (y1 (y (lambda (y) y)))))
```

(Hinweis: Es reicht, die Funktion `new-subst` durch eine andere zu ersetzen.)

Sie werden eine Funktion zum Erzeugen neuer Symbole brauchen. Da wir die dafür nötigen SCHEME-Funktionen noch nicht besprochen haben, „schenke“ ich Ihnen diese Funktion:¹⁰

```
(define fresh-var
  (lambda (old forbidden counter)
    (let ((candidate (string->symbol
```

¹⁰Falls Sie neugierig sind: `format` macht im Prinzip dasselbe wie `printf` (siehe Aufgabe 11.5), erzeugt aber einen `String`, statt in die `Standardausgabe` zu schreiben. `string->symbol` macht genau das, was der Name der Funktion bereits andeutet.

```
(format "~A~A" old counter))))
(cond ((not (member? candidate forbidden)) candidate)
      (else (fresh-var old forbidden (+ 1 counter))))))
```

`fresh-var` wird aufgerufen mit einem Symbol `old`, einer Liste `forbidden` und einer Zahl `counter`. `old` ist die Variable, für die ein Ersatz gesucht wird und die gewissermaßen als „Muster“ dient, `forbidden` ist eine Liste von Namen, die vermieden werden müssen, und `counter` ist ein Zähler, der einfach 1 sein sollte.¹¹ Der Aufruf `(fresh-var 'x '(x1 y) 1)` liefert z. B. das Symbol `x2`.

Die Sache mit der „prinzipiellen Austauschbarkeit“ von Variablen können wir nun ebenfalls präzisieren:

Wir definieren rekursiv, wann zwei λ -Ausdrücke α -äquivalent sind:

- (α_1) Identische Ausdrücke sind α -äquivalent.
- (α_2) Zwei Anwendungen $(A_1 B_1)$ und $(A_2 B_2)$ sind α -äquivalent, wenn sowohl A_1 und A_2 als auch B_1 und B_2 α -äquivalent sind.
- (α_3) Zwei Abstraktionen $(\lambda x_1. A_1)$ und $(\lambda x_2. A_2)$ sind α -äquivalent, wenn x_2 nicht frei in A_1 ist und A_2 und $A_1[x_1 \leftarrow x_2]$ α -äquivalent sind.

Weiter oben hätten wir statt w z. B. auch z nehmen können. Dann hätte sich $\lambda z.(z y)$ statt $\lambda w.(w y)$ ergeben. Sind die beiden Ausdrücke α -äquivalent? Ja, denn w ist nicht frei in $(z y)$ und $(w y)$ und $(z y)[z \leftarrow w]$ sind nach (α_1) α -äquivalent.¹²

Informell kann man also sagen, dass zwei Ausdrücke α -äquivalent sind, wenn sie sich höchstens durch Umbenennung gebundener Variablen unterscheiden.

★**Aufgabe 12.7.** Um das Konzept der α -Äquivalenz richtig zu verstehen, ist es sicher hilfreich, es als Prädikat für die λ -Ausdrücke aus Aufgabe 10.2 zu programmieren.

```
> (alpha-equiv? '(lambda (x) (x y)) '(lambda (a) (a y)))
#t
> (alpha-equiv? '(lambda (x) (x y)) '(lambda (y) (y y)))
#f
> (alpha-equiv? '(lambda (x) (lambda (x) x))
                 '(lambda (y) (lambda (x) x)))
#t
> (alpha-equiv? '(lambda (x) (lambda (y) x))
                 '(lambda (y) (lambda (x) y)))
#t
```

(Hinweis: Ich würde in diesem Fall nicht „wortwörtlich“ die Definition umsetzen, sondern folgendermaßen vorgehen: Ob zwei Teilausdrücke α -äquivalent sind, hängt davon ab, in welchem „Einflussbereich“ gebundener Variablen sie sich befinden.

¹¹ Den Zähler hätte man natürlich auch wie z. B. bei `prime?-helper` im letzten Kapitel durch eine Hilfsfunktion verbergen können.

¹² Falls Sie ganz genau aufgepasst haben: Die *Definition* ist nicht symmetrisch, die *Relation* aber schon. Mit anderen Worten: Wenn man die Rollen der beiden Ausdrücke vertauscht, klappt es auch.

Das Symbol x kann durchaus α -äquivalent zum Symbol a sein, wie man am ersten Beispiel sieht. Ihre Funktion sollte also quasi „Buch führen“ über die existierenden Bindungen.)

Wir sahen bereits im ersten Beispiel in Abschnitt 11, in dem es mehr als einen Reduktionsschritt gab, dass man in so einem Fall vor der Entscheidung steht, welchen Redex man zuerst reduziert. Außerdem demonstriert Ω (siehe Aufgabe 11.4), dass es λ -Ausdrücke gibt, bei denen man niemals alle Redexe entfernen kann. Daher sind ein paar allgemeine Anmerkungen angebracht.

- Wenn man durch Reduktion, also durch eine endliche Folge von Reduktionsschritten, einen λ -Ausdruck in eine Form bringen kann, in der alle Redexe verschwunden sind, so nennt man das eine β -**Normalform** dieses λ -Ausdrucks.
- So eine β -Normalform ist bis auf α -Äquivalenz eindeutig bestimmt.¹³
- Wenn für einen λ -Ausdruck eine β -Normalform existiert, dann wird sie durch β -Reduktion in Standardreihenfolge erreicht.
- Am Beispiel $(\lambda x.y)\ \Omega$ sieht man, dass es *nicht* egal ist, in welcher Reihenfolge man vorgeht: Man kann einerseits beliebig oft Ω reduzieren, ohne dass sich etwas ändert, man kann aber andererseits auch zuerst den äußersten Redex reduzieren und landet sofort bei der β -Normalform y .
- Dieses Beispiel demonstriert ein allgemeines Prinzip: Gibt es eine β -Normalform, so kann man diese immer erreichen, wenn man eine „falsche“ Folge von Reduktionsschritten abbricht und anschließend „richtig“ fortfährt.

Wir verzichten darauf, das zu beweisen, und glauben es einfach. Es sei jedoch angemerkt, dass die Beweise keinesfalls trivial sind. Siehe dazu insbesondere den [Satz von Church-Rosser](#), aus dem einige der obigen Aussagen folgen.

Aufgabe 12.8. Nutzen Sie [DRLAMBDA](#), um sich mit β -Reduktion vertraut zu machen und den Umgang mit λ -Ausdrücken zu üben. Sie sollten in der Lage sein, λ -Ausdrücke fehlerfrei einzugeben und diese auch ohne Computerhilfe zu normalisieren.

Das Tool reduziert immer in Standardreihenfolge und findet daher immer die β -Normalform, wenn sie existiert. Wie bereits erwähnt kann es auf Wunsch auch die einzelnen Reduktionsschritte und nicht nur das Endergebnis anzeigen.

13. Church-Kodierung

Am [Anfang des Kapitels](#) habe ich geschrieben, dass der Lambda-Kalkül [turingvollständig](#) ist, dass man mit ihm also im Prinzip jedes beliebige Computerprogramm simulieren kann. Obwohl wir inzwischen die Funktionsweise des Kalküls kennen, dürfte bisher noch völlig unklar sein, wie man mit so einem scheinbar primitiven System komplexe Berechnungen durchführen kann.

¹³Das Beispiel aus Aufgabe 12.5 hat z. B. die im Text vor der Aufgabe erwähnte Normalform $\lambda w.(w\ y)$, aber auch die vom Programm angezeigte $\lambda y_1.(y_1\ y)$.

Wir fangen mit einer sehr cleveren Art und Weise an, im Lambda-Kalkül natürliche Zahlen darzustellen. Dafür schauen wir uns zunächst an, wie die Mathematik diese Zahlen charakterisiert. Das geschieht dort mithilfe der sogenannten **Dedekind-Peano-Axiome**:

- (DP1) 0 ist eine natürliche Zahl.
- (DP2) Jede natürliche Zahl n hat einen **Nachfolger**, der ebenfalls eine natürliche Zahl ist und für den man S_n schreibt.¹⁴
- (DP3) 0 ist *nicht* der Nachfolger einer natürlichen Zahl.
- (DP4) Haben zwei natürliche Zahlen denselben Nachfolger, so sind sie gleich.¹⁵
- (DP5) Hat 0 eine bestimmte Eigenschaft und hat mit jeder natürlichen Zahl, die diese Eigenschaft hat, auch ihr Nachfolger diese Eigenschaft, dann haben *alle* natürlichen Zahlen diese Eigenschaft.

Zumindest bei den ersten vier Axiomen wird jemand, der so etwas zum ersten Mal sieht, wahrscheinlich mit den Achseln zucken und sich fragen, warum man solche Selbstverständlichkeiten aufschreiben und mit einem Namen versehen müsse. Das liegt daran, dass wir alle eine *intuitive* Vorstellung von den natürlichen Zahlen haben, die im Prinzip auf „0, 1, 2, 3 und dann immer so weiter“ hinausläuft. Die Axiome sind dafür da, die natürlichen Zahlen *ohne* Rückgriff auf die Intuition eindeutig zu charakterisieren, indem man ihre wesentlichen Eigenschaften herausarbeitet, aus denen alles Weitere folgt.¹⁶

Was wir aus diesen Axiomen mitnehmen sollten, ist das Folgende:

- Wenn man eine Aussage zu den natürlichen Zahlen begründen will, kann man sie im Prinzip immer auf diese Axiome zurückführen.
- Es gibt nur zwei Möglichkeiten, eine natürliche Zahl durch einen geschlossenen **Term** darzustellen: entweder durch den Term 0 nach (DP1) oder durch Terme der Form S_0 , SS_0 , SSS_0 und so weiter nach (DP2).
- Wenn wir natürliche Zahlen anders schreiben, so beruht das auf Konventionen wie dem **Dezimalsystem**. Formal sollten wir jedoch ein Symbol wie 4 als Abkürzung für $SSSS_0$ betrachten.¹⁷
- Axiom (DP5) beschreibt das, was man als **vollständige Induktion** bezeichnet. Dieses Axiom ist die Grundlage dafür, dass wir in Abschnitt 9 die gesamte Arithmetik mit ganz wenigen Grundbausteinen aufbauen konnten. Die dort verwendeten Funktionen `zero?` und `add1` entsprachen (DP1) und (DP2).

¹⁴Das S steht für *successor*. Manchmal verwendet man auch die Schreibweise n' .

¹⁵Die umgekehrte Aussage gilt auch: Sind zwar natürliche Zahlen gleich, so haben sie denselben Nachfolger. Das wird allerdings nicht explizit gesagt, da es eine direkte Folgerung aus den Eigenschaften der **Gleichheit** und der Tatsache ist, dass man *Nachfolger* als eine **Funktion** betrachtet. Mit anderen Worten: Der Nachfolger einer Zahl ist eindeutig bestimmt.

¹⁶Ob die Charakterisierung tatsächlich *eindeutig* ist, hängt davon ab, wie genau der etwas vage Begriff der *Eigenschaft* in (DP5) formalisiert wird. Das spielt für uns aber keine Rolle.

¹⁷Und wir hätten stattdessen auch das Symbol IV der **römischen Zahlschrift** verwenden können oder 100 im **Dualsystem**.

Wir interpretieren im Folgenden eine Zeichenkette wie $\lambda x y.A$ als Abkürzung für $\lambda x.\lambda y.A$. Analog interpretieren wir auch Zeichenketten mit mehr als zwei Variablen zwischen λ und dem Punkt als „mehrstellige“ Abstraktionen im Sinne von [Currying](#).

Aufgabe 13.1. Welche der Zeichenketten aus Aufgabe 10.1, die eigentlich keine λ -Ausdrücke sind, ist nach dieser neuen Konvention nun eine erlaubte Abkürzung?

Aufgabe 13.2. Geben Sie die β -Normalformen der folgenden λ -Ausdrücke an.

$$(\lambda x y.x) v w \quad (\lambda x y.y x) v w \quad (\lambda x y z.y x) v w$$

Aufgabe 13.3. [DRLAMBDA](#) akzeptiert diese Form der Abkürzung übrigens auch. Probieren Sie es aus!

Church-Numerale¹⁸ sind λ -Ausdrücke, die nach dem folgenden Muster rekursiv definiert werden:

- (i) $\lambda f n.n$ ist ein Church-Numeral.
- (ii) Ist $\lambda f n.A$ ein Church-Numeral, dann ist auch $\lambda f n.f A$ ein Church-Numeral. (Dabei ist A als ein vollständiger λ -Ausdruck inklusive der Klammern zu verstehen.)

Nach dieser Definition sind

$$\lambda f n.n, \quad \lambda f n.f n, \quad \lambda f n.f(f n), \quad \lambda f n.f(f(f n)) \quad (13.1)$$

und so weiter Church-Numerale.

Aufgabe 13.4. Geben Sie die β -Normalformen der folgenden λ -Ausdrücke an.

$$(\lambda f n.n) s z \quad (\lambda f n.f n) s z \quad (\lambda f n.f(f n)) s z \quad (\lambda f n.f f n) s z$$

0 1 2 etc.

Für die Church-Numerale führen wir als abkürzende Schreibweise natürliche Zahlen ein. Dabei soll die Zahl n angeben, wie oft die Variable f hinter dem Punkt steht. Die Numerale in (13.1) sind also 0, 1, 2 und 3.

Außerdem führen wir für die [Identität](#) $\lambda x.x$ die Abkürzung I ein.

Wenn wir Abkürzungen für λ -Ausdrücke verwenden und diese in andere λ -Ausdrücke einsetzen, so ist das immer so gemeint, dass die Abkürzungen für die vollständigen Ausdrücke *mit allen Klammern* stehen. 1 $x y$ steht also beispielsweise für $(\lambda f n.f n) x y$ und *nicht* für $\lambda f n.f n x y$.

Aufgabe 13.5. Geben Sie die β -Normalformen von $(\lambda f n.f n) x y$ und $\lambda f n.f n x y$ an.

¹⁸Numeral(*e*) ist in der Sprachwissenschaft ein anderer Ausdruck für ein [Zahlwort](#).

Wie man an Aufgabe 13.4 erkennen kann, bedeutet ein Ausdruck wie $(2\ s\ z)$ also, dass s zweimal auf z angewendet wird – wodurch man $s\ (s\ z)$ erhält. $(4\ s\ z)$ würde viermalige Anwendung mit dem Resultat $s\ (s\ (s\ (s\ z)))$ bedeuten und so weiter. Insbesondere bedeutet $(0\ s\ z)$, dass s gar nicht angewendet wird und man einfach z erhält. Betrachtet man es durch die Brille der Informatik, so erhält man durch die Church-Numerale eine einfache Form einer Schleife.

Setzt man sich stattdessen die Brille der Mathematik auf, kann man diese Numerale auch folgendermaßen interpretieren: Ist s die Nachfolgerfunktion und z die Zahl null, so ist die Anwendung des Church-Numerals n der Term, der gemäß der Dedekind-Peano-Axiome die natürliche Zahl n beschreibt.

Aufgabe 13.6. [DRLAMBDA](#) akzeptiert ebenfalls natürliche Zahlen als Abkürzungen für Church-Numerale. Auch damit sollten Sie zu Eingewöhnung ein bisschen herumspielen.

Aufgabe 13.7. Wir haben gesehen, dass $3\ f\ x$ die dreimalige Anwendung von f auf x bedeutet. Was ist mit $3\ f$ ohne x ?

Den λ -Ausdruck $\lambda x. \lambda f n. f\ (x\ f\ n)$ bezeichnen wir im Folgenden als SUCC.¹⁹

SUCC

SUCC steht für *successor* (also Nachfolger) und der Sinn dieser Bezeichnung erschließt sich, wenn man SUCC auf ein Church-Numeral anwendet. SUCC 3 steht beispielsweise für $(\lambda x. (\lambda f n. f\ (x\ f\ n)))\ 3$ und ein Reduktionsschritt macht daraus $\lambda f n. f\ (3\ f\ n)$. $3\ f\ n$ wiederum wird, wie wir gerade gesehen haben, zu $f\ (f\ (f\ n))$. Damit erhalten wir insgesamt $\lambda f n. f\ (f\ (f\ (f\ n)))$, also das Church-Numeral 4. Offensichtlich „berechnet“ SUCC also den Nachfolger eines Church-Numerals.

Aufgabe 13.8. In [DRLAMBDA](#) kann man Abkürzungen wie SUCC oder I im Definitionsfenster eingeben. Schauen Sie dazu in der Dokumentation nach. Wenn Sie Abkürzungen verwenden, werden außerdem in der Ausgabe auch Church-Numerale angezeigt. Bringen Sie das System dazu, dass bei der Eingabe `Succ 5` das Ergebnis 6 angezeigt wird. Benutzen Sie außerdem `-->` im Interaktionsfenster, um sich die Reduktion schrittweise und ohne Abkürzungen anzeigen zu lassen.

Aufgabe 13.9. Was kommt heraus, wenn man die Nachfolgerfunktion dreimal auf das Numeral 7 anwendet, wenn man also den Nachfolger des Nachfolgers des Nachfolgers von 7 berechnet? Und was kommt heraus, wenn man die Nachfolgerfunktion siebenmal auf 3 anwendet? Kommt Ihnen das bekannt vor?

Aufgabe 13.9 sollte Sie an unsere Definition der Addition von Seite 32 erinnern haben. Das dreimalige Anwenden von `Succ` auf 7 entspricht offenbar der Addition $3 + 7$. Und für das dreimalige Anwenden einer Abstraktion auf ein Argument haben

¹⁹Man hätte das auch kürzer als $\lambda x\ f\ n. f\ (x\ f\ n)$ schreiben können, aber durch die etwas ausführlichere Schreibweise wird deutlicher, dass SUCC als Abstraktion mit einem Argument zu betrachten ist, deren Rumpf eine Abstraktion mit zwei Argumenten ist.

wir doch schon ein Kürzel: das Church-Numeral 3. $3 + 7$ lässt sich also im Lambda-Kalkül (mit den entsprechenden Abkürzungen) ganz einfach so darstellen:

3 SUCC 7

PLUS **Aufgabe 13.10.** Definieren Sie einen λ -Ausdruck PLUS, so dass beispielsweise 42 die β -Normalform von PLUS 30 12 ist.

TIMES **Aufgabe 13.11.** Nach der Addition kam in Abschnitt 9 die Multiplikation. Definieren Sie mithilfe der Lösung von Aufgabe 13.10 einen λ -Ausdruck TIMES, so dass beispielsweise 42 die β -Normalform von TIMES 6 7 ist.

Aufgabe 13.12. Probieren Sie Addition und Multiplikation in **DRLAMBDA** aus. Lassen Sie sich auch die einzelnen Reduktionsschritte zeigen.

Aufgabe 13.13. Man hätte Multiplikation auch durch $\lambda x y. \lambda a. x (y a)$ ($y a$) definieren können. Machen Sie sich anhand von Beispielen klar, dass und warum das funktioniert.

Wie im Abschnitt 6 wenden wir uns nun **bedingten Anweisungen und Verzweigungen** zu. In SCHEME hatten wir dafür aber zumindest schon ein paar vorhandene Prädikate wie `eqv?` oder `zero?`, während wir im Lambda-Kalkül nicht einmal **Wahrheitswerte** haben. Damit fangen wir an.

T F Die λ -Ausdrücke $\lambda x y. x$ und $\lambda x y. y$ nennen wir T bzw. F (was natürlich für *true* und *false* stehen soll).

Beides sind also zweistellige Abstraktionen, wobei T das erste Argument auswählt und F das zweite. Diese Festlegung ist zunächst einmal völlig willkürlich wie die übliche Zuordnung von 0 und 1 zu falsch und wahr. Es wird sich jedoch gleich zeigen, dass sie sehr praktisch ist.

Aufgabe 13.14. Kommt Ihnen F bekannt vor?

AND Den λ -Ausdruck $\lambda x y. x y F$ bezeichnen wir als AND.

Aufgabe 13.15. Vervollständigen Sie die folgenden Tabelle.

	λ -Ausdruck	β -Normalform
AND T T	$(\lambda x y. x y (\lambda x y. y)) (\lambda x y. x)$	$(\lambda x y. x) T$
AND T F		
AND F T		
AND F F		

Das ist so gemeint, dass Sie vom linken Ausdruck zunächst die ausführliche Form hinschreiben, ihn dann auf Normalform bringen und schließlich die Abkürzung der Normalform hinschreiben, wenn es eine gibt.

Aufgabe 13.16. Bevor Sie weiterlesen: Haben Sie eine Idee, wie man auf ähnliche Weise **OR** realisieren könnte?

Wie man an Aufgabe 13.15 nachvollziehen konnte, realisiert der λ -Ausdruck **AND** die **Konjunktion** für unsere Wahrheitswerte T und F. Die anderen beiden wichtigen **Junktoren** kann man ähnlich einfach aufbauen.

Die λ -Ausdrücke $\lambda x y.x T y$ und $\lambda x.x F T$ bezeichnen wir als **OR** bzw. **NOT**.

OR NOT

Aufgabe 13.17. Überzeugen Sie sich durch Tabellen wie in Aufgabe 13.15, dass **OR** und **NOT** tatsächlich **Disjunktion** und **Negation** realisieren.

Aufgabe 13.18. Man hätte **NOT** auch anders bauen können – als „Vorrichtung“, die einfach die beiden Argumente für T bzw. F vertauscht. Haben Sie eine Idee, wie man das machen könnte?

Wir können nun etwas definieren, was ähnlich wie **zero?** arbeitet.

Den λ -Ausdruck $\lambda x.x F NOT F$ werden wir **Z** nennen.

Z

Was macht **Z**? Wenden wir diesen Ausdruck auf das Numeral 0 an, so erhalten wir

$$Z\ 0 = (\lambda x.x F NOT F)\ 0 \rightarrow (0\ F\ NOT)\ F \rightarrow NOT\ F \rightarrow T.$$

Dabei steht $\rightarrow$ für einen oder mehrere Reduktionsschritte, nicht unbedingt in Standardreihenfolge, und das letzte Klammerpaar befindet sich dort nur, um die Wirkung von 0 (die ja identisch mit der von F ist) deutlich zu machen. Nehmen wir nun ein anderes Numeral wie z. B. 2, so ergibt sich.

$$Z\ 2 \rightarrow (2\ F\ NOT)\ F \rightarrow (F\ (F\ NOT))\ F \rightarrow (F\ I)\ F \rightarrow I\ F \rightarrow F.$$

Die letzten Schritte beruhen darauf, dass $F\ A$ für jeden λ -Ausdruck A immer die Identität ist, siehe Aufgabe 13.7. **Z** wird also zu T, wenn das Argument das Numeral 0 ist, und bei allen anderen Numeralen N ist F die Normalform von $Z\ N$.

In einer Sprache wie **SCHEME** würde nun noch ein Operator wie **if** fehlen. Erstaunlicherweise sind T und F so geschickt definiert worden, dass so etwas im Lambda-Kalkül gar nicht nötig ist, wie die folgende Aufgabe zeigt.

Aufgabe 13.19. Überlegen Sie sich, wie man die **SCHEME**-Funktion

```
(lambda (x) (if (zero? x) 2 3))
```

als Abstraktion im Lambda-Kalkül umsetzen könnte. Schauen Sie sich dafür noch einmal an, wie T und F definiert wurden.

★**Aufgabe 13.20.** Man könnte in **SCHEME** die obigen Wahrheitswerte durch

```
(define true (lambda (x y) (x)))
(define false (lambda (x y) (y)))
```

simulieren und dazu eine Funktion wie die folgende definieren:

```
(define Z (lambda (n) (if (zero? n) true false)))
```

Wie könnte man dann etwas wie

```
(if (zero? n) (do-this (* n n)) (do-that (+ n 3)))
```

realisieren, wenn man `if` und `zero?` nicht verwenden darf, sondern nur `Z`, `true`, `false` und `lambda`? (Natürlich darf nur eine der beiden Funktionen `do-this` oder `do-that` aufgerufen werden, siehe Aufgabe 6.12.)

Das große Finale dieses Abschnitts wird eine rekursive „Funktion“ im Stile von Abschnitt 9 sein. Dazu brauchen wir jedoch noch ein Pendant zu `sub1` für den Vorgänger eines Numerals, das man in diesem Zusammenhang typischerweise `PRED` nennt. Wir definieren zuerst die nötigen Ausdrücke. Danach werden Sie dann erklärt.

Wir führen die folgenden Abkürzungen ein:

- | | |
|--------|--|
| PAIR | – PAIR für $\lambda a b. \lambda z. z a b$ |
| Φ | – Φ für $\lambda p. \text{PAIR} (\text{SUCC } (p \text{ T})) (p \text{ T})$ |
| PRED | – PRED für $\lambda n. (n \Phi (\text{PAIR } 0 \ 0)) F$ |

- PAIR funktioniert im Prinzip wie `cons`: Sind A und B λ -Ausdrücke, dann ist `PAIR A B` so etwas wie ein **geordnetes Paar** und man kommt an die linke Komponente, indem man das Paar auf `T` anwendet: `(PAIR A B) T`. Analog kommt man mit `F` an die rechte Komponente.
- Φ erwartet als Argument ein Paar p aus Numeralen. Der „Rückgabewert“ ist ein Paar mit den Komponenten $n + 1$ und n , wenn n die erste Komponente von p ist. (Die zweite wird ignoriert.)
- Für unsere Zwecke ist relevant, was herauskommt, wenn man Φ mehrfach auf das Paar `PAIR 0 0` anwendet:

$$(0, 0) \rightarrow (1, 0) \rightarrow (2, 1) \rightarrow (3, 2) \rightarrow (4, 3) \rightarrow \dots$$

(Hier steht (a, b) als Abkürzung für $\lambda z. z a b$.) Nach n Anwendungen von Φ auf $(0, 0)$ erhält man also $(n, n - 1)$.

- Und genau das nutzt die Abstraktion `PRED` aus, die als Argument ein Numeral n erwartet. Φ wird n -mal auf $(0, 0)$ angewendet und vom Ergebnis wird mit `F` die zweite Komponente extrahiert. Diese zweite Komponente ist $n - 1$. Außer für den Fall $n = 0$, dann ist die zweite Komponente 0. `PRED` arbeitet also wie versprochen wie `sub1`.

Wenn hier übrigens steht, dass eine Abstraktion eine bestimmte Sorte Argument „erwartet“, dann ist damit nicht gemeint, dass es wie in SCHEME eine Fehlermeldung gibt, wenn das Argument den falschen „Typ“ hat. Im (klassischen) Lambda-Kalkül gibt es gar keine Typen. Man kann *jede* Abstraktion auf *jeden* λ -Ausdruck anwenden. Die Formulierung ist lediglich so gemeint, dass die obigen Ausdrücke sich bei bestimmten Argumenten (z. B. bei Numeralen) auf die gewünschte Art verhalten.

Aufgabe 13.21. Geben Sie eine Abstraktion MINUS mit zwei Argumenten an, die Subtraktion wie in Aufgabe 9.7 „berechnet“. MINUS 7 4 soll also die Normalform 3 haben und MINUS 4 7 die Normalform 0. (Hinweis: Orientieren Sie sich am Beispiel der Addition oben.)

MINUS

Aufgabe 13.22. Geben Sie eine Abstraktion an, die zu T wird, wenn Ihre beiden Argumente gleich sind, und sonst zu F. Sie muss nur für Argumente korrekt funktionieren, die Church-Numerale sind. (Hinweis: Verwenden Sie MINUS aus Aufgabe 13.21.)

Aufgabe 13.23. Es gibt eine konzisere Art, PRED auszudrücken,²⁰ und zwar so:

$$\lambda m. \lambda f n. m (\lambda g. \lambda h. h (g f)) (\lambda u. n) I$$

Können Sie erklären, warum das funktioniert?

Der krönende Abschluss soll wie schon erwähnt **Rekursion** sein. Es scheint jedoch einen Haken zu geben. Die rekursiven Funktionen, die wir in SCHEME programmiert haben, hatten einen Namen, mit dem sie sich selbst aufrufen konnten. Die „Funktionen“ des Lambda-Kalküls haben jedoch keine Namen. Zum Glück gibt es auch dafür eine sehr clevere Lösung.

Der λ -Ausdruck $\lambda y. (\lambda x. y (x x)) (\lambda x. y (x x))$ wird traditionell mit Y abgekürzt und ist unter dem Namen **Y combinator** bekannt.

Y

Wendet man Y auf einen λ -Ausdruck f an, so erhält man:

$$\begin{aligned} Y f &= (\lambda y. (\lambda x. y (x x)) (\lambda x. y (x x))) f \rightarrow (\lambda x. f (x x)) (\lambda x. f (x x)) \\ &\rightarrow f ((\lambda x. f (x x)) (\lambda x. f (x x))) \end{aligned}$$

Vereinfacht gesagt (wenn man einen Reduktionsschritt unterschlägt), wird aus $Y f$ der Ausdruck $f(Y f)$. Ist f z. B. die Identität I , so führt das zu einer **Endlosschleife**, weil $f(Y f)$ wieder zu $Y f$ führt. (Probieren Sie es in **DR LAMBDA** aus!) Ist f dagegen etwa $\lambda x. z$, so wird $f(Y f)$ einfach zu z .

Man kann das jedoch auch geschickt so ausnutzen, dass f das Argument $Y f$ als Möglichkeit nutzt, sich selbst rekursiv „aufzurufen“. Insbesondere kann f auch mehr als ein Argument haben. Aus $Y f A$ würde beispielsweise $f(Y f) A$ werden und f könnte anhand von A entscheiden, ob die Rekursion fortgesetzt oder beendet wird.

²⁰Ich glaube allerdings, dass die obige leichter zu verstehen ist.

Hier ein Beispiel, das die Summe der Zahlen von 1 bis n berechnet (ohne die [gaußsche Summenformel](#) zu verwenden):

$$\text{SUM} = \lambda r\ n. (Z\ n)\ 0\ (\text{PLUS}\ n\ (r\ (\text{PRED}\ n)))$$

So eine „rekursive Funktion“ hat typischerweise mindestens zwei Argumente, wobei das erste (das hier r heißt) für die Funktion selbst steht (die über den „Umweg“ Y aufgerufen wird). Der Rumpf der Abstraktion verwendet die Technik aus Aufgabe 13.19, um die Fälle $n = 0$ und $n > 0$ zu unterscheiden.

Aufgerufen wird SUM z. B. durch $Y\ \text{SUM}\ 5$, um die Summe der Zahlen von 1 bis 5 zu berechnen. (Wenn Sie das in [DRLAMBDA](#) ausprobieren, werden Sie bemerken, dass das System bei Werten über 10 recht schnell an seine Grenzen stößt. Aber es geht hier nur darum, was prinzipiell möglich ist.)

Aufgabe 13.24. Geben Sie einen λ -Ausdruck an, der rekursiv die [Fakultät](#) berechnet. (Siehe Aufgabe 9.14.)

Aufgabe 13.25. Geben Sie einen λ -Ausdruck an, der wie in Aufgabe 9.8 den ganzzahligen Quotienten zweier natürlicher Zahlen berechnet.

Aufgabe 13.26. Würde man den *Y combinator* wortwörtlich in SCHEME übersetzen, dann würde es so aussehen:

```
(define Y
  (lambda (y)
    ((lambda (x) (y (x x)))
     (lambda (x) (y (x x))))))
```

Warum klappt das nicht?

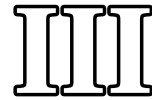
Aufgabe 13.27. Wenn wir wie auf Seite 17 *Currying* verwenden und damit die Konvention, dass jede Funktion nur genau ein Argument erwartet, dann kann man den Effekt des *Y combinators* folgendermaßen simulieren:

```
(define Y
  (lambda (f)
    (lambda (x)
      ((f (Y f)) x))))
```

Wie würde die dazu passende Funktion aussehen, die dasselbe wie der λ -Ausdruck SUM von oben macht?

★Aufgabe 13.28. In Aufgabe 13.27 wurde in gewissem Sinne „geschummelt“, weil die Funktion Y sich selbst aufruft und dabei verwendet, dass SCHEME [Rekursion](#) bereits beherrscht. Der „Witz“ des *Y combinators* ist aber eigentlich, dass er Rekursion erst ermöglicht. Schreiben Sie in SCHEME eine Version von Y , die sich wie die in Aufgabe 13.27 verhält, dabei aber keine Rekursion verwendet. (Hinweis: Arbeiten Sie mit verzögerter Ausführung wie in der Lösung von Aufgabe 13.20.)

Das war nur ein erster Einblick in das, was man mit dem Lambda-Kalkül anstellen kann. Weitere Beispiele finden Sie im [englischen Wikipedia-Artikel zur Church-Kodierung](#). Wie man aus solchen primitiven Bausteinen beliebig komplexe Algorithmen erstellen kann, wird im Modul *Theoretische Informatik* besprochen. Siehe dazu insbesondere das Dokument *Compile C Code to Turing Machines* mit den zugehörigen Programmen.



Typentheorie

In diesem Kapitel beschäftigen wir uns mit den Basics der [Typentheorie](#). Die Grundidee für solche Theorien geht auf [Bertrand Russell](#) zurück, der damit auf Probleme in der [Mengenlehre](#) reagierte. Moderne Typentheorien spielen inzwischen eine wichtige Rolle in [Beweisassistenten](#) wie [LEAN](#), die nicht nur in der Mathematik, sondern z. B. auch für die [Verifikation](#) von Schaltkreisen und Prozessoren verwendet werden oder für die Überprüfung von Autorisierungsrichtlinien bei [Amazon Web Services](#). (Mehr dazu in Kapitel [IV](#).)

Wir verwenden die Sprache [PIE](#), die man in [DRRACKET](#) installieren kann.¹ Sie wurde ursprünglich für das Lehrbuch *The Little Typer* entwickelt, das sich deswegen auch als begleitender Text für die folgenden Abschnitte eignet. Die Sprache wird dadurch aktiviert, dass in der ersten Zeile des Definitionsfensters `#lang pie` steht.

14. Ausdrücke, Typen, Konstruktoren

Wenn Sie im Interaktionsfenster `42` eingeben, erhalten Sie

```
(the Nat 42)
```

als Antwort. Das ist anders als in `SCHEME`, wo die Antwort einfach `42` gewesen wäre. Hier wird zusätzlich der *Typ* der Antwort ausgegeben. (`Nat` steht für *natürliche Zahl*.)

Wenn Sie `(add1 41)` eingeben, erhalten Sie dieselbe Antwort wie eben. Die Vermutung liegt nahe, dass `add1` wie in `SCHEME` eine Funktion ist, die 1 zu ihrem Argument addiert. Das stimmt aber nicht! Den Unterschied sehen Sie, wenn Sie in `SCHEME` und in `PIE` nur `add1` eingeben. In `SCHEME` erhalten Sie

```
#<procedure:add1>
```

wie erwartet. In `PIE` wird Ihnen hingegen mitgeteilt, dass es sich nicht um einen erlaubten Ausdruck handelt.

¹Über `File` → `Install Package` im Menü.

Allerdings wird – anders als in SCHEME – die Eingabe `zero` (ohne **Hochstrich**) akzeptiert und führt wie die Eingabe `0` zur folgenden Ausgabe:

```
(the Nat 0)
```

Was geht hier vor?

Alle erlaubten Ausdrücke (*expressions*) in PIE haben einen **Typ**. Zu einem Typ gehören **Konstruktoren**, mit denen man Ausdrücke dieses Typs erstellen kann. Die Namen von Typen fangen in PIE mit einem Großbuchstaben an.

Im den folgenden Abschnitten werden wir die Aussage, dass jeder Ausdruck einen Typ hat, noch mehrfach präzisieren müssen. Als provisorische „Arbeitsgrundlage“ ist sie jedoch ganz hilfreich.

Was wir bisher gesehen haben, sind natürliche Zahlen.

Nat
zero
add1

Der Typ `Nat` hat zwei Konstruktoren. Einer heißt `zero` und er erzeugt den Ausdruck, der für die natürliche Zahl 0 (null) steht. Der andere Konstruktor heißt `add1`. Ist n ein Ausdruck vom Typ `Nat`, dann ist `(add1 n)` ebenfalls ein Ausdruck vom Typ `Nat` und steht für den Nachfolger von n , also für $n + 1$. `Nat`-Ausdrücke werden wir auch **natürliche Zahlen** nennen.

Das ist offenbar eine **rekursive** Beschreibung der natürlichen Zahlen analog zu den **Dedekind-Peano-Axiomen**: Mit `zero` ist auch `(add1 zero)` ein `Nat`-Ausdruck, damit auch `(add1 (add1 zero))` und so weiter.

Dass wir statt `(add1 (add1 zero))` einfach `2` eingeben können und dass uns nicht

```
(the Nat (add1 (add1 (add1 (add1 zero)))))
```

angezeigt wird, wenn wir `4` eingeben, ist "**syntaktischer Zucker**" von PIE. Aus der eingegebenen `2` wird intern tatsächlich `(add1 (add1 zero))` und so ein Ausdruck wird erst für die Ausgabe wieder in die besser lesbare Darstellung `2` zurückverwandelt. (Dass das wirklich geschieht, merkt man, wenn man größere Zahlen wie `10 000` oder `100 000` eingibt. Probieren Sie es aus!) Man sollte sich jedoch klarmachen, dass die interne Darstellung der „wirkliche“ Ausdruck ist.

Aufgabe 14.1. Wie sieht die interne Darstellung der natürlichen Zahl 5 aus?

Aufgabe 14.2. Wie sieht die interne Darstellung der natürlichen Zahl 0 aus?

Ein zweiter grundlegender Typ neben `Nat` ist `Atom`:

Atom

Ein **Hochstrich** gefolgt von ein oder mehreren Buchstaben oder **Bindestrichen** ist ein Konstruktor für einen Ausdruck vom Typ `Atom`, das dieselbe Form wie sein Konstruktor hat.

Atom-Ausdrücke werden wir auch **Atome** nennen.

'elphi ist also beispielsweise der Konstruktor für das Atom 'elphi und '-höhö- ist der Konstruktor für das Atom '-höhö-. Während Nat nur zwei Konstruktoren hat, hat Atom somit potentiell unendlich viele Konstruktoren.

Aufgabe 14.3. Ist pups ein Atom?

Aufgabe 14.4. Sind '--, 'r2d2 und 'zero Atome?

Wie in SCHEME kann man statt 'haw auch (quote haw) eingeben. Aber quote hat in PIE eine komplett andere Funktion als in SCHEME! In SCHEME verhindert quote die *Evaluation* eines Ausdrucks. In PIE werden Ausdrücke *nie* in diesem Sinne evaluiert. Sie werden höchstens in andere, identische Ausdrücke umgewandelt – wobei noch zu klären sein wird, was genau mit *identisch* gemeint ist. Es gibt also nicht Ausdrücke und ihre Werte, sondern – wie im [Lambda-Kalkül](#) – nur Ausdrücke.² In PIE wird quote ausschließlich als Bestandteil von Atomen verwendet.

In der einfachen Typentheorie, die durch PIE realisiert wird, kann man nur vier Arten von sogenannten **Urteilen** (*judgments*) bilden. Damit sind Aussagen über Ausdrücke gemeint. Zwei davon³ sind

- (J1) X ist ein Ausdruck vom Typ T .
- (J3) X ist derselbe Ausdruck vom Typ T wie Y .⁴

Das erste Urteil kann man mit the-Ausdrücken überprüfen, die wir schon in Antworten von PIE gesehen haben. Geben wir

```
(the Nat 42)
```

ein, so hat das denselben Effekt wie die Eingabe von 42. Der Unterschied ist lediglich, dass wir behaupten, 42 sei ein Ausdruck vom Typ Nat. Das hätte PIE in diesem Fall allerdings selbst herausgefunden. Manchmal kann das System Typen von Ausdrücken jedoch nicht selbst herleiten und wir müssen ihm auf diesem Wege helfen.

Außerdem können wir the-Ausdrücke wie gesagt verwenden, um (J1)-Urteile zu prüfen. Mittels

```
(the Atom 42)
```

erfahren wir z. B., dass 42 *kein* Atom ist.

²Wir werden trotzdem manchmal Verben wie *evaluieren* oder *auswerten* verwenden. Es sollte aber dann klar sein, dass damit nur die Umwandlung von Ausdrücken gemeint sein kann.

³Der Rest folgt im nächsten Abschnitt.

⁴Ich habe lange überlegt, ob ich hier *derselbe* oder *der gleiche* schreibe. Ich habe mich schließlich für *derselbe* entschieden, weil z. B. auch in der Mathematik 7 und $3 + 4$ für *dieselbe* Zahl stehen: die Sieben gibt es nur einmal. Ich gebe jedoch zu, dass man sich über die Wortwahl streiten könnte.

Für das Verständnis ist (J3) besonders wichtig. In einer Typentheorie ergibt die Frage, ob zwei Ausdrücke identisch sind, nur Sinn, wenn man sie zusammen mit einem Typ stellt. Damit die Antwort *ja* ist, müssen die beiden Ausdrücke erstens denselben Typ haben und zweitens entscheiden dann die Regeln des Typs darüber, ob Gleichheit vorliegt.

Für die beiden Typen, die wir bisher definiert haben, können wir bereits Gleichheitsregeln festhalten.

zero ist derselbe Nat-Ausdruck wie zero. Ist k derselbe Nat-Ausdruck wie n , dann ist $(\text{add1 } k)$ derselbe Nat-Ausdruck wie $(\text{add1 } n)$.
Zwei Ausdrücke, die jeweils aus einem Hochstrich gefolgt von derselben Sequenz aus Buchstaben und Bindestrichen bestehen, sind als Atom-Ausdrücke identisch.

Für Nat haben wir es offenbar wieder mit Rekursion zu tun. Beispielsweise sind zero und zero als Nat-Ausdrücke gleich und daher sind das auch (add1 zero) und (add1 zero) . Und $(\text{add1 } (\text{add1 zero}))$ und $(\text{add1 } (\text{add1 zero}))$ deswegen dann auch und so weiter. Außerdem ist z. B. 'quark dasselbe Atom wie 'quark. Das hatten Sie wohl auch erwartet ...

check-same

Das Urteil (J3) lässt sich in PIE folgendermaßen überprüfen:

```
(check-same Nat (add1 zero) 1)
(check-same Atom 'bratwurst 'bratwurst)
(check-same Atom 0 1)
```

Wenn das System nicht reagiert, entspricht das der Antwort *ja*. Anderenfalls erhalten Sie wie im dritten Beispiel eine entsprechende Fehlermeldung. Hier kann man noch einmal gut erkennen, wie (J3) gemeint ist, weil check-same zwingend die Angabe eines Typen erwartet.

Aufgabe 14.5. Ist 2 derselbe Nat-Ausdruck wie $(\text{add1 } (\text{add1 zero}))$?

Aufgabe 14.6. Ist 'quark dasselbe Atom wie 'joghurt? Ist $(\text{add1 } (\text{add1 zero}))$ derselbe Nat-Ausdruck wie 3? Begründen Sie Ihre Antwort, ohne PIE zu fragen!

$(\text{check-same } \dots)$ ist übrigens eine syntaktische Form, die *kein* Ausdruck im Sinne der Typentheorie ist, sondern eine **Deklaration** der Sprache PIE. Das gilt ebenso für claim und define, denen wir im nächsten Abschnitt begegnen werden.

Aufgabe 14.7. Welche Antwort würden Sie bei der Eingabe der Ausdrücke -3 bzw. 3.14 in PIE erwarten?

15. Urteile und Normalformen

In PIE gibt es car, cdr und cons, die eine gewisse Ähnlichkeit mit ihren Namensvettern in SCHEME haben. Man sollte aber lieber auf die Unterschiede achten!

Der Ausdruck

```
(cons 'two 2)
```

ist zwar erlaubt, PIE wird jedoch sagen, dass der Typ dieses Ausdrucks nicht ermittelt werden kann.

Ist X_1 ein Ausdruck vom Typ T_1 und X_2 ein Ausdruck vom Typ T_2 , dann ist $(\text{cons } X_1 \ X_2)$ ein Ausdruck vom Typ $(\text{Pair } T_1 \ T_2)$. Ist Y_1 derselbe T_1 -Ausdruck wie X_1 und Y_2 derselbe T_2 -Ausdruck wie X_2 , dann ist $(\text{cons } X_1 \ X_2)$ dasselbe $(\text{Pair } T_1 \ T_2)$ wie $(\text{cons } Y_1 \ Y_2)$. Wir nennen solche Ausdrücke auch T_1 - T_2 -**Paare** (oder kurz T_1 -**Paare**, falls T_2 derselbe Typ wie T_1 ist). X_1 und X_2 werden die erste bzw. zweite **Komponente** oder **Projektion** des Paares genannt.

cons

Pair

cons ist also der (einzige) Konstruktor für solche Paare. Das Paar vom Anfang des Abschnitts kann man damit als

```
(the (Pair Atom Nat) (cons 'two 2))
```

eingeben. Und Gleichheit würde man folgendermaßen prüfen:

```
(check-same (Pair Atom Nat) (cons 'two 2) (cons 'two (add1 1)))
```

Kurze Abschweifung: Wenn man anfangs herumexperimentiert, ist das leider ziemlich mühsam, weil man fast immer mithilfe von the den Typ angeben muss, um von PIE eine Antwort zu erhalten. Man fragt sich, warum es nicht „offensichtlich“ ist, dass (Pair Atom Nat) der Typ von (cons 'two 2) ist. Das wird erst [später](#) klar werden. Wir werden sehen, dass auch [andere Typen](#) möglich gewesen wären und dass PIE hier absichtlich nicht „rät“.

Benötigt man Ausdrücke öfter, so kann man zur Abkürzung Namen vergeben. Das ist immer ein zweistufiger Prozess.

Ist X ein Ausdruck vom Typ T , so kann man mithilfe der Deklarationen $(\text{claim } \langle \text{name} \rangle \ T)$ und $(\text{define } \langle \text{name} \rangle \ X)$ (in dieser Reihenfolge) den Namen $\langle \text{name} \rangle$ für X vergeben. Dieser Name kann dann in Ausdrücken als Platzhalter für X bzw. für $(\text{the } T \ X)$ verwendet werden.

claim

define

Solche Deklarationen können nur im Definitionsfenster eingegeben werden. Als Namen sind alle Zeichenketten zulässig, die auch in SCHEME als Namen erlaubt sind, wenn es sich nicht um [reservierte](#) Namen handelt.⁵ Ein einmal verwendeter Name kann in PIE später jedoch nicht mehr „umdefiniert“ werden.

Hier ein paar Beispiele für solche Definitionen:

⁵Reserviert sind beispielsweise die Namen von Konstruktoren wie `add1`, von Deklarationen wie `define` sowie von vordefinierten Typen wie `Nat`.

```
(claim four Nat)

(define four
  (add1 (add1 (add1 (add1 zero)))))

(claim my-pair (Pair Atom Nat))

(define my-pair
  (cons 'four four))

(claim my-second-pair (Pair Nat (Pair Atom Nat)))

(define my-second-pair
  (cons 17 my-pair))
```

Beachten Sie, dass es sich im letzten Beispiel um ein „verschachteltes“ Paar handelt, bei dem die zweite Komponente ebenfalls ein Paar ist. Im Interaktionsfenster kann man nun so etwas wie `my-second-pair` eingeben und sich anschauen, was PIE daraus macht. Aber wie kommt man an die Komponenten eines solchen Paares?

`car` `cdr`

Ist P ein Ausdruck vom Typ $(\text{Pair } T_1 T_2)$, dann ist $(\text{car } P)$ derselbe T_1 -Ausdruck wie die erste Komponente von P und $(\text{cdr } P)$ derselbe T_2 -Ausdruck wie die zweite.

`car` und `cdr` sind sogenannte **Eliminatoren** für Paare. Während Konstruktoren Ausdrücke eines bestimmten Typs erzeugen, machen Eliminatoren gewissermaßen das Gegenteil. Sie stellen die Information zur Verfügung, die in diesen Ausdrücken enthalten ist.

Es folgen ein paar Beispiele für die Verwendung von `car` und `cdr`, bei denen Sie sich zuerst ohne Computer überlegen sollten, was herauskommen muss:

```
(car my-pair)
(car my-second-pair)
(cdr my-second-pair)
(car (cdr my-second-pair))
```

Aufgabe 15.1. Geben Sie für die folgenden Ausdrücke jeweils an, was herauskommen muss. Machen Sie das zunächst ohne Computer.

```
(car (cons 23 42))
(cdr (cons (cons 42 23) 'quark))
(cdr (car (cons (cons 42 23) 'quark)))
(cons (car (cons 23 42)) (cdr (cons 23 42)))
```

Überlegen Sie dann, wie Sie die Ausdrücke in PIE eingeben müssen, damit Sie nicht die Meldung erhalten, dass der Typ des Ausdrucks nicht ermittelt werden kann. (In der Regel müssen alle `cons`-Ausdrücke wie beim Eingangsbeispiel dieses Abschnitts innerhalb eines `the`-Ausdrucks stehen, der PIE den richtigen Typ mitteilt.)

Wir haben „nebenbei“ in diesem Abschnitt noch etwas anderes beobachtet. `Pair` ist ein **Typ-Konstruktor**. Auch so etwas wie `(Pair Atom Nat)` ist ein Ausdruck in PIE; und zwar einer, der einen Typ repräsentiert. So ein Ausdruck wird im Wesentlichen wie andere Ausdrücke behandelt.

Aufgabe 15.2. Ist `Pair` der erste Typ-Konstruktor, den wir kennengelernt haben?

Wenn nach Abschnitt 14 jeder Ausdruck einen Typ hat, welchen Typ hat dann ein Ausdruck, der einen Typ repräsentiert? Sie können das selbst herausfinden, indem Sie im Interaktionsfenster einfach `Atom` eingeben. Die Antwort wird `(the U Atom)` sein.

`U` (für **Universum**) ist ein Typ-Konstruktor ohne Argumente. Jeder Ausdruck, der den Typ `U` hat, repräsentiert einen Typ.

`U`

Da PIE nur eine recht simple Typentheorie implementiert gilt jedoch nicht die umgekehrte Aussage. Es hat also *nicht* jeder Ausdruck, der einen Typ repräsentiert, den Typ `U`. Sonst hätte nämlich auch `U` den Typ `U` und das würde zu Problemen führen. Wir müssen also die Aussage aus Abschnitt 14 dahingehend korrigieren, dass es einige Ausdrücke gibt, die in PIE *keinen* Typ haben. Dazu gehören `U` und Ausdrücke wie `(Pair U Atom)`, in denen `U` vorkommt.⁶ Wie sich das auswirkt, sehen wir später in Aufgabe 21.8. Für das, was wir hier lernen wollen, wird das aber keine wesentlichen Konsequenzen haben.

Als **Urteile** (*judgments*) bezeichnet man grundlegende **metasprachliche**⁷ Aussagen über Ausdrücke der Typentheorie. In PIE sind die folgenden vier Arten von Urteilen möglich.

- (J1) X ist ein Ausdruck vom Typ T .
- (J2) Der Ausdruck T ist ein Typ.
- (J3) X ist derselbe Ausdruck vom Typ T wie Y .
- (J4) T_1 und T_2 sind derselbe Typ.⁸

Diese Urteile lassen sich im Prinzip ohne die Hilfe von PIE überprüfen. Bisher wissen wir allerdings noch nicht, nach welchen Regeln das abläuft. (Siehe z. B. Aufgabe 14.6.) Und wir wissen auch noch nicht, wie PIE entscheidet, wann Ausdrücke umgeformt werden.

⁶„Ausgewachsene“ Typentheorien haben eine Hierarchie von Universen. Dort hätten dann „einfache“ Typen wie `Atom` oder `(Pair Nat Nat)` den Typ U_1 , während Typen wie U_1 oder `(Pair Nat  $U_1$ )` den Typ U_2 hätten. Und es gäbe entsprechend auch den Typ U_3 und so weiter.

⁷Damit ist gemeint, dass die Urteile nicht Bestandteil der Typentheorie sind, sondern Aussagen, die die Theorie „von außen“ beschreiben. Das ändert sich jedoch in Abschnitt 20...

⁸Noch etwas genauer formuliert: T_1 und T_2 sind Ausdrücke, die denselben Typ beschreiben.

Jeder Ausdruck des Typs T hat eine eindeutig bestimmte **Normalform** bezüglich dieses Typs (die in gewissem Sinne die einfachste Art ist, den Ausdruck hinzuschreiben). Zwei Ausdrücke des Typs T sind genau dann gleich, wenn sie dieselbe Normalform bezüglich T haben.⁹

Das erinnert an die β -Normalformen der Lambda-Kalküls. Die präzise Spezifikation der jeweiligen Normalformen in PIE ist allerdings komplizierter und wird erst nach und nach klar werden. Fürs Erste begnügen wir uns damit, dass das System versucht, Ausdrücke so weit zu „vereinfachen“, dass sie nur noch aus Konstruktoren bestehen. Das geschieht typischerweise durch die Auflösung von Eliminator-Ausdrücken.

Für die Typen, die wir bereits kennen, gilt insbesondere: Besteht ein Ausdruck nur aus Konstruktoren, so ist er in Normalform. Und zwei Normalformen sind gleich, wenn sie (bis auf Leerzeichen und Zeilenumbrüche) aus denselben Zeichen in derselben Reihenfolge bestehen. Das wird ebenfalls für viele andere Typen gelten; auf Ausnahmen gehen wir wenn nötig ein. Allerdings werden wir noch sehen, dass eine Normalform auch anders aussehen kann.

Im folgenden Kasten sind die ersten drei Ausdrücke normal, die Ausdrücke dahinter nicht.¹⁰

```
'joghurt
(add1 (add1 zero))
(cons 23 42)
(car (cons 'joghurt (add1 (add1 zero))))
(cdr (cons 'joghurt (add1 (add1 zero))))
(cons (car (cons 23 42)) (cdr (cons 23 42)))
```

Der erste Ausdruck ist die Normalform des vierten, der zweite die des fünften und der dritte die des sechsten. Das heißt insbesondere, dass der erste und vierte Ausdruck *dasselbe* Atom sind, der zweite und fünfte *dieselbe* natürliche Zahl und der dritte und sechste *dasselbe* Nat-Paar.

Aufgabe 15.3. Begründen Sie mithilfe der Information über Normalformen, warum die Ausdrücke in Aufgabe 14.6 nicht identisch sind.

Aufgabe 15.4. Sind die Ausdrücke `(add1 0)` und `(cdr (cons 'zero 2))` dieselbe natürliche Zahl?

Was vorhin bereits kurz erwähnt wurde, soll nun noch einmal explizit gesagt werden: In Typentheorien sind Typen auch „nur“ Ausdrücke und werden im Großen und Ganzen genauso behandelt. Das ist ein ganz grundlegender Unterschied zu

⁹Die Begriffe *eindeutig* und *dieselbe* werden bei [Funktionen](#) etwas relativiert werden müssen.

¹⁰Dabei gilt wie üblich die Konvention, dass Ausdrücke wie 23 entsprechend ihrer internen Darstellung interpretiert werden.

„normalen“ Programmiersprachen wie JAVA.¹¹ Man könnte auch sagen, dass Typen *First-Class-Objekte* im Sinne von Abschnitt 5 sind.

Insbesondere haben Typen ebenfalls eine Normalform, anhand derer man beurteilen kann, ob (J4) auf sie zutrifft. Beispielsweise sind die beiden folgenden Ausdrücke erlaubt, liegen jedoch nicht in Normalform vor.

```
(car (cons Nat Atom))
(cons (cdr (cons Nat Atom)) (car (cons Nat Atom)))
```

Aufgabe 15.5. Geben Sie für die Ausdrücke aus dem obigen Beispiel die Normalform an. Überlegen Sie sich dann, wie man die Ausdrücke in PIE eingeben muss, um Fehlermeldungen zu vermeiden. Überprüfen Sie damit Ihre Antwort.

Aufgabe 15.6. Welchen Typ hat der Ausdruck `(cons zero 'four)`?

Aufgabe 15.7. Ist `'(3 4)` derselbe Ausdruck wie `(cons 3 4)`?

16. Funktionen

Funktionen in PIE sehen aus wie Funktionen in SCHEME, aber die Ähnlichkeit ist ebenfalls nur äußerlich. Die Funktionen in PIE sind – wie alles andere auch – Ausdrücke und eher mit [Abstraktionen](#) im [Lambda-Kalkül](#) vergleichbar.

Fangen wir mit einem Beispiel an und geben im Definitionsfenster das hier ein:

```
(claim add2
  (-> Nat Nat))

(define add2
  (lambda (n)
    (add1 (add1 n))))
```

Man kann nun im Interaktionsfenster `(add2 1)` eingeben und erhält (Überraschung!) als Ergebnis 3. Was genau geschieht hier?

- **Variablen** sind Zeichenketten,¹² die als Namen verwendet werden, z. B. `n`.
- Wir lassen Ausdrücke zu, in denen Teilausdrücke durch Variablen ersetzt wurden, z. B. `(add1 (add1 n))`.
- Sind A und B Ausdrücke und ist x eine Variable, so schreiben wir $A[x \leftarrow B]$ für den Ausdruck, den man erhält, wenn man x in A durch B ersetzt, wobei wir freie und gebundene Variablen wie in der Definition auf Seite 49 behandeln.

$A[x \leftarrow B]$

¹¹In so einer Sprache gibt es zwar Typen wie `int` oder `float`, aber sie dürfen nur an ganz bestimmten Stellen im Programm auftauchen. Man kann sie z. B. nicht in Variablen speichern oder mit ihnen „rechnen“.

¹²Zugelassen sind im Prinzip dieselben Zeichenketten, die auch in SCHEME zugelassen sind. Einige Zeichenketten sind jedoch [reserviert](#) und dürfen nicht verwendet werden. Siehe Fußnote 5.

Dies ist eine weitere Relativierung der Aussage aus Abschnitt 14, dass jeder Ausdruck einen Typ hat. Ausdrücke wie $(\text{car } p)$, in denen Variablen vorkommen, haben natürlich für sich genommen keinen Typ und führen entsprechend zu einer Fehlermeldung („unknown variable“), wenn man sie im Interaktionsfenster eingibt. Sie werden jedoch typischerweise in einem Kontext auftauchen, in dem den Variablen ein Typ zugeordnet ist.

Um zu wissen, was genau beim Ersetzen zu beachten ist, muss Klarheit darüber bestehen, wann Variablen in PIE **frei oder gebunden** sind. Im Lambda-Kalkül sind Abstraktionen die einzigen Ausdrücke, die Variablen binden. In PIE binden die im Folgenden eingeführten *Funktionen* (die genauso aussehen wie die in Aufgabe 10.2 definierten Abstraktionen) ebenfalls Variablen, sind aber nicht die einzigen Ausdrücke, die das tun. Später werden wir weitere kennenlernen – siehe dazu Aufgabe 21.17.

-> lambda

Seien T_1 und T_2 Typen. Ein Ausdruck vom Typ $(\rightarrow T_1 T_2)$ hat die Form $(\text{lambda } (x) A)$, wobei A ein Ausdruck, x eine Variable und $A[x \leftarrow B]$ vom Typ T_2 ist, wenn B vom Typ T_1 ist. Wir werden solche Ausdrücke auch **Funktionen** von T_1 nach T_2 nennen.

Die **Anwendung** einer solchen Funktion auf einen Ausdruck B schreibt man $((\text{lambda } (x) A) B)$. Sie ist dasselbe T_2 wie $A[x \leftarrow B]$.

lambda ist also der Konstruktor für Funktionen und Anwendung ist der (einzige) Eliminator. (Eliminatoren müssen keine Namen haben wie car.)

Und im Eingangsbeispiel ist $(\text{add2 } 1)$ dieselbe natürliche Zahl wie 3, weil formal einfach in $(\text{add1 } (\text{add1 } n))$ für n der Ausdruck $(\text{add1 } \text{zero})$ eingesetzt wurde. (3 ist bekanntlich nur syntaktischer Zucker.) Wichtig ist dabei, dass keine „Berechnung“ wie in SCHEME stattfand. Hier wurde nur der Ausdruck $(\text{add2 } 1)$ nach bestimmten Regeln in den Ausdruck 3 überführt. Oder ausführlich: Es wurde

$$((\text{lambda } (n) (\text{add1 } (\text{add1 } n))) (\text{add1 } \text{zero}))$$

durch Ersetzen in

$$(\text{add1 } (\text{add1 } (\text{add1 } \text{zero})))$$

überführt.¹³

Der entscheidende Unterschied zum Lambda-Kalkül, in dem auch nur Ausdrücke ineinander überführt werden, sind die Typen.

Ein weiteres Beispiel:

¹³Wenn trotzdem Formulierungen verwendet werden, in denen Funktionen „aufgerufen“ werden oder etwas „zurückgeben“, geschieht das, weil es prägnanter ist. Sie sollten jedoch im Hinterkopf haben, dass wir faktisch immer nur über Ersetzungsvorgänge sprechen.

```
(claim wrong (-> Nat Atom))

(define wrong
  (lambda (n)
    (add2 n)))
```

PIE lässt das nicht zu, weil die Typangabe falsch ist. Das wird Sie vielleicht nicht überraschen, weil der [Compiler](#) einer Sprache wie JAVA auch solche Prüfungen vornimmt. Wir werden allerdings am Anfang [des nächsten Abschnitts](#) sehen, dass die Prüfungen, die ein System wie PIE vornimmt, strenger sind.

Aufgabe 16.1. Mit der am Anfang des Abschnitts definierten Funktion `add2`, deren Variable den Namen `n` hatte, würde

```
(check-same (-> Nat Nat) add2 (lambda (k) (add1 (add1 k)))))
```

zu einer positiven Antwort führen. Was kann man daraus über das Kriterium für Gleichheit bei Funktionen schließen?

Aufgabe 16.2. Welchen Typ hat die Funktion `car`?

Weiter geht's mit Beispielen:

```
(claim Nat-Pair-Pair U)

(define Nat-Pair-Pair
  (Pair
    (Pair Nat Nat)
    (Pair Nat Nat)))
```

Neu ist hier zunächst, dass wir mit `define` auch Name für Typen vergeben können. `U` ist das „Universum“ aus [Abschnitt 15](#). Die `claim`-Deklaration besagt also, dass `Nat-Pair-Pair` der Name eines Typen sein wird.

Geben wir `(cons (cons 1 2) (cons 3 4))` ein, so bekommen wir die schon bekannte Meldung, dass PIE den Typ nicht zweifelsfrei ermitteln kann. Mit

```
(the Nat-Pair-Pair
  (cons (cons 1 2) (cons 3 4)))
```

klappt es natürlich.

Nun die Funktion:

```
(claim cadr
  (-> Nat-Pair-Pair Nat))

(define cadr
  (lambda (p)
    (car (cdr p))))
```

Diese Funktion, die die SCHEME-Funktion `cadr` kopiert, ist natürlich nicht sehr beeindruckend. Außerdem funktioniert Sie nur für einen ganz bestimmten Datentyp.¹⁴ Wir können aber zumindest eine Sache beobachten. Wenn wir

```
(cadr (cons (cons 1 2) (cons 3 4)))
```

eingeben, erhalten wir nicht nur die zu erwartende Antwort 3, sondern PIE beschwert sich auch nicht über einen nicht zu determinierenden Typ. Der konnte nämlich in diesem Fall aus dem Kontext – dem `cadr` davor – geschlossen werden. Genau genommen passierte das weiter oben bereits: `(cons 1 2)` würde PIE, wie wir schon wissen, normalerweise auch nicht einfach so schlucken. Aber als Teil des obigen Ausdrucks, der mit `(the Nat-Pair-Pair` anfangt, war es OK.

Aufgabe 16.3. Hätte man dieselbe Funktion `cadr` auch ohne die Definition von `Nat-Nat-Pair` definieren können? Wie?

Das nächste Beispiel:

```
(claim APair U)

(define APair (Pair Atom Atom))

(claim APair-Maker
  (-> Atom Atom APair))

(define APair-Maker
  (lambda (a b)
    (cons b a)))
```

`APair` ist nur eine Abkürzung, die uns Schreibarbeit sparen soll. Neu ist, dass wir es hier mit einer Funktion mit zwei Argumenten zu tun haben. In PIE ist das tatsächlich nur "[syntaktischer Zucker](#)" für das auf Seite 17 zum ersten Mal erwähnte *Currying*. Die Definition von `APair-Maker` wird wie im Lambda-Kalkül lediglich als Abkürzung für

```
(define APair-Maker
  (lambda (a)
    (lambda (b)
      (cons b a))))
```

interpretiert. Und dazu passend ist so etwas wie `(-> Atom Atom APair)` eine Abkürzung für `(-> Atom (-> Atom APair))` und `(APair-Maker 'eins 'zwei)` eine für `((APair-Maker 'eins) 'zwei)`. Sie könnten in allen drei Fällen (oder nur einigen) die Abkürzungen durch die Langformen ersetzen und es würde sich nichts ändern. Für mehr als zwei Argumente funktioniert das natürlich analog.¹⁵

¹⁴Wie man das allgemeiner macht, [lernen wir noch](#).

¹⁵Man beachte, dass die Anwendung wie im Lambda-Kalkül [linksassoziativ](#) ist, während bei `->` rechts geklammert wird. Das ist nicht etwa ein Versehen, sondern absichtlich so gebaut, damit es zueinander passt.

Wir können bisher keine besonders interessanten Funktionen schreiben, weil wir nur wenige Eliminatoren kennen. Bevor wir [im nächsten Abschnitt](#) Eliminatoren für den Typ `Nat` kennenlernen, schauen wir uns aber noch eine theoretische Alternative zu `car` und `cdr` an. Sie verdeutlicht ein Prinzip, das noch öfter verwendet werden wird.

```
(claim elim-APair
  (-> APair (-> Atom Atom Atom) Atom))

(define elim-APair
  (lambda (p fn)
    (fn (car p) (cdr p))))
```

Formal neu ist, dass `elim-APair` als zweites Argument eine Funktion erwartet. Aber so etwas kennen wir ja schon aus der [funktionalen Programmierung](#).

Interessanter ist, dass man `elim-APair` als einen Eliminator für Ausdrücke vom Typ `APair` interpretieren kann. Es handelt sich gewissermaßen um einen „indirekten“ Eliminator, da er nicht wie `car` und `cdr` die einzelnen Komponenten einfach zurückgibt, sondern diese stattdessen einer vom Aufrufenden spezifizierten Funktion `fn` übergibt.

Aufgabe 16.4. Schreiben Sie Eliminatoren `car-APair` und `cdr-APair` für `APair`, die wie `car` und `cdr` funktionieren. Sie dürfen allerdings dafür `car` und `cdr` nicht verwenden, sondern haben nur `elim-APair` zur Verfügung.

17. Rekursion über natürliche Zahlen

Zuerst die schlechte Nachricht: In PIE darf eine Funktion sich nicht selbst aufrufen. Der technische Grund ist, dass – anders als in SCHEME – in Definitionen nur Namen vorkommen dürfen, die bereits *vorher* definiert waren. Das wiederum liegt daran, dass es sonst möglich wäre, „Funktionen“ zu definieren, die keinen Typ haben, beispielsweise so:

```
(define forever
  (lambda (and-ever)
    (forever and-ever)))
```

Auch so etwas wie der [Y combinator](#) des Lambda-Kalküls ist keine Option, denn wir haben gesehen, dass damit ebenfalls [Endlosschleifen](#) möglich sind. PIE akzeptiert im Rumpf einer Funktion nur Ausdrücke, von denen das System beweisen kann, dass die Auswertung der Funktion zu einer Normalform führt. Anders ausgedrückt: Funktionen in PIE müssen **total** sein, d. h., sie müssen für jedes mögliche Argument terminieren und ein Ergebnis vom angegebenen Typ liefern.¹⁶

¹⁶Im Prinzip gilt diese Einschränkung auch für „ausgewachsene“ Systeme wie [LEAN](#). Die machen sich aber mehr Arbeit als PIE. Sie erlauben in gewissen Fällen, dass eine Funktion sich selbst aufruft,

Aber wie soll man rekursive Funktionen schreiben, wenn diese sich nicht selbst aufrufen dürfen? Ist das nicht das Wesen der Rekursion? Die Antwort von PIE auf diese Frage ist eine Art „kontrollierte Rekursion“, für die entsprechende Eliminatoren zur Verfügung gestellt werden.

Im Folgenden sehen wir drei Eliminatoren für natürliche Zahlen, von denen eigentlich nur der dritte benötigt wird. Die anderen beiden werden aus [didaktischen](#) Gründen zuerst eingeführt, um zu verdeutlichen, welche „Zutaten“ vonnöten sind.

Bei der Rekursion über natürliche Zahlen, z. B. in Abschnitt 9, war ein wesentlicher Punkt immer eine Fallunterscheidung zwischen dem einfachen Fall, dass das Argument null ist, und der Alternative, dass das Argument positiv ist. In SCHEME kann man dafür das [Prädikat zero?](#), eine Funktion für den „Vorgänger“ wie `sub1` aus Aufgabe 6.9 und eine der [Spezialformen if](#) oder `cond` verwenden.¹⁷ PIE vereinigt das in einem Eliminator:

`which-Nat`

`which-Nat` ist eine Funktion mit dem Typ $(\rightarrow \text{Nat } T (\rightarrow \text{Nat } T) T)$, wobei T ein Typ ist.¹⁸ Die Funktion bekommt also drei Argumente. Diese werden typischerweise `target`, `base` und `step` genannt. Ist `target` dieselbe natürliche Zahl wie `zero`, dann ist

$$(\text{which-Nat target base step}) \quad (17.1)$$

dasselbe T wie `base`. Und ist `target` dieselbe natürliche Zahl wie `(add1 n)`, dann ist der Ausdruck (17.1) dasselbe T wie `(step n)`.

Informell ausgedrückt wird `which-Nat` mit einer Zahl m als erstes Argument aufgerufen und gibt einen Ausdruck vom Typ T zurück. War m die Zahl 0, so gibt die Funktion einfach ihr zweites Argument zurück. War m nicht 0, so ruft `which-Nat` ihr drittes Argument, eine Funktion, mit dem Argument $m - 1$ auf.

Damit können wir nun z. B. so etwas schreiben:¹⁹

```
(claim positive? ( $\rightarrow$  Nat Atom))

(define positive?
  (lambda (n)
    (which-Nat n
      'no
      (lambda (n-1) 'yes))))
```

und prüfen dann, ob sie terminiert. Ein hundertprozentig zuverlässiger Test ist jedoch aufgrund des [Halteproblems](#) nicht möglich. (Mehr zu Letzterem im Modul [Theoretische Informatik](#).)

¹⁷Siehe Aufgabe 17.3.

¹⁸`which-Nat` ist also auf die gleiche Art „flexibel“ wie `car` – siehe Aufgabe 16.2.

¹⁹Beachten Sie, dass `n-1` ein *Name* ist. Hier wird *nicht* subtrahiert.

Zugegeben, auch das ist keine besonders aufregende Funktion. Aber immerhin hätten wir sie ohne `which-Nat` gar nicht schreiben können. Die folgende ebenfalls nicht.

Aufgabe 17.1. Schreiben Sie mithilfe von `which-Nat` eine PIE-Funktion `sub1`, die das Verhalten der gleichnamigen SCHEME-Funktion aus Aufgabe 6.9 kopiert.

Aufgabe 17.2. Schreiben Sie mithilfe von `which-Nat` eine PIE-Funktion `switch`, die aus einer Null eine Eins und aus einer Eins eine Null macht. Das Verhalten bei anderen Zahlen ist irrelevant.

Aufgabe 17.3. Schreiben Sie eine SCHEME-Funktion, die das Verhalten des Eliminators `which-Nat` simuliert.

`sub1` aus Aufgabe 17.1 ist eine der wenigen sinnvollen Funktionen, die man mithilfe von `which-Nat` definieren kann. `which-Nat` bietet zwar bei positiven natürlichen Zahlen die Möglichkeit, „einen Schritt zurück“ zu gehen, aber von dort aus geht es nicht weiter. Eine echte Rekursion geht jedoch so lange „rückwärts“, bis der einfache Fall null erreicht ist. Wir brauchen einen „mächtigeren“ Eliminator.

`iter-Nat` ist eine Funktion mit dem Typ $(\rightarrow \text{Nat } T (\rightarrow T T) T)$, wobei T ein Typ ist.²⁰ Ist `target` dieselbe natürliche Zahl wie `zero`, dann ist

$$(\text{iter-Nat target base step}) \quad (17.2)$$

dasselbe T wie `base`. Ist `target` dieselbe natürliche Zahl wie `(add1 n)`, dann ist der Ausdruck (17.2) dasselbe T wie `(step (iter-Nat n base step))`.

`iter-Nat`

Der Unterschied zu `which-Nat` ist, dass bei positivem Argument m die Funktion `step` auf `(iter-Nat  $m - 1$  base step)` angewendet wird und nicht einfach auf $m - 1$. Wir können nun z. B. so etwas definieren:

```
(claim add3 ( $\rightarrow$  Nat Nat))

(define add3
  (lambda (n)
    (iter-Nat n
      3
      (lambda (prev-step)
        (add1 prev-step)))))
```

Aufgabe 17.4. Bevor Sie es in PIE ausprobieren: Setzen Sie sich mit Zettel und Stift hin und schreiben Sie auf, was bei `(add3 0)`, `(add3 1)` und `(add3 2)` herauskommt. Schreiben Sie die Umformungen so ausführlich auf, dass Ihnen klar ist, *wie* die Ergebnisse zustande kommen. Was „macht“ `add3`?

²⁰Der Typ des dritten Arguments unterscheidet sich von dem bei `which-Nat`!

OK, werden Sie sagen, aber `add3` hätten wir auch ohne `iter-Nat` schreiben können – siehe `add2` am Anfang von Abschnitt 16. Das war jedoch nur eine Vorbereitung für die nächste Aufgabe.

Aufgabe 17.5. Schreiben Sie mithilfe von `iter-Nat` eine Funktion zum Addieren zweier natürlicher Zahlen. (Hinweis: `add3` ist *fast* schon die Lösung. Man muss nur aus der „fest verdrahteten“ Zahl 3 ein Argument der Funktion machen.)

Aufgabe 17.6. Schreiben Sie mithilfe von `iter-Nat` sowie von Aufgabe 17.5 eine Funktion zum Multiplizieren zweier natürlicher Zahlen. (Hinweis: Schauen Sie sich ggf. noch einmal die Lösung von Aufgabe 9.3 an.)

Aufgabe 17.7. Definieren Sie mithilfe von `iter-Nat` und Aufgabe 17.1 eine Funktion, die die Differenz zweier natürlichen Zahlen wie in Aufgabe 9.7 berechnet.

Aufgabe 17.8. Schreiben Sie mithilfe von `iter-Nat` sowie von Aufgabe 17.2 eine Funktion `parity`, die bei geraden Zahlen mit 0 antwortet und bei ungeraden mit 1.

Aufgabe 17.9. Schreiben Sie eine SCHEME-Funktion, die das Verhalten von `iter-Nat` simuliert.

Aufgabe 17.9 sollte noch einmal verdeutlichen, was `iter-Nat` eigentlich macht. In Pseudocode könnte man es so darstellen:

```
result ← base
counter ← 0
WHILE counter < target:
  result ← step(result)
  counter ← counter + 1
RETURN result
```

Eigentlich handelt es sich dabei gar nicht um Rekursion, sondern um **Iteration** – daher auch der Name. `counter` ist lediglich ein Zähler, der dafür sorgt, dass `target` mal die Funktion `step` auf `base` angewendet wird.

Wollte man in SCHEME die Summe der Zahlen von 1 bis n rekursiv programmieren,²¹ so könnte man es folgendermaßen machen:

```
(define sum
  (lambda (n)
    (cond ((zero? n) 0)
          (else (+ n (sum (sub1 n)))))))
```

Hier benötigt man für die Addition im `else`-Teil zwei „Zutaten“: erstens das aktuelle Argument `n` und zweitens den Wert `(sum (sub1 n))` des vorherigen Rekursionsschrittes. Die erste Zutat kann `which-Nat` liefern, die zweite `iter-Nat`, aber beiden fehlt jeweils die andere Hälfte. Der „richtige“ Eliminator für die Rekursion fehlt noch.

²¹Weil man die *gaußsche Summenformel* nicht kennt ...

`rec-Nat` ist eine Funktion mit dem Typ $(\rightarrow \text{Nat } T (\rightarrow \text{Nat } T T) T)$, wobei T ein Typ ist. Ist `target` dieselbe natürliche Zahl wie `zero`, dann ist

$$(\text{rec-Nat target base step}) \quad (17.3)$$

dasselbe T wie `base`. Ist `target` dieselbe natürliche Zahl wie `(add1 n)`, dann ist der Ausdruck (17.3) dasselbe T wie `(step n (rec-Nat n base step))`.

`rec-Nat`

Dass `rec-Nat` tatsächlich die Fähigkeiten von `which-Nat` und `iter-Nat` in sich vereinigt, zeigen die Aufgaben 17.11 und 17.12.

Damit ist `sum` in PIE nun kein Problem mehr:

```
(claim sum (-> Nat Nat))

(define sum
  (lambda (n)
    (rec-Nat n
      0
      (lambda (n-1 step-1)
        (+ (add1 n-1) step-1))))))
```

Aufgabe 17.10. Definieren Sie mithilfe von `rec-Nat` und Aufgabe 17.6 eine Funktion, die die Fakultät wie in Aufgabe 9.14 berechnet.

Aufgabe 17.11. Definieren Sie mithilfe von `rec-Nat` eine Funktion `my-which-Nat`, die dasselbe wie `which-Nat` macht, wenn man für T den Typ `Nat` wählt. (Mit anderen Worten: Mit `my-which-Nat` sollte man `sub1` wie in der Lösung von Aufgabe 17.1 definieren können.)

Aufgabe 17.12. Definieren Sie mithilfe von `rec-Nat` eine Funktion `my-iter-Nat`, die dasselbe wie `iter-Nat` macht, wenn man für T den Typ `Nat` wählt. (Mit anderen Worten: Mit `my-iter-Nat` sollte man `+` wie in der Lösung von Aufgabe 17.5 definieren können.)

Aufgabe 17.13. Ein Kommilitone von Ihnen meint, `sum` ließe sich auch mit `iter-Nat` realisieren, und zwar so:

```
(claim sum (-> Nat Nat))

(define sum
  (lambda (n)
    (iter-Nat n
      0
      (lambda (step-1)
        (+ n step-1)))))
```

Wenn man es ausprobiert, merkt man, dass es nicht funktioniert. Aber wo ist hier eigentlich der Denkfehler?

Aufgabe 17.12 hat demonstriert, dass man [Iteration](#) im Prinzip immer durch Rekursion ersetzen kann. Umgekehrt gilt das allerdings auch, wie die folgende Aufgabe zeigt.

★**Aufgabe 17.14.** Definieren Sie mithilfe von `iter-Nat` eine Funktion `my-rec-Nat`, die dasselbe wie `rec-Nat` macht, wenn man für T den Typ `Nat` wählt. (Hinweis: Die vor der Definition von `rec-Nat` angesprochene zweite „Zutat“, die `iter-Nat` fehlt, kann man in einem [Paar](#) „verpacken“.)

18. Parametrisierte und abhängige Typen

Damit wir mehr Beispiele haben, schauen wir uns noch weitere Typen an.

```
nil :: List
```

Ist T ein Typ, dann ist `nil` ein Ausdruck vom Typ `(List T)`. Ist X ein Ausdruck vom Typ T und L ein Ausdruck vom Typ `(List T)`, dann ist `(:: X L)` ebenfalls ein Ausdruck vom Typ `(List T)`. Wir nennen solche Ausdrücke auch **T -Listen**.

Das ist eine rekursive Definition ähnlich wie die für den Typ [Nat](#); und die Konstruktoren sind `nil` und `::`. Im Prinzip werden also Listen in PIE so konstruiert, wie wir es aus Abschnitt 4 für Listen in SCHEME kennen. `nil` spielt hier die Rolle von `()` und `::` die von `cons`.²² Wie in SCHEME werden wir auch hier von den **Elementen** der Liste sprechen. Im Unterschied zu SCHEME müssen allerdings in PIE alle Elemente einer Liste vom selben Typ sein.

Leider gibt es in PIE keine Kurzschreibweise für Listen und auch keine Funktion wie [list](#) oder eine andere Art, bequem Listen mit mehreren Elementen einzugeben. Es geht nur so:

```
(the (List Atom) nil)
(the (List Nat) nil)
(the (List Atom) (:: 'quark nil))
(the (List Nat) (:: 23 (:: 42 nil)))
```

Gleichheit von Listen bedeutet intuitiv einfach, dass die einzelnen Elemente jeweils paarweise gleich sind.

```
(check-same (List Nat)
  (:: 1 (:: 2 nil))
  (:: (cdr (the (Pair Nat Nat) (cons 2 1)))
     (:: (add2 zero) nil)))
```

Wir halten das jedoch noch einmal formal fest, um die Denkweise der Typentheorie zu verinnerlichen und uns zu vergegenwärtigen, wie PIE intern prüft, ob zwei Ausdrücke gleich sind.

²²Bekanntlich gibt es `cons` in PIE für Paare, aber man kann hier nicht denselben Konstruktor verwenden.

`nil` ist derselbe Ausdruck vom Typ $(\text{List } T)$ wie `nil`. Ist X_1 derselbe Ausdruck vom Typ T wie X_2 und L_1 derselbe Ausdruck vom Typ $(\text{List } T)$ wie L_2 , dann ist $(:: X_1 L_1)$ derselbe Ausdruck vom Typ $(\text{List } T)$ wie $(:: X_2 L_2)$.

Für Listen gibt es nur einen Eliminator, der analog zu `rec-Nat` die Möglichkeit zur Rekursion bietet.

`rec-List` ist eine Funktion mit dem Typ

$$(\rightarrow (\text{List } T) S \rightarrow T (\text{List } T) S S) S,$$

wobei T und S Typen sind. Ist `target` dieselbe T -Liste wie `nil`, dann ist

$$(\text{rec-List target base step}) \tag{18.1}$$

dasselbe S wie `base`. Ist `target` dieselbe T -Liste wie $(:: X L)$, dann ist der Ausdruck (18.1) dasselbe S wie $(\text{step } X L (\text{rec-List } L \text{ base step}))$.

`rec-List`

Im Rekursionsschritt wird eine Liste, die nicht `nil` ist, also „zerlegt“ und die Funktion `step` bekommt die beiden Teile – erstes Element und Restliste – sowie das Ergebnis des vorherigen Schritts als Argumente übergeben. Füllen wir das durch ein Beispiel mit Leben:

```
(claim length-Atom ( $\rightarrow$  (List Atom) Nat))

(define length-Atom
  (lambda (list)
    (rec-List list
      0
      (lambda (first rest previous)
        (add1 previous)))))

(length-Atom (:: 'quark (:: 'wurst nil)))
```

Diese Funktion macht im Prinzip das, was das Beispiel `length` auf Seite 27 macht. Der Unterschied ist die in Abschnitt 17 schon angesprochene „kontrollierte Rekursion“. Während die `SCHEME`-Funktion sich selbst aufruft, um die Länge der kürzeren Liste zu ermitteln, liefert uns `rec-List` diese im Argument `previous`.

Störend an Funktionen wie `length-Atom` oder `car-APair` aus Aufgabe 16.4 ist allerdings, dass sie auf einen bestimmten Typ festgelegt sind. Das muss nicht so sein. Ich habe Ihnen bisher lediglich die allgemeine Art und Weise der Funktionsdefinition verschwiegen.

Um die vorzubereiten, erinnern wir uns noch einmal, dass Typen auch nur Ausdrücke sind, mit denen wir „rechnen“ können, z. B. so:

```
(claim pair-Type (-> U U U))

(define pair-Type
  (lambda (T1 T2)
    (Pair T1 T2)))

(pair-Type Atom Nat)
```

Solche „Berechnungen“ kann man auch innerhalb der Definition eines Typs durchführen:

Pi Ist x eine Variable, T ein Typ und S ein Ausdruck, so ist $(\text{Pi } ((x \ T)) \ S)$ ein Typ, dessen Ausdrücke **Funktionen** sind, die wie üblich mit dem Konstruktor `lambda` geschrieben werden und deren Argumente den Typ T haben müssen. Ist f eine Funktion dieses Typs und a ein Objekt vom Typ T , so hat $(f \ a)$ den Typ $S[x \leftarrow a]$.
Im Sinne des **Currying** ist $(\text{Pi } ((x_1 \ T_1) (x_2 \ T_2)) \ S)$ eine Abkürzung für

$$(\text{Pi } ((x_1 \ T_1)) (\text{Pi } ((x_2 \ T_2)) S))$$

und analog für mehr als zwei Argumente.

Der Typ des Rückgabewerts so einer Funktion f hängt also vom Typ des Arguments ab, während bei den Funktionen, die wir bisher kannten, dieser Typ konstant war. Anders ausgedrückt stellt der Typ-Konstruktor `Pi` die allgemeine Form bereit und $(\rightarrow T_1 \ T_2)$ ist lediglich eine Abkürzung für $(\text{Pi } ((x \ T_1)) \ T_2)$, wenn x in T_2 nicht vorkommt.²³

Falls Sie das verwirrend fanden, hier ein einfaches Beispiel:

```
(claim duplicate (Pi ((T U))
                     (-> T (Pair T T))))

(define duplicate
  (lambda (T x)
    (cons x x)))

(duplicate Nat 2)
(duplicate Atom 'two)
(duplicate (List Nat) (: 2 nil))
```

Wir können also nun mit ein und derselben Funktion Objekte ganz unterschiedlichen Typs als Argumente akzeptieren und als Rückgabewert ausgeben. Dafür bekommt der Typ des ersten Arguments im `Pi`-Ausdruck einen Namen (hier T), der dann im Rest des Ausdrucks verwendet werden kann.

²³Wie `lambda` kann man auch `Pi` und das **später noch auftauchende** Sigma durch griechische Buchstaben ersetzen. Siehe Fußnote 28 auf Seite 14.

Man hätte übrigens den Typ von `duplicate` auch so angeben können:

```
(claim duplicate (Pi ((T U) (x T)) (Pair T T)))
```

Das ist kürzer, hat aber den „Schönheitsfehler“ des ungenutzten Arguments `x`. Welche Version man bevorzugt, ist reine Geschmackssache.

Aufgabe 18.1. Schreiben Sie `elim-APair` und `car-APair` aus Aufgabe 16.4 so um, dass diese Funktionen für beliebige Paare einsetzbar sind. (Beachten Sie, dass wir für allgemeine Paare *zwei* Typen benötigen. Außerdem sollte sich die Funktion `fn` idealerweise einen Ergebnistyp „aussuchen“ dürfen.)

Aufgabe 18.2. Ändern Sie die Definition von `length-Atom` so ab, dass die Funktion die Länge von beliebigen Listen ermitteln kann.

Aufgabe 18.3. Der Konstruktor `::` setzt ein neues Element an den Anfang einer Liste. Schreiben Sie mithilfe von `rec-List` eine Funktion, die ein Element an das *Ende* einer List anhängt. Das sollte so funktionieren:

```
> (append Atom 'quark nil)
(the (List Atom)
  (:: 'quark nil))
> (append Atom 'wurst (append Atom 'quark nil))
(the (List Atom)
  (:: 'quark
    (:: 'wurst nil)))
> (append Nat 4 (:: 1 (:: 2 (:: 3 nil))))
(the (List Nat)
  (:: 1
    (:: 2
      (:: 3
        (:: 4 nil)))))
```

Was wir bisher hier machen, kennen Sie im Prinzip auch von konventionellen Sprachen. In JAVA könnte man beispielsweise so etwas schreiben:

```
static <T> T first(List<T> list) {
  return list.get(0);
}
```

Hier steht `T` ebenfalls für einen Typen und die so definierte Funktion `first` funktioniert sowohl für `List<String>` als auch für `List<Integer>` oder andere Typen. Man spricht auch von **parametrisierten Typen**, wobei `T` sowohl bei `first` als auch bei `append` aus Aufgabe 18.3 der Parameter ist.

Gleichzeitig fällt spätestens jetzt auf, dass die Syntax von PIE trotz `Pi` nach wie vor umständlich ist. Wir müssen bei jedem Aufruf von `car-Pair` aus Aufgabe 18.1 explizit zwei Typen angeben, während so etwas beim Aufruf von `first` in JAVA nicht nötig wäre. Wir haben außerdem immer noch keine Möglichkeit, eine Funktion wie

Was soll an der Stelle der Fragezeichen stehen? Aber eigentlich ist es ganz einfach. Dort muss die Länge der Liste stehen, für die wir in Aufgabe 18.2 schon eine Funktion gebastelt haben! In einem System mit abhängigen Typen darf man so etwas schreiben:

```
(claim list->vec (Pi ((T U) (L (List T)))
                    (Vec T (length T L))))
```

Allerdings werden Sie ein Problem bekommen, wenn Sie versuchen, unsere obige Definition von `list->vec` zu verwenden. PIE wird sich darüber beschweren, dass Typen nicht klar zu erkennen sind. Wir benötigen dafür eine Erweiterung von `rec-List`.

`ind-List` funktioniert im Prinzip wie `rec-List`, aber mit erweiterten Typ-Informationen. Dafür gibt es ein zusätzliches Argument `motive` vom Typ $(\rightarrow (\text{List } T) U)$ an zweiter Stelle (zwischen `target` und `base`) sowie die folgenden Änderungen:

`ind-List`

- (i) `base` hat statt `S` den Typ `(motive nil)`.
- (ii) Das Ergebnis von `rec-List` hat statt `S` den Typ `(motive target)`.
- (iii) Der Typ von `step` ist nun


```
(Pi ((first T) (rest (List T)))
      (-> (motive rest) (motive (:: first rest))))
```

Der feste Typ `S` ist also verschwunden und wurde durch eine Funktion `motive` ersetzt, die einen *abhängigen Typen* berechnet. Dieser wird determiniert durch den aktuellen Wert der Liste in der Rekursion. Die folgende Tabelle verdeutlicht den Unterschied, indem der feste Typ `S` aus der Definition von `rec-List` dem berechneten Typen von `ind-List` gegenübergestellt wird.

	<code>rec-List</code>	<code>ind-List</code>
Argumente	<code>target base step</code>	<code>target motive base step</code>
Typ <code>target</code>	<code>(List T)</code>	<code>(List T)</code>
Typ <code>base</code>	<code>S</code>	<code>(motive nil)</code>
Typ <code>step</code>	<code>(-> T (List T) (-> S S))</code>	<code>(Pi ((first T) (rest (List T))) (-> (motive rest) (motive (:: first rest))))</code>
Typ Ergebnis	<code>S</code>	<code>(motive target)</code>

Damit kann man den Job folgendermaßen erledigen:²⁵

```
(define list->vec
  (lambda (T list)
    (ind-List list ;; ind- statt rec-
      (lambda (lst) (Vec T (length T lst))) ;; neu
      vecnil
```

²⁵Damit es nicht zu Verwirrung kommt, habe ich `lst` statt `list` verwendet. Letzteres hätte aber auch funktioniert.

```
(lambda (first rest previous)
  (vec:: first previous))))

(list->vec Nat (:: 3 (:: 2 (:: 1 nil))))
```

Zwei Anmerkungen zur Terminologie:

- Der Begriff *motive* ist in der Typentheorie üblich, wenn es um Funktionen geht, die einen (abhängigen) Ergebnistypen angeben. Mit dem Wort ist so etwas wie der „Zweck“ der Rekursion gemeint ist. Ich finde aber weder *Zweck* noch *Intention* noch *Motiv* so richtig passend und werde im Folgenden den Begriff **Zieltypfunktion** verwenden.
- Dass es einmal **ind-List** und einmal **rec-List** heißt, ist auch dem üblichen Sprachgebrauch in der Typentheorie geschuldet. Während rein mathematisch Rekursion und Induktion eng verwandte Begriffe sind, spricht man dort häufig erst dann von *Induktion*, wenn es sich um eine Rekursion mit einer Zieltypfunktion handelt, die nicht konstant ist. (Siehe z. B. die Aufgaben 18.4 und 18.5.)

Aufgabe 18.4. Wenn man die Lösung von Aufgabe 18.2 mit **ind-List** statt **rec-List** geschrieben hätte, wie hätte dann die Funktion ausgesehen, die in der Definition von **ind-List** *motive* heißt?

ind-Nat

Aufgabe 18.5. Es gibt ein Pendant namens **ind-Nat** zu **rec-Nat**, das wie **ind-List** im Vergleich zu **rec-List** ein zusätzliches Argument *motive* erwartet, das Informationen über die Typen bereitstellt.²⁶

Verwenden Sie **ind-Nat**, um eine Funktion **repeat** zu schreiben, die so etwas macht:

```
> (repeat Atom 3 'wurst)
(the (Vec Atom 3)
  (vec:: 'wurst (vec:: 'wurst (vec:: 'wurst vecnil))))
> (repeat Nat 5 42)
(the (Vec Nat 5)
  (vec:: 42 (vec:: 42 (vec:: 42 (vec:: 42 vecnil)))))
> (repeat (Vec Nat 0) 0 vecnil)
(the (Vec (Vec Nat 0) 0) vecnil)
```

19. Gleichheit von Funktionen

In diesem Abschnitt klären wir, wann aus der Sicht von PIE zwei Funktionen gleich sind. (Siehe Aufgabe 16.1.) Damit bereiten wir uns auf die Frage vor, wofür auf Typentheorien basierende Programmiersprachen überhaupt verwendet werden.

²⁶Weitere Informationen finden Sie in der [PIE-Dokumentation](#). Es sollte auch reichen, **ind-Nat** in **DRRACKET** einzutippen und dann die Taste **F1** zu drücken.

Für das folgende Beispiel sollte `add2` exakt so wie am Anfang von Abschnitt 16 definiert sein. Außerdem benötigen wir die Lösung + von Aufgabe 17.5 (auch genau in der dortigen Form) sowie das hier:

```
(claim plus (-> Nat Nat Nat))

(define plus
  (lambda (k n)
    (iter-Nat k
      n
      (lambda (step-1)
        (add1 step-1))))))
```

Aufgabe 19.1. Wie unterscheiden sich `plus` und `+`?

`plus` und `+` unterscheiden sich formal im "Quelltext", aber beide Funktionen erledigen korrekt ihren Job der Addition zweier Zahlen.

```
(check-same Nat (+ 19 23) (plus 19 23))
```

Beide sind oberflächlich betrachtet „zweistellige“ Funktionen, aber wir wissen ja, dass das nur *Currying* ist. `(plus 2)` ist eine (einstellige) Funktion, die wir wie jede andere Funktion anwenden können: `((plus 2) 40)`. Und offenbar „macht“ die Funktion `(plus 2)` dasselbe wie `add2`. Sollten die beiden etwa aus der Sicht von PIE *dieselbe*²⁷ Funktion sein? Das ist in der Tat der Fall:

```
(check-same (-> Nat Nat) (plus 2) add2)
```

Das müsste doch auch für `(+ 2)` klappen, oder? Tut es aber nicht. Geben Sie konkrete Zahlen ein, so liefern `(+ 2)` und `add2` immer dasselbe Ergebnis, aber auf die Frage

```
(check-same (-> Nat Nat) (+ 2) add2)
```

antwortet PIE mit einer Fehlermeldung. Diese Meldung gibt auch schon Aufschluss darüber, warum die beiden Funktionen *nicht* gleich sind. Man kann das durch den folgenden Dialog im Interaktionsfenster noch etwas deutlicher machen:

```
> (the (-> Nat Nat) add2)
(the (-> Nat Nat)
  (lambda (n) (add1 (add1 n))))
> (the (-> Nat Nat) (plus 2))
(the (-> Nat Nat)
  (lambda (n) (add1 (add1 n))))
> (the (-> Nat Nat) (+ 2))
```

²⁷Genauer: dieselbe Funktion *des Typs* `(-> Nat Nat)`. Erinnern Sie sich an Urteil (J3). Aber auch ohne diesen Zusatz, den ich manchmal weglassen sollte hoffentlich klar sein, was gemeint ist.

```
(the (-> Nat Nat)
  (lambda (n)
    (iter-Nat n
      (the Nat 2)
      (lambda (step-1) (add1 step-1))))))
```

Bei der zweiten Eingabe kommt exakt das heraus, was als Definition von `add2` eingegeben wurde. Bei der dritten jedoch nicht. Wie kam die dritte Antwort zustande?

In Abschnitt 15 hatten wir über Normalformen gesprochen. Vereinfacht gesagt besteht die Normalform eines Ausdrucks idealerweise nur noch aus Konstruktoren und das System versucht, bei Fragen nach der Gleichheit Ausdrücke in Normalform zu überführen. Bei einfachen Objekten wie am Anfang des Kapitels war das kein Problem. Und weil sowohl `add2` als auch `(+ 2)` solche einfachen Objekte zurückgeben, führt auch eine Frage wie

```
(check-same Nat (add2 40) ((+ 2) 40))
```

zu einer positiven Antwort. Wenn es jedoch darum geht, die Normalform einer *Funktion* zu finden, dann versucht PIE, den *Rumpf* der Funktion zu vereinfachen, der eventuell Variablen enthält. `(plus 2)` ist nach Definition die folgende Funktion, weil in der Definition von `plus` für die Variable `k` der Wert 2 substituiert wurde.²⁸

```
(lambda (n)
  (iter-Nat 2
    n
    (lambda (step-1) (add1 step-1))))
```

Den Rumpf dieser Funktion kann PIE weiter vereinfachen, weil 2 ein konkreter Wert ist. Es ist klar, dass die `step`-Funktion in der letzten Zeile auf den „vorherigen“ Wert angewendet werden muss. Das ergibt:

```
(add1 (iter-Nat 1
  n
  (lambda (step-1) (add1 step-1))))
```

Erneut haben wir einen konkreten Wert – nun die Eins – und PIE kann weiter reduzieren:

```
(add1 (add1 (iter-Nat 0
  n
  (lambda (step-1) (add1 step-1))))))
```

Und ein weiteres Mal ist klar, was zu passieren hat. In diesem Fall wird die erste Alternative von `iter-Nat` gewählt:

²⁸Rein formal steht da (siehe Abschnitt 14) nicht 2, sondern `(add1 (add1 zero))`.

```
(add1 (add1 n))
```

Dieser Ausdruck besteht *nicht* nur aus Konstruktoren, aber er lässt sich wegen der Variable n , deren Wert unbekannt ist, nicht weiter reduzieren. PIE hört also auf und dieses Ergebnis – mit `(lambda (n) ...)` drumherum – ist die Normalform von `(plus 2)`. Bei `add2` kommt man auf dieselbe Normalform, da es gar nichts zu reduzieren gibt. Weil diese beiden Normalformen identisch sind,²⁹ sind die beiden Funktionen für PIE dieselbe Funktion.

`(+ 2)` sieht hingegen so aus:

```
(lambda (n)
  (iter-Nat n
    2
    (lambda (step-1) (add1 step-1))))
```

Hier muss PIE sofort die Waffen strecken, weil n irgendeine natürliche Zahl sein kann und daher nicht klar ist, wie es mit `iter-Nat` weitergeht. Wir sehen hier also bereits die Normalform von `(+ 2)` und sie sieht offenbar anders aus als die von `add2`.

Im nächsten Abschnitt überzeugen wir PIE davon, dass `(+ 2 n)` und `(add2 n)` für alle natürlichen Zahlen n dasselbe ist.

Aufgabe 19.2. Die folgende Frage bezieht sich auf die Lösung von Aufgabe 18.5. Beantworten Sie sie *nicht*, indem Sie es einfach ausprobieren, sondern überlegen Sie erst mit Zettel und Bleistift, was herauskommen muss. Danach vergleichen Sie dann.

Was wird die Antwort von PIE sein, wenn man `(repeat Nat 2)` im Interaktionsfenster eingibt?

Aufgabe 19.3. Vereinfachen Sie den folgenden Ausdruck so weit wie möglich.

```
(rec-Nat (add1 n)
  (add2 n)
  (lambda (n-1 step-1)
    (+ step-1 (add1 n-1))))
```

Was können Sie in PIE eingeben, um Ihre Lösung zu überprüfen? (Hinweis: Das ist etwas Fleißarbeit. Man sollte das aber mal gemacht haben, damit man versteht, was intern passiert.)

20. Die Curry-Howard-Korrespondenz

Der Grund dafür, wie Typensysteme aktuell in der Informatik eingesetzt werden, ist die nach zwei amerikanischen Mathematikern benannte **Curry-Howard-Korrespondenz**. Dabei geht es darum, dass in gewissem Sinne das **mathematische**

²⁹ α -Äquivalenz hätte gereicht, siehe Aufgabe 16.1.

Beweisen³⁰ strukturell dasselbe ist wie das, was in einer abhängigen Typentheorie abläuft – wenn man mathematische Aussagen mit Typen identifiziert und Ausdrücke der entsprechenden Typen als Beweise für die zugehörigen Aussagen interpretiert. Wir werden nicht so weit kommen, diese Korrespondenz in Gänze zu verstehen oder gar zu beweisen, wollen sie aber zumindest an einfachen Beispielen nachvollziehen.

Wie ist das also gemeint? Gibt man in PIE

```
(claim forty-two Nat)
(define forty-two 42)
```

ein, so ist `forty-two` in diesem Sinn ein „Beweis“ für den Typ `Nat`. Das erklärt zumindest den Begriff *claim*, den man mit *Anspruch* oder *Forderung* übersetzen kann, aber auch mit *Behauptung*, womit man in der Mathematik etwas meint, was noch bewiesen werden muss. In der ersten Zeile behaupte ich also, dass ich einen Beweis namens `forty-two` für `Nat` habe, in der zweiten präsentiere ich den Beweis. Wenn das System beides akzeptiert, ist der Beweis überprüft.

Da der letzte Satz wichtig ist, will ich auf ihn noch etwas ausführlicher eingehen. Eine Deklaration wie `(define forty-two 42)` wird von einem System wie PIE nicht einfach so hingenommen, sondern es wird wirklich überprüft, ob der Ausdruck, dem der Name `forty-two` gegeben werden soll, den Typ `Nat` hat. Bei 42 sieht das offensichtlich aus,³¹ aber es hätte sich zum Beispiel auch um so etwas wie `(cdr (cons 'quark 23))` handeln können oder um einen noch wesentlich komplizierteren Term. Das System wird ihn so lange reduzieren, bis es sich sicher sein kann, dass der Ausdruck wirklich den richtigen Typ hat. Diese Arbeit ist der adäquate Ersatz für das herkömmliche Überprüfen von Beweisen mit den Mitteln der **mathematischen Logik**.³²

Typen wie `Nat`, die wir bisher gesehen haben, eignen sich nicht dafür, interessante mathematische Aussagen zu formulieren. Es gibt dafür spezielle Typen und einen von denen lernen wir nun kennen.

= Ist T ein Typ und sind x_1 und x_2 Ausdrücke vom Typ T , dann ist $(= T x_1 x_2)$ ein (abhängiger) Typ.

Dieser Typ (mit dem **Typ-Konstruktor** `=`) ist von einer ganz anderen Sorte als beispielsweise `Nat` oder `Atom`. Zunächst einmal ist es ein *abhängiger* Typ wie `Vec`, weil x_1 und x_2 irgendwelche Ausdrücke sein können und unterschiedliche Ausdrücke zu unterschiedlichen Typen führen. Entscheidend ist jedoch die Bedeutung des

³⁰Und auch die **Verifikation** von Programmen.

³¹Arbeit ist es trotzdem, denn 42 ist ja `(add1 (add1 (add1 (add1 ...`

³²Man kann Beweise auch ohne Typentheorie und den Rückgriff auf die Curry-Howard-Korrespondenz maschinell überprüfen. Das machen z. B. Systeme wie **HOL LIGHT** oder **METAMATH**. Aktuell scheinen jedoch die mit abhängigen Typen arbeitenden Systeme die Nase vorn zu haben, wenn es um den Zuspruch in der Mathematikerzunft geht.

Typs: $(= T x_1 x_2)$ steht für die Behauptung, dass x und y derselbe Ausdruck vom Typ T seien. Sie entsinnen sich hoffentlich, dass wir das als Urteil (J3) bezeichnet hatten. Nur war (J3) bisher eine externe Aussage, mit der wir *über* das System sprachen. Durch diesen Typ wird daraus ein Objekt der Sprache. (Man nennt das die **Internalisierung** des Urteils.)

Wie sehen die „Beweise“ für solche Urteile aus? Der Konstruktor zu diesem neuen Typ heißt `same`.

Ist x vom Typ T , so ist $(\text{same } x)$ ein Ausdruck vom Typ $(= T x x)$.

`same`

Es ist auf den ersten Blick verwirrend, dass in der Definition des Typs *zwei* Ausdrücke (x_1 und x_2) vorkommen im Konstruktor aber nur einer, nämlich x . Andererseits ist es aber wiederum logisch, denn weil der Typ für Gleichheit steht, sollen x_1 und x_2 ja *derselbe* Ausdruck (vom Typ T) sein. Schauen wir uns ein Beispiel an und überlegen wir uns dabei, was das System macht.³³

```
> (the (= Nat (add2 40) ((+ 2) 40)) (same 42))
(the (= Nat 42 42)
    (same 42))
```

Die Antwort sieht nicht besonders spektakulär aus. Man könnte sie lesen als den Satz: „Die Zahl 42 ist ein Beweis dafür, dass 42 dasselbe wie 42 ist.“ Diese scheinbare Banalität liegt jedoch daran, dass PIE den Ausdruck, der den Typ beschreibt, für die Ausgabe bereits in Normalform gebracht hat. Wichtiger ist, was wir eingegeben haben. Das System bekommt den Ausdruck $(\text{same } 42)$, der per definitionem vom Typ $(= \text{Nat } 42 \ 42)$ ist, und muss prüfen, ob das derselbe Typ ist, den wir mit `the` angegeben haben. Dafür werden die Ausdrücke $(\text{add2 } 40)$ und $((+ \ 2) \ 40)$ in Normalform überführt und beide mit 42 verglichen – was zum Erfolg führt.

Hätten wir

```
(the (= Nat (add2 40) ((+ 2) 40)) (same 43))
```

eingegeben, so hätte es offensichtlich nicht geklappt. Man kann für den „Beweis“ nicht *irgendeinen* Ausdruck nehmen. Wichtiger ist jedoch, dass die Variante

```
(the (= Nat (add2 40) ((+ 2) 41)) (same ???))
```

nie klappen würde – was auch immer wir für `???` eingeben. Weil die beiden Ausdrücke, die den Typ definieren, *nicht* gleich sind, ist es unmöglich, einen Ausdruck dieses Typs anzugeben.

OK, schön und gut, aber auf den ersten Blick sind wir nicht weiter, als wir mit `check-same` im [letzten Abschnitt](#) bereits waren. Wir könnten statt 40 andere Zahlen einsetzen und würden in jedem Einzelfall Erfolg haben. Wir möchten die

³³Wir setzen das Beispiel aus [dem letzten Abschnitt](#) fort.

Gleichheit jedoch für *alle* natürlichen Zahlen nachweisen. Der Fortschritt ist, dass ein Typ uns diese Gelegenheit gibt, und zwar so:

```
(claim add2=+2 (Pi ((n Nat)) (= Nat (add2 n) ((+ 2) n))))
```

Die Funktion `add2=+2`, deren Typ hier angegeben ist, ordnet jeder natürlichen Zahl n einen Ausdruck des Typs `(= Nat (add2 n) ((+ 2) n))` zu. So etwas wie

`(PI ((n Nat)) ...`

entspricht also dem **Allquantor**³⁴

$\forall n \in \mathbb{N} \dots$

Wenn es uns gelingt, so eine Funktion zu schreiben, die von `PIE` akzeptiert wird, dann hat sich das System überzeugt, dass es sich um eine *totale Funktion* handelt, die wirklich das tut, was sie tun soll! Diese Funktion wäre also ein Beweis für ihren Typ und damit für die Aussage, dass `(add2 n)` tatsächlich derselbe Ausdruck wie `((+ 2) n)` ist, wenn n irgendein `Nat`-Ausdruck ist!

Aufgabe 20.1. Wenn man im Sinne der Curry-Howard-Korrespondenz einen Ausdruck wie `(= Nat (add2 40) ((+ 2) 40))` mit einer Aussage identifiziert, wie würde man dann mit den Begrifflichkeiten [aus der Mathematik-Vorlesung](#) einen Ausdruck wie `(= Nat (add2 n) ((+ 2) n))` nennen?

Es fehlt nur noch die Funktion `add2=+2`. Es liegt nahe, sie mithilfe von `ind-Nat` (siehe Aufgabe 18.5) zu schreiben. Das Gerüst dafür sieht so aus:

```
(define add2=+2
  (lambda (n)
    (ind-Nat n
      (lambda (k) (= Nat (add2 k) ((+ 2) k))) ;; motive
      TODO                                     ;; base
      TODO)))                                ;; step
```

Die **Zieltypfunktion** (`motive`) liegt auf der Hand. Was noch fehlt, sind der Rekursionsanfang und der Rekursionsschritt. `PIE` bietet das nette Feature, dass man einzelne Ausdrücke, bei denen man sich noch nicht sicher ist, durch `TODO` ersetzen kann. Das System sagt uns dann, welchen Typ der fehlende Ausdruck haben muss.³⁵

Aufgabe 20.2. Das wäre eine gute Gelegenheit, es erst einmal selbst zu probieren, bevor Sie weiterlesen. Was muss an den beiden Stellen stehen, wo etwas fehlt? Zumindest für die erste Stelle (`base`) sollte das vergleichsweise einfach sein.

³⁴Den kennen Sie aus dem Abschnitt über [Prädikatenlogik](#) im Modul *Mathematik 1*.

³⁵Das funktioniert allerdings nur, wenn `PIE` die Typen herleiten kann. Ohne die `motive`-Zeile klappt es in diesem Fall nicht.

Für das zweite TODO (die `step`-Funktion) zeigt uns PIE den folgenden Typ an und verrät damit eigentlich schon die Lösung:³⁶

```
(Pi ((n-1 Nat))
  (-> (= Nat
    (add1 (add1 n-1))
    (iter-Nat n-1)
    (the Nat 2)
    (lambda (step-1)
      (add1 step-1))))
  (= Nat
    (add1 (add1 (add1 n-1)))
    (add1 (iter-Nat n-1)
      (the Nat 2)
      (lambda (step-1)
        (add1 step-1))))))
```

Gehen wir das im Einzelnen durch: Es gibt bereits einen Beweis dafür, dass die Ausdrücke $(\text{add2 } n-1)$ und $((+ 2) n-1)$ gleich sind.³⁷ Der Job der gesuchten Funktion ist es, daraus einen Beweis für die Terme zu konstruieren, die man erhält, wenn man jeweils $n-1$ durch $(\text{add1 } n-1)$ ersetzt. PIE zeigt netterweise bereits die so weit wie möglich vereinfachten Ausdrücke an, die wir [im letzten Abschnitt](#) untersucht hatten. Man kann das so lesen: Die `step`-Funktion muss einen Beweis dafür liefern, dass $(\text{add1 } (\text{add2 } n-1))$ und $(\text{add1 } ((+ 2) n-1))$ ebenfalls gleich sind.

Aufgabe 20.3. Überzeugen Sie sich, dass $((+ 2) (\text{add1 } n-1))$ tatsächlich die Form $(\text{add1 } ((+ 2) n-1))$ hat, die von PIE angezeigt wird.

Das ist jedoch etwas, was mathematisch völlig klar ist: Wenn ich weiß, dass $x = y$ gilt und wenn ich außerdem weiß, dass f eine [Funktion](#) ist, dann folgt zwangsläufig, dass auch $f(x) = f(y)$ gelten muss. Man nennt das in der Logik das **Leibniz-Gesetz**.

Aufgabe 20.4. Was sind in unserem Fall x , y und f ?

Für das Leibniz-Gesetz gibt es einen [Eliminator](#):³⁸

Ist E ein Ausdruck vom Typ $(= T x_1 x_2)$ und f eine Funktion von T zum Typ S , dann ist $(\text{cong } E f)$ ein Ausdruck vom Typ $(= S (f x_1) (f x_2))$.

`cong`

Aufgabe 20.5. Überlegen Sie erst, was PIE antworten sollte, wenn Sie den folgenden Ausdruck eingeben, und vergleichen Sie dann das Ergebnis mit Ihrer Prognose.

³⁶Vergleichen Sie die farblich hervorgehobenen Teile. Siehe auch den Dialog auf Seite [87](#).

³⁷Denken Sie daran, dass $n-1$ nur ein Name ist!

³⁸`cong` steht dabei für *Kongruenz*.

```
(cong (the (= Nat (add1 4) (add1 (add1 3)))
  (same 5))
  (the (-> Nat (Pair Nat Nat))
    (lambda (n) (cons n (add1 n))))))
```

Im Prinzip ist nun klar, was zu tun ist. Der einzige kleine Haken, der noch bleibt: `add1` ist – siehe Aufgabe 17.4 – keine Funktion. Das lässt sich aber leicht dadurch beheben, dass wir eine Funktion um `add1` „herumwickeln“. Dann sieht unser kompletter Beweis so aus:³⁹

```
(claim incr (-> Nat Nat))

(define incr (lambda (n) (add1 n)))

(claim add2=+2 (Pi ((n Nat)) (= Nat (add2 n) ((+ 2) n))))

(define add2=+2
  (lambda (n)
    (ind-Nat n
      (lambda (k) (= Nat (add2 k) ((+ 2) k)))
      (same 2)
      (lambda (n-1 almost) (cong almost incr))))))
```

Falls Sie noch nie davon gehört haben sollten: Sie haben gerade einen Beweis durch **vollständige Induktion** gesehen.

Der „Witz“ an dieser Vorgehensweise ist, dass wir Funktionen wie `add2=+2` nicht (nur) – wie in anderen Programmiersprachen – schreiben, um sie zu anzuwenden, sondern dass es uns in erster Linie darum geht, dass man sie überhaupt schreiben *darf*. Wenn der **Compiler** nicht meckert, haben wir etwas bewiesen. Das ist gemeint, wenn man über Typensysteme manchmal den Satz „*Programs are proofs*“ liest.

Aufgabe 20.6. Da wir sogar simple Funktionen wie `+` und `*` selbst definiert haben, können wir nicht einfach davon ausgehen, dass diese die Eigenschaften haben, die wir erwarten. Wir müssen sie stattdessen beweisen. Nehmen wir als Beispiel die **Assoziativität** der Addition. Wie müsste ein Typ aussehen, der behauptet, dass unser Operator `+` aus Aufgabe 17.5 assoziativ ist?

Aufgabe 20.7. Fortsetzung von Aufgabe 20.6: Geben Sie nun einen Beweis für den Typen

```
(claim +-assoc
  (Pi ((a Nat) (b Nat) (c Nat))
    (= Nat (+ a (+ b c)) (+ (+ a b) c))))
```

³⁹Das, was bisher `step-1` hieß, habe ich hier `almost` genannt, um deutlich zu machen, dass es „fast“ der Beweis ist. Wir müssen ihn nur noch von $n-1$ auf n übertragen.

an. Aufgrund der Art und Weise, wie wir $+$ definiert haben, bietet sich Induktion über c an. Das Gerüst ist hier schon einmal vorgegeben. Sie müssen noch – analog zum obigen Beispiel – die beiden `TODO`-Ausdrücke ersetzen.

```
(define +-assoc
  (lambda (a b c)
    (ind-Nat c
      (lambda (c)
        (= Nat (+ a (+ b c)) (+ (+ a b) c)))
      TODO
      TODO)))
```

Aufgabe 20.8. Geben Sie einen Beweis dafür an, dass $(+ 0 n)$ immer dieselbe natürliche Zahl wie n ist.

Aufgabe 20.9. Geben Sie einen Beweis dafür an, dass $(+ (\text{add1 } a) b)$ immer dieselbe natürliche Zahl wie $(\text{add1 } (+ a b))$ ist.

Nach dieser Vorarbeit können wir uns nun an einen Beweis für die **Kommutativität** der Addition heranwagen. Unser Ziel sieht so aus:

```
(claim +-comm
  (Pi ((a Nat) (b Nat))
    (= Nat (+ a b) (+ b a))))
```

Wir können wieder mit einem Gerüst anfangen:

```
(define +-comm
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ a b) (+ b a)))
      TODO
      TODO)))
```

Im ersten Fall (b ist `zero`) brauchen wir einen Beweis für $(= \text{Nat } (+ a 0) (+ 0 a))$. Nach Definition von $+$ ist $(+ a 0)$ einfach a ; wir müssen somit $(= \text{Nat } a (+ 0 a))$ beweisen. Mit der Funktion aus Aufgabe 20.8 liefert $(\text{zero} \rightarrow + a)$ einen Beweis für den Typ $(= \text{Nat } (+ 0 a) a)$. Das sieht mathematisch nach derselben Aussage aus, aber innerhalb von PIE ist es *nicht* derselbe Typ. Zum Glück „weiß“ das System, dass Gleichheit **symmetrisch** ist, und hat dafür einen **Eliminator**, der die Rolle der beiden Ausdrücke in einem $=$ -Typ vertauscht:

Ist E ein Ausdruck vom Typ $(= T x_1 x_2)$, dann ist $(\text{symm } E)$ ein Ausdruck vom Typ $(= T x_2 x_1)$.

`symm`

Damit können wir die erste Lücke füllen:

```
(define +-comm
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ a b) (+ b a)))
      (symm (zero-+ a))
      TODO)))
```

Für die zweite Lücke sind wir in der folgenden Situation:

- (i) Wir haben einen Beweis für $(= \text{Nat } (+ a b) (+ b a))$. Nennen wir ihn wieder *almost*.
- (ii) Wir brauchen einen Beweis für $(= \text{Nat } (+ a (\text{add1 } b)) (+ (\text{add1 } b) a))$ und sollen *almost* in einen solchen umformen.
- (iii) Nach Definition der Funktion $+$ hat der benötigte Typ die vereinfachte Form $(= \text{Nat } (\text{add1 } (+ a b)) (+ (\text{add1 } b) a))$.
- (iv) Mit der Anwendung $(\text{add1-+ } b a)$ aus Aufgabe 20.9 können wir einen Beweis für den Typ $(= \text{Nat } (+ (\text{add1 } b) a) (\text{add1 } (+ b a)))$ erzeugen.
- (v) $(\text{symm } (\text{add1-+ } b a))$ liefert daher einen Beweis für den „umgekehrten“ Typ $(= \text{Nat } (\text{add1 } (+ b a)) (+ (\text{add1 } b) a))$.
- (vi) Ferner liefert die Anwendung $(\text{cong almost incr})$ des Leibniz-Gesetzes einen Beweis für $(= \text{Nat } (\text{add1 } (+ a b)) (\text{add1 } (+ b a)))$.

Wie vorhin sind wir aus mathematischer Sicht fertig. (vi) und (v) sind Aussagen der Form $x = y$ bzw. $y = z$, aus denen $x = z$ folgt.

Aufgabe 20.10. Was wären im konkreten Fall x , y und z ?

Das ist die sogenannte **Transitivität** der Gleichheit. Wie Sie sich schon gedacht haben, fällt auch dafür ein Eliminator **vom Himmel**.

trans Ist E_1 ein Ausdruck vom Typ $(= T x_1 x_2)$ und E_2 einer vom Typ $(= T x_2 x_3)$, dann ist $(\text{trans } E_1 E_2)$ ein Ausdruck vom Typ $(= T x_1 x_3)$.

Jetzt haben wir alles, um den Beweis zusammenzusetzen:

```
(define +-comm
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ a b) (+ b a)))
      (symm (zero-+ a))
      (lambda (b-1 almost)
        (trans (cong almost incr)
              (symm (add1-+ b-1 a)))))))
```

Damit Sie auch einmal ein etwas weniger triviales Beispiel gesehen haben, finden Sie unter der URL <https://weitz.de/files/gauss.pie> einen vollständigen Beweis für die **gaußsche Summenformel**, in dem nur Mittel eingesetzt werden, die

bereits besprochen wurden. Wie Sie sehen, ist es sogar für solche „Selbstverständlichkeiten“, die man an der Tafel in zwei Minuten erklären kann, vergleichsweise mühsam, sie formal zu beweisen. Die gute Nachricht ist, dass man so etwas nur einmal machen muss. Ein einmal geführter Beweis kann archiviert und später als Baustein für komplexere Beweise verwendet werden. Ein Beispiel für so ein Archiv ist die umfangreiche Bibliothek [MATHLIB4](#) für [LEAN](#).

★**Aufgabe 20.11.** Schreiben Sie einen Beweis für diesen Typ:

```
(claim twice (Pi ((n Nat))
                  (= Nat (add1 (add1 (* n 2))) (* (add1 n) 2))))
```

Gehen Sie dabei folgendermaßen vor:

- (i) Lassen Sie sich von PIE den obigen Typ in vereinfachter Form anzeigen.
- (ii) Überlegen Sie sich, was $(\text{add1} \rightarrow + \text{ n n})$ aus Aufgabe 20.9 liefert.
- (iii) Wandeln Sie das Ergebnis aus dem letzten Schritt mit `cong` und `symm` passend um. Das reicht schon, Rekursion ist hier nicht nötig.

Noch ein letzter Punkt zur Gleichheit, die auch für die Typen selbst eine Rolle spielen kann. Nehmen wir dazu an, wir wollten die Funktion `list→vec` aus Abschnitt 18 etwas abwandeln, so dass im Vektor jedes Element der Liste doppelt vorkommt. Das sollte scheinbar mit ein paar simplen Anpassungen erledigt sein:

```
(claim list→vec*2 (Pi ((T U) (L (List T)))
                      (Vec T (* (length T L) 2)))) ;; geändert

(define list→vec*2
  (lambda (T list)
    (ind-List list
      (lambda (lst) (Vec T (* (length T lst) 2))) ;; geändert
      vecnil
      (lambda (first rest previous)
        (vec:: first (vec:: first previous)))))) ;; geändert
```

PIE wird das jedoch nicht akzeptieren. Was passiert hier? Im Rekursionsschritt in der letzten Zeile wird ein Vektor erzeugt. Bezeichnen wir die Länge der Liste `rest` mit k , so besagt einerseits die Zieltypfunktion, dass `previous` die Länge $(* k 2)$ hat und der neue Vektor die Länge $(* (\text{add1 } k) 2)$ haben soll. Andererseits „weiß“ PIE, dass sich durch das Hinzufügen von zwei Elementen zu `previous` die Länge $(\text{add1 } (\text{add1 } (* k 2)))$ ergeben muss. Da PIE aber nicht *rechnet*, ergeben sich aus Sicht des System zwei sich widersprechende Typen.

Zum Glück gibt es auch dafür eine Lösung:

Sei T ein Typ, f eine Funktion von T in das [Universum \$U\$](#) , E ein Ausdruck vom Typ $(= T x_1 x_2)$ und X vom Typ $(f x_1)$. Dann ist `(replace  $E f X$ )` der Ausdruck X mit dem Typ $(f x_2)$.

replace

Was genau macht `replace`? Der Typ des Ausdrucks X wird geändert von $(f\ x_1)$ zu $(f\ x_2)$. Mit der Sprechweise der Typentheorie, in der man nur über Gleichheit sprechen kann, wenn die beiden verglichenen Ausdrücke denselben Typ haben, ist es allerdings schwer zu erklären, was sich *nicht* ändert, obwohl es intuitiv hoffentlich klar ist. X als Objekt, das in irgendeiner Form im Computer gespeichert ist, bleibt dasselbe; z. B. wird ein Vektor mit den Elementen 23 und 42 auch danach ein Vektor mit den Elementen 23 und 42 sein. Aus der Sicht des Systems handelt es sich nach einer Änderung des Typs jedoch um einen *anderen* Ausdruck.

Welche Bedeutung haben die anderen Argumente für `replace`? Die Funktion f stellt die Typinformation in Abhängigkeit von einem Ausdruck vom Typ T her, es handelt sich also um die **Zieltypfunktion**. Der Ausdruck E garantiert, dass x_1 und x_2 austauschbare Ausdrücke vom Typ T sind. Das System prüft, ob X tatsächlich den Typ $(f\ x_1)$ hat, und führt dann den Typwechsel durch, den E gewissermaßen „erlaubt“.

Damit können wir `list->vec*2` schreiben, wobei `twice` aus Aufgabe 20.11 die Rolle von E einnimmt.

```
(define list->vec*2
  (lambda (T list)
    (ind-List list
      (lambda (lst) (Vec T (* (length T lst) 2)))
      vecnil
      (lambda (first rest previous)
        (replace (twice (length T rest))
          (lambda (n) (Vec T n))
          (vec:: first (vec:: first previous)))))))
```

Der in der letzten Zeile erzeugte Vektor hat – wie oben erklärt – nach Konstruktion den Typ $(\text{Vec } T\ (\text{add1}\ (\text{add1}\ (*\ k\ 2))))$. `replace` „überzeugt“ das System jedoch davon, dass er den Typ $(\text{Vec } T\ (*\ (\text{add1}\ k)\ 2))$ hat.

Tatsächlich ist `replace` so aussagekräftig, dass man auf die Eliminatoren `symm`, `trans` und `cong`, die wir in diesem Abschnitt kennengelernt haben, verzichten könnte. `symm` könnte man beispielsweise so ersetzen:

```
(claim my-symm
  (Pi ((A U) (from A) (to A))
    (-> (= A from to) (= A to from))))

(define my-symm
  (lambda (A from to E)
    (replace E
      (lambda (x) (= A x from))
      (same from)))))
```

Damit könnten wir `+-comm` **von oben** wie folgt schreiben:

```
(define +-comm
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ a b) (+ b a)))
      (my-symm Nat (+ 0 a) (+ a 0) (zero-+ a))
      (lambda (b-1 almost)
        (trans (cong almost incr)
          (my-symm Nat (+ (add1 b-1) a) (add1 (+ b-1 a))
            (add1-+ b-1 a))))))))
```

Wie bereits nach der [Einführung von Pi](#) angemerkt wurde, wird dadurch alles umständlicher. Aber – wie gesagt – in Systemen, die in der Praxis eingesetzt werden, werden Sie solche Probleme eher nicht haben.

★**Aufgabe 20.12.** Schreiben Sie analog mithilfe von replace Eliminatoren `my-trans` und `my-cong`, die `trans` und `cong` ersetzen könnten. `+-comm` sollte dann so geschrieben werden können:

```
(define +-comm
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ a b) (+ b a)))
      (my-symm Nat (+ 0 a) (+ a 0)
        (zero-+ a))
      (lambda (b-1 almost)
        (my-trans Nat (add1 (+ a b-1))
          (add1 (+ b-1 a))
          (+ (add1 b-1) a)
          (my-cong Nat Nat (+ a b-1) (+ b-1 a)
            almost incr)
          (my-symm Nat (+ (add1 b-1) a)
            (add1 (+ b-1 a))
            (add1-+ b-1 a))))))))
```

21. Logik in PIE

Im letzten Abschnitt haben wir durch kleine Beweise einen ersten Eindruck von der [Curry-Howard-Korrespondenz](#) bekommen. Insbesondere haben wir gesehen, dass und wie man eine [PIE-Funktion](#) als [Allquantor](#) aus der [Prädikatenlogik](#) interpretieren kann. Will man beliebige Beweise „nachbauen“, so braucht man jedoch auch den [Existenzquantor](#) sowie [Junktoren](#) aus der [Aussagenlogik](#), die sich so verhalten, wie man es aus der [mathematischen Logik](#) gewohnt ist. Zwei Junktoren haben wir sogar schon, wir haben sie bloß bisher nicht in diesem Sinne verwendet.

Rufen wir uns dafür noch einmal ins Gedächtnis, dass man unter der besagten Korrespondenz einen Typ T als Behauptung interpretiert und einen Term x von diesem Typ als Beweis für T . Welche Bedeutung hat dann eine Funktion f vom Typ $(\rightarrow T_1 T_2)$? Wenn das System sie akzeptiert, dann ist sie nachweislich *total*, wird

also jeden T_1 -Ausdruck in einen T_2 -Ausdruck umwandeln. Mit anderen Worten: Gibt es einen Beweis für T_1 , dann gibt es auch einen für T_2 . f ist also ein Beweis dafür, dass aus der Wahrheit von T_1 die Wahrheit von T_2 folgt. Das ist eine **Implikation**, für die man in der Logik oft $\Rightarrow$ schreibt. (Dadurch kann man es sich auch ganz gut merken, weil $\rightarrow$ und $\Rightarrow$ ähnlich aussehen.)

Aufgabe 21.1. In der Logik gibt es bereits seit der Antike die Schlussfigur *Modus ponens*, die das Folgende besagt: Hat man Beweise für die Implikation $A \Rightarrow B$ und für die Aussage A , dann hat man auch einen Beweis für die Aussage B . In PIE würde das so aussehen:

```
(claim mp (Pi ((T1 U) (T2 U))
               (-> (-> T1 T2) T1 T2)))
```

Geben Sie einen Beweis für mp an.

Aufgabe 21.2. Wie steht es mit der **Konjunktion**? In der Logik ist das Zeichen dafür $\wedge$, in Programmiersprachen oft `&&` – oder auch einfach `and` wie in SCHEME oder PYTHON. „Übersetzt“ müsste das bedeuten: Ein Typ, für den man nur dann einen Ausdruck angeben kann, wenn man einen Ausdruck vom Typ T_1 *und* einen vom Typ T_2 angeben kann. Hatten wir so etwas nicht schon?

Aufgabe 21.3. In der Aussagenlogik ist beispielsweise $A \wedge B \Rightarrow B \wedge A$ eine **Tautologie** (die besagt, dass die Konjunktion **kommutativ** ist). Formulieren Sie diese Aussage mithilfe von Aufgabe 21.2 in PIE und geben Sie einen Beweis dafür an.

Aufgabe 21.4. Beweisen Sie analog zu Aufgabe 21.3, dass die Konjunktion auch **assoziativ** ist. Es geht also um die Tautologie $(A \wedge B) \wedge C \Rightarrow A \wedge (B \wedge C)$.

Für die **Disjunktion**⁴⁰ hat PIE einen Typ, den wir bisher nicht verwendet haben.

Either
left right

Sind T_1 und T_2 Typen, dann ist $(\text{Either } T_1 T_2)$ ein Typ. Sind x_1 bzw. x_2 Ausdrücke vom Typ T_1 bzw. T_2 , dann sind sowohl $(\text{left } x_1)$ als auch $(\text{right } x_2)$ Ausdrücke vom Typ $(\text{Either } T_1 T_2)$.

Die **Konstruktoren** für diesen Typ sind also `left` und `right` und anders als bei `Pair` reicht es, wenn man für einen der beiden Typen T_1 und T_2 einen Ausdruck angeben kann, um einen Ausdruck vom Typ $(\text{Either } T_1 T_2)$ zu kreieren. Das entspricht genau der Semantik der Disjunktion. Hier sind zwei Möglichkeiten, so einen Ausdruck zu erhalten:

```
(the (Either Nat Atom) (left 23))
(the (Either Nat Atom) (right 'quark))
```

Aufgabe 21.5. Beweisen Sie die **Tautologie** $A \wedge B \Rightarrow A \vee B$.

⁴⁰Also $\vee$ in der Mathematik bzw. `||` in Sprachen wie JAVA, die die C-Syntax kopieren.

An die einzelnen Komponenten eines Paares kommt man mit den **Eliminatoren** `car` und `cdr`. So ein direkter Zugriff ist mit `Either` nicht möglich, weil wir, wenn wir es mit einem solchen Ausdruck zu tun haben, gegebenenfalls nicht wissen, ob er mit `left` oder mit `right` erzeugt wurde. Der Eliminator muss also anders aussehen.⁴¹

`ind-Either` ist eine Funktion mit vier Argumenten `target`, `motive`, `on-left` und `on-right`. `target` ist dabei vom Typ $(\text{Either } T_1 \ T_2)$ und `motive` ist eine Funktion vom Typ $(\rightarrow (\text{Either } T_1 \ T_2) \ U)$.⁴² Ist `target` der Ausdruck $(\text{left } x_1)$, dann ist

$$(\text{ind-Either target motive on-left on-right}) \quad (21.1)$$

der Ausdruck $(\text{on-left } x_1)$. Ist `target` der Ausdruck $(\text{right } x_2)$, dann ist (21.1) der Ausdruck $(\text{on-right } x_2)$. In beiden Fällen ist (21.1) vom Typ (motive target) .

Die Funktion `on-left` hat den Typ $(\text{Pi } ((x_1 \ T_1)) (\text{motive } (\text{left } x_1)))$ und `on-right` hat den Typ $(\text{Pi } ((x_2 \ T_2)) (\text{motive } (\text{right } x_2)))$.

`ind-Either`

Ganz grob kann man `ind-Either` mit `cond` in SCHEME vergleichen, wobei es allerdings immer nur zwei Fälle gibt und `else` garantiert nicht benötigt wird. Hier ein einfaches Beispiel für die Verwendung von `ind-Either`:

```
(claim either-example
  (-> (Either Nat Atom) (Pair Nat Atom)))

(define either-example
  (lambda (E)
    (ind-Either E
      (lambda (E) (Pair Nat Atom))
      (lambda (L) (cons L 'whatever))
      (lambda (R) (cons 42 R)))))
```

Diese Funktion „vervollständigt“ quasi ein `Either` zu einem `Pair`, indem es die eine Hälfte erhält und die andere mit einem willkürlichen Wert auffüllt.

```
> (either-example (left 23))
(the (Pair Nat Atom)
  (cons 23 'whatever))
> (either-example (right 'quark))
(the (Pair Nat Atom)
  (cons 42 'quark))
```

Aufgabe 21.6. Beweisen Sie mithilfe von `ind-Either`, dass $\vee$ kommutativ ist.

⁴¹Man vergleiche mit Aufgabe 18.1 sowie mit `which-Nat`.

⁴²Wie üblich stellt `motive` die **Zieltypfunktion** zur Verfügung.

Aufgabe 21.7. Beweisen Sie mithilfe von `ind-Either`, dass `v` assoziativ ist.

★**Aufgabe 21.8.** Warum ist die folgende Definition nicht zulässig?

```
(claim payload-type
  (-> (Either Nat Atom) U))

(define payload-type
  (lambda (E)
    (ind-Either E
      (lambda (E) U)
      (lambda (L) Nat)
      (lambda (R) Atom)))))
```

★**Aufgabe 21.9.** Bei den bisherigen Beispielen mit `ind-Either` war die **Zieltypfunktion** immer konstant. Fällt Ihnen ein Beispiel ein, bei dem das nicht so ist? (Hinweis: Versuchen Sie so etwas wie in Aufgabe 21.8, bei dem das Ergebnis ein `=-Typ` ist.)

Es fehlt noch die **Negation**. Dafür benötigen wir zunächst einmal so etwas wie **Wahrheitswerte**.

Trivial sole

Absurd

Trivial ist ein Typ, für den es nur den Konstruktor `sole` gibt. Alle Ausdrücke vom Typ `Trivial` sind gleich.

Absurd ist ein Typ, für den es keinen Konstruktor gibt.

Trivial spielt also die Rolle einer Aussage, die garantiert wahr ist, weil man sie durch `sole` beweisen kann.⁴³ Absurd steht für eine Aussage, die garantiert falsch ist – weil es ohne Konstruktor unmöglich ist, einen Beweis anzugeben.

Damit können wir Negation im Sinne der Curry-Howard-Korrespondenz formulieren. In der **Aussagenlogik** ist $\neg A$ äquivalent zu $A \Rightarrow \perp$, wenn $\perp$ für eine **Kontradiktion** steht. Übersetzt bedeutet das: Ist T ein Typ, so ist $(\rightarrow T \text{ Absurd})$ die Aussage, dass T nicht wahr ist. Wäre T nämlich wahr, so könnte man durch eine Funktion dieses Typs aus einem Beweis für T einen Beweis für Absurd machen – den es nicht geben kann.

Damit haben wir alles zusammen, was wir für die Aussagenlogik benötigen:⁴⁴

⁴³Natürlich ist auch `Nat` garantiert wahr, weil man als Beweis `zero` nehmen könnte. Man könnte als Beweis aber auch `42` nehmen. In diesem Sinne enthält ein Beweis von `Nat` „zu viel Information“. Ein Wahrheitswert sollte nur ein **Bit** (eigentlich ein **Shannon**) an Information liefern. Ein Beweis von `Trivial` sagt nichts weiter aus als: die Aussage ist wahr.

⁴⁴ $\top$ ist das übliche Zeichen für eine **Tautologie**.

Aussage	Typ
A_k	T_k
$\top$	Trivial
$\perp$	Absurd
$A_1 \wedge A_2$	(Pair $T_1 T_2$)
$A_1 \vee A_2$	(Either $T_1 T_2$)
$A_1 \Rightarrow A_2$	($\rightarrow T_1 T_2$)
$\neg A_1$	($\rightarrow T_1$ Absurd)

Aufgabe 21.10. Formulieren und beweisen Sie in PIE die Tautologie $A \Rightarrow \neg \neg A$.

Obwohl es keinen Ausdruck des Typs Absurd gibt, gibt es dafür einen **Eliminator**:

Ist x vom Typ Absurd und T ein Typ, dann ist $(\text{ind-Absurd } x \ T)$ ein Ausdruck vom Typ T .

ind-Absurd

Klingt absurd, oder? So einen Ausdruck x kann es doch gar nicht geben! Dahinter steckt jedoch die Tautologie $\perp \Rightarrow A$, die unter der Bezeichnung *ex falso quodlibet* („aus Falschem folgt Beliebiges“) bekannt ist: Aus einer falschen Aussage kann man alles folgern.⁴⁵ Und in der Tat braucht man das manchmal.

Aufgabe 21.11. Formulieren und beweisen Sie mithilfe von ind-Absurd die Tautologie $B \vee \neg A \Rightarrow (A \Rightarrow B)$.

Aufgabe 21.12. Formulieren und beweisen Sie die Tautologie $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$.

Nachdem wir die wesentlichen Elemente der Aussagenlogik nun zusammen haben, fehlt nur noch der am Anfang des Abschnitts erwähnte Existenzquantor. Dafür führen wir einen letzten Typ ein.

Ist x eine Variable, T ein Typ und S ein Ausdruck, dann ist $(\text{Sigma } ((x \ T)) \ S)$ ein Typ. Ausdrücke dieses Typs sind Paare, die mit $(\text{cons } a \ d)$ erstellt werden können, wenn a den Typ T und d den Typ $S[x \leftarrow a]$ hat.

Sigma

Es handelt sich somit um einen **abhängigen Typen**, bei dem der Typ der zweiten Komponente von der ersten abhängt. Den Konstruktor kennen wir schon. Ein Beispiel:

```
(claim Nat-Vec U)

(define Nat-Vec
  (Sigma ((n Nat)) (Vec Nat n)))
```

Man kann nun mit Ausdrücken wie

⁴⁵Wenn der Mond aus Käse besteht, dann ist die Erde eine Scheibe.

```
(the Nat-Vec (cons 0 vecnil))
(the Nat-Vec (cons 1 (vec:: 23 vecnil)))
(the Nat-Vec (cons 2 (vec:: 42 (vec:: 23 vecnil))))
```

Objekte dieses Typs erzeugen. `(the Nat-Vec (cons 23 vecnil))` würde hingegen zu einer Fehlermeldung führen. Die erste Komponente des Paares schreibt der zweiten vor, wie lang sie zu sein hat.

Beachten Sie, dass auch das hier funktioniert hätte:

```
(the (Pair Nat (Vec Nat 0)) (cons 0 vecnil))
```

Das sollte die Frage aus Abschnitt 15 beantworten, warum PIE nicht automatisch den Typ von so etwas wie `(cons 0 vecnil)` ermitteln kann. So, wie $\rightarrow$ ein Spezialfall von `Pi` ist, ist jedenfalls `Pair` ein Spezialfall von `Sigma`; nämlich der, dass x in S nicht vorkommt (dass der Typ der zweiten Komponente also *nicht* von der ersten abhängt).

Um zu erklären, wieso wir damit einen Existenzquantor haben, definieren wir zunächst eine Hilfsfunktion:

```
(claim double (-> Nat Nat))

(define double
  (lambda (n)
    (iter-Nat n
      zero
      (lambda (n-1) (add1 (add1 n-1))))))
```

`double` verdoppelt offensichtlich sein Argument. Wir hätten das auch einfacher mit `(+ n n)` oder `(* n 2)` haben können, weil wir Addition und Multiplikation in Abschnitt 17 bereits definiert haben. Aber dann hätten wir wie bei `list->vec*2` in Abschnitt 20 kleinteilige Beweise über arithmetische Identitäten führen müssen, die uns vom Thema ablenken würden.

Nun definieren wir eine Funktion, die Typen zurückgibt.

```
(claim Even (-> Nat U))

(define Even
  (lambda (n)
    (Sigma ((k Nat))
      (= Nat n (double k)))))
```

Aufgabe 21.13. Geben Sie einen Ausdruck an, der den Typ `(Even 10)` hat.

Offensichtlich basiert der Typ `(Even n)` auf der mathematischen Definition, dass eine Zahl n *gerade* ist, wenn es eine Zahl k mit $n = 2k$ gibt.⁴⁶ Nur dann finden

⁴⁶Wir reden hier natürlich nur über *ganze Zahlen*.

wir ein Paar von diesem Typ, nämlich $(\text{cons } k \text{ (same } n))$. Der Typ ist also das Pendant zur mathematischen Aussage

$$\exists k \in \mathbb{N} (n = 2k)$$

und in diesem Sinne kann man Sigma als **Existenzquantor** einsetzen. Ein Paar, das den entsprechenden Typ hat, ist ein Beweis dafür, dass n eine gerade Zahl ist.

Aufgabe 21.14. Konstruieren Sie nach demselben Muster einen Typen für ungerade Zahlen.

Mit `Odd` aus der Lösung von Aufgabe 21.14 und `incr` aus Abschnitt 20 können wir nun die Aussage beweisen, dass der **Nachfolger** einer geraden Zahl ungerade ist:

```
(claim even-succ (Pi ((n Nat))
                     (-> (Even n) (Odd (add1 n)))))

(define even-succ
  (lambda (n S)
    (cons (car S)
          (cong (cdr S) incr))))
```

Unsere Funktion bekommt ein Paar S , das die Form

```
(cons k (= Nat n (double k)))
```

für ein k vom Typ `Nat` hat. Wir bauen daraus ein neues Paar mit derselben linken Komponente, erhöhen aber die beiden Seiten der Gleichheit in der rechten. Das ergibt dann

```
(cons k (= Nat (add1 n) (add1 (double k))))
```

und das ist $(\text{Odd } (\text{add1 } n))$.

Aufgabe 21.15. Konstruieren Sie nach demselben Muster einen Beweis dafür, dass der Nachfolger einer ungeraden Zahl gerade ist.

Aufgabe 21.16. Formulieren und beweisen Sie die Aussage, dass jede natürliche Zahl gerade oder ungerade ist. (Hinweis: Sie werden sowohl `ind-Nat` als auch `ind-Either` benötigen. Die gerade eben definierten Funktionen sind ebenfalls hilfreich.)

Aufgabe 21.17. In Abschnitt 16 ergab sich die Frage, wann Variablen in PIE gebunden bzw. frei sind. Da wir nun alle Typen dieser überschaubaren Sprache kennen, kann man die Frage beantworten. In welchen Ausdrücken können Variablen gebunden werden?

★Intuitionistische Logik

Erwähnenswert ist im Zusammenhang mit der mathematischen Logik, dass die Curry-Howard-Korrespondenz in Reinform auf etwas hinausläuft, das man *intuitionistische Logik* nennt: Damit eine Aussage A als wahr gilt, muss man einen Beweis für A erbringen; es reicht nicht, zu zeigen, dass $\neg A$ nicht möglich ist. Das steht im Gegensatz zur in der Mathematik eher üblichen *klassischen Logik*, in der solche „indirekten“ Beweisprinzipien häufig verwendet werden.

Beispielsweise sind die folgenden Aussagen, die in der klassischen Logik Tautologien sind, in der intuitionistischen Logik alle nicht beweisbar:

(i) $A \vee \neg A$

Das ist der sogenannten *Satz vom ausgeschlossenen Dritten*.

(ii) $\neg\neg A \Rightarrow A$

Das nennt man auch *Gesetz der doppelten Negation*. Für die umgekehrte Richtung siehe Aufgabe 21.10.

(iii) $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$

Das wird *Kontraposition* oder manchmal auch *indirekter Beweis* genannt. Für die umgekehrte Richtung siehe Aufgabe 21.12.

(iv) $(A \Rightarrow B) \Rightarrow B \vee \neg A$

Für die umgekehrte Richtung siehe Aufgabe 21.11.

Dass das nicht geht, kann man erkennen, wenn man eine dieser Aussagen in PIE zu beweisen versucht. Der Satz vom ausgeschlossenen Dritten würde z. B. diesen Typ haben:

```
(claim excl-middle
  (Pi ((T U)) (Either T (-> T Absurd))))
```

Diese Funktion müsste also für *jede* Aussage entweder einen Beweis erbringen oder ihre Negation beweisen können. Es liegt wohl auf der Hand, dass das nicht sein kann.

Große Teile der Mathematik verlassen sich jedoch auf dieses Prinzip. Darum ist es in professionellen Systemen wie LEAN auch als Hilfsmittel vorhanden, denn sonst könnte man mit denen nur sehr wenig Mathematik abbilden.

Wie man das realisieren kann, ohne deshalb die Curry-Howard-Korrespondenz aufzugeben, kann man mit einem kleinen Trick sogar in PIE zeigen. Dazu füge man die nicht machbare Implementation der obigen Funktion einfach als `TODO` ein:

```
(define excl-middle
  (lambda (T)
    TODO))
```

PIE „glaubt“ nun, dass wir das noch fertigstellen werden und akzeptiert die Funktion `excl-middle` im Folgenden unter dem angegebenen Typ. Nun können wir beispielsweise die doppelte Negation „beweisen“:

```
(claim double-neg*
  (Pi ((T U))
    (-> (-> (-> T Absurd) Absurd) T)))

(define double-neg*
  (lambda (T f)
    (ind-Either (excl-middle T)
      (lambda (E) T)
      (lambda (L) L)
      (lambda (R) (ind-Absurd (f R) T))))))
```

Und auch die Punkte (iii) und (iv) von oben folgen aus dem Satz vom ausgeschlossenen Dritten, wie die beiden Aufgaben unten zeigen. Dieser Satz ist gewissermaßen das „trojanische Pferd“, das aus der intuitionistischen Logik die klassische macht.

★**Aufgabe 21.18.** Beweisen Sie mithilfe von `excl-middle` das Prinzip der Kontraposition, also die umgekehrte Richtung von Aufgabe 21.12.

★**Aufgabe 21.19.** Beweisen Sie mithilfe von `excl-middle` die umgekehrte Richtung von Aufgabe 21.11.

IV

Ausblick: LEAN

Nachdem wir durch [PIE](#) nun eine Vorstellung davon haben, wie eine abhängige Typentheorie auf einem Computer implementiert werden kann, werfen wir zum Abschluss noch einen Blick auf ein professionelles System. Solche Systeme basieren auf denselben Prinzipien wie [PIE](#), sind aber wesentlich umfangreicher und leistungsfähiger und in der Regel auch angenehmer zu bedienen. Bekannte Beispiele sind [ROCQ](#) – das es bereits seit 1989 gibt und mit dem unter anderem der Beweis des [Vier-Farben-Satzes](#) formalisiert wurde – oder [AGDA](#).

Wir schauen uns in diesem kurzen Kapitel das schon mehrfach erwähnte System [LEAN](#) an. [LEAN](#) wurde ab 2013 von dem brasilianischen Informatiker [Leonardo de Moura](#) bei [Microsoft Research](#) entwickelt und ist gleichzeitig ein [Beweisassistent](#) und eine (funktionale) [Programmiersprache](#). Der ursprüngliche Zweck des Systems war die [Verifikation](#) von Software.¹ [LEAN](#) hat sich jedoch „nebenbei“ in den letzten Jahren zum wohl beliebtesten Beweisassistenten in der Forschungsmathematik entwickelt und wird zunehmend auch für die Weiterentwicklung [großer Sprachmodelle](#) eingesetzt.

Auf den folgenden Seiten soll in erster Linie an einfachen Beispielen gezeigt werden, wie sich [LEAN](#) und [PIE](#) unterscheiden.

Typ-Hierarchien

Während in [PIE](#) manche Ausdrücke wie [U](#) oder $(\text{Pair } U \text{ Atom})$ keinen Typ haben, hat in [LEAN](#) tatsächlich jeder Ausdruck einen Typ. Erreicht wird das mit der in Fußnote 6 auf Seite 69 skizzierten Hierarchie von „Universen“: Der Konstruktor `true` hat z. B. den Typ `Bool`, `Bool` hat den Typ `Type`, `Type` hat den Typ `Type 1`, `Type 1` hat den Typ `Type 2` und immer so weiter.

¹Der ebenfalls von de Moura geschriebene Vorgänger [Z3](#) von [LEAN](#) soll ein Drittel aller Fehler in [Windows 7](#) gefunden haben, die vor der Veröffentlichung behoben wurden. Mehr zur Geschichte von [LEAN](#) findet man in dem Buch [The Proof in the Code](#).

Syntax

Während PIE (wie auch [SCHEME](#) oder der [Lambda-Kalkül](#)) für alle Ausdrücke eine gleichförmige, oft umständliche Syntax verwendet, bietet LEAN an vielen Stellen "[syntaktischen Zucker](#)", damit Ausdrücke eher so aussehen, wie man sie auch sonst schreiben würde. Man kann beispielsweise $2+3=5$ schreiben und das System macht daraus intern `Eq (HAdd.hAdd 2 3) 5`. Auch mathematische Symbole wie $\forall$ oder $\mathbb{N}$ können verwendet werden. Die folgenden Beispielen zeigen teilweise weitere Beispiele für syntaktische Erleichterungen.

Optionale Argumente

Wir haben in Abschnitt 18 gelernt, wie man in PIE Funktionen schreiben kann, die mit verschiedenen Typen umgehen können. Leider mussten wir diese Typen immer auch explizit angeben. Nehmen wir als Beispiel die Funktion `length` aus Aufgabe 18.2, die man als `(length Nat ...)` aufrufen muss, wenn es sich um eine Liste von natürlichen Zahlen handelt, und als `(length Atom ...)` für eine Liste von Atomen.

In LEAN könnte man diese Funktion folgendermaßen definieren (wobei `List.rec` wie [ind-List](#) funktioniert):

```
def len {α : Type} (L : List α) : Nat :=
  List.rec
    0
    (fun head tail prev => prev + 1)
    L
```

In Lean gibt es wie in PIE für jeden Typ α einen eigenen Listentyp `List α` und dieser Typ wird auch in der Definition von `len` angegeben. Dass er in geschweiften Klammern steht, signalisiert dem System jedoch, dass er beim Aufruf der Funktion in der Regel weggelassen wird, weil er aus dem Kontext hervorgeht. Und so kann man dann auch einfach

```
len [3,4,5]
```

schreiben, ohne dass es eine Fehlermeldung gibt. Generell gibt es viele Situationen, in denen das System ohne Zutun des Benutzers Typen selbstständig herleitet.

Pattern Matching

Die Funktion `len` aus dem letzten Abschnitt kann man jedoch wesentlich eleganter mittels [Pattern Matching](#) definieren:

```
def len {α : Type} : List α -> Nat
| [] => 0
| _ :: tail => len tail + 1
```

Das ist offensichtlich viel knapper und besser zu lesen als die vorherige Version. Dieses Verfahren wird in LEAN an vielen Stellen verwendet, wie z. B. auch das folgende Beispiel zeigt.

Benutzerdefinierte Typen

In PIE kann man Typen wie `(Pair Atom Nat)` erzeugen, aber sie basieren auf Typen, die in PIE bereits vorhanden sind. In LEAN kann man hingegen komplett neue Typen generieren, mit eigenen Konstruktoren und allem Drum und Dran. Gäbe es `List` nicht bereits, so könnte man folgendermaßen einen Ersatz `Lst` mit den Konstruktoren `nil` und `cons` dafür basteln:

```
universe u

inductive Lst ( $\alpha$  : Type u) where
  | nil
  | cons (head :  $\alpha$ ) (tail : Lst  $\alpha$ )
```

LEAN würde sogar automatisch einen Eliminator `Lst.rec` bauen, den man wie im letzten Abschnitt verwenden könnte:

```
open Lst

def len { $\alpha$  : Type} : List  $\alpha$  -> Nat
  | nil => 0
  | (cons _ tail) => len tail + 1
```

Man hätte lediglich noch nicht die oben gesehene Syntax, mit der man `[3, 4, 5]` oder `3 :: [4, 5]` schreiben kann. Aber auch das kann man in LEAN realisieren.

Beweise

Da LEAN inzwischen hauptsächlich für mathematische Beweise eingesetzt wird, gibt es für diesen Zweck auch eine ganze Reihe von praktischen Hilfsmitteln, von denen ich zumindest ein paar anhand eines einfachen Beispiels vorstellen möchte. Wir nehmen als Vergleich den Beweis der [gaußschen Summenformel](#) von Seite 96.²

Zunächst definieren wir mit [Pattern Matching](#) die Funktion:

```
def gauss : Nat -> Nat
  | 0 => 0
  | n + 1 => gauss n + n + 1
```

Und dann ein klassischer Beweis durch [vollständige Induktion](#), in dem die einzelnen Umformungsschritte jeweils begründet werden.

²Natürlich ist der Beweis nicht nötig, weil LEAN diese Formel bereits „kennt“.

```

theorem gauss_twice (n : Nat) : gauss n * 2 = n * (n+1) := by
  induction n with
  | zero => rfl
  | succ n ih =>
    calc
      gauss (n+1) * 2 = (gauss n + n + 1) * 2 := rfl
      _ = (gauss n + (n+1)) * 2 := by rw [Nat.add_assoc]
      _ = gauss n * 2 + (n+1) * 2 := by rw [Nat.add_mul]
      _ = n * (n+1) + (n+1) * 2 := by rw [ih]
      _ = (n+1) * n + (n+1) * 2 := by rw [Nat.mul_comm n (n+1)]
      _ = (n+1) * (n+2) := by rw [← Nat.mul_add]

```

ih steht dabei für *induction hypothesis*, also die Induktionsvoraussetzung. `rfl` steht für die [Reflexivität](#) der [Gleichheit](#) und wäre in PIE so etwas wie das automatische Einsetzen von [same](#). Die anderen Umformungen beruhen auf Rechenregeln, die in LEAN schon vorhanden sind, wie z. B. `Nat.mul_comm` für die [Kommutativität](#) der Multiplikation.

Man beachte außerdem, dass Applikation in LEAN eine höhere [Priorität](#) als `+` oder `*` hat, so dass z. B. `gauss n * 2` dasselbe wie `(gauss n) * 2` ist.

Taktiken

Obwohl dieser kleine Beweis schon sehr nett anzusehen ist, wäre das für einen geübten Benutzer von Lean immer noch zu viel Arbeit. Mit sogenannten *Taktiken* kann man in vielen Fällen dem System die Arbeit überlassen, die richtigen Umformungen zu finden. Der komplette Beweis sieht dann so aus:

```

theorem gauss_twice (n : Nat) : gauss n * 2 = n * (n+1) := by
  induction n with
  | zero => rfl
  | succ n ih =>
    simp [gauss, Nat.mul_succ, Nat.succ_mul] at ih
    omega

```

Hier wird nicht mehr wie im letzten Beispiel Schritt für Schritt gerechnet, sondern dem System wird der Auftrag erteilt, zu beweisen, dass zwei Terme gleich sind, wobei es den Hinweis erhält, welche Rechengesetze für diesen Beweis eingesetzt werden sollen.

Taktiken sind also dafür da, dem Computer wenn möglich Routinearbeiten zu überlassen. (Das sind häufig die Stellen in Beweisen, bei denen man in Fachartikeln und Lehrbüchern Formulierungen wie „offensichtlich“ oder „trivial“ liest. Oder: „Den Beweis überlassen wir als Übungsaufgabe dem Leser.“)

Compiler

Wie eingangs erwähnt steckt im LEAN-System nicht nur eine Typentheorie, sondern auch eine ausgewachsene Programmiersprache. Den Unterschied kann man an

dem folgenden kleinen Beispiel erklären. LEAN hat wie [PIE](#) den Typ `Nat` und er ist im Prinzip genauso definiert (mit `succ` statt `add1`):

```
inductive Nat where
| zero : Nat
| succ (n : Nat) : Nat
```

Und wie in [PIE](#) ist so etwas wie 2 nur Oberflächensyntax für die interne Darstellung `succ (succ zero)`. Gibt man nun

```
#reduce 2+3
```

ein, so erhält man als Ergebnis wie erwartet 5. Intern ist LEAN dabei wie [PIE](#) vorgegangen: Wie der Name schon andeutet, wurde eine [Reduktion](#) durchgeführt, die auch die rekursive Definition von `+` einbezieht.

Man kann aber auch

```
#eval 2+3
```

eingeben. In diesem Fall macht LEAN das, was auch [SCHEME](#) oder [JAVA](#) machen würden: Der [Prozessor](#) wird gebeten, die Summe auszurechnen.

Letzteres kann bei komplexen Berechnungen signifikant schneller als Reduktion sein, gehört allerdings nicht zum *trusted kernel* des Systems und kann daher gewissermaßen nur „unter Vorbehalt“ in einen mathematischen Beweis aufgenommen werden.

Fazit

LEAN ist inzwischen ein beeindruckendes Tool und bietet dank einer aktiven Community mit [MATHLIB4](#) auch eine umfangreiche Bibliothek, die bereits viele mathematische Definitionen und Theoreme aus sehr unterschiedlichen Bereichen enthält. Man muss allerdings trotzdem sagen, dass die Art, wie Mathematiker Beweise aufschreiben, sich nach wie vor signifikant von der formalen Sprache unterscheidet, die LEAN oder vergleichbare Programme erwarten. Die Schnittmenge zwischen der Menge der Menschen, die mit LEAN arbeiten, und der Menge der aktiv forschenden Mathematiker ist ziemlich klein.

Um dieses Problem zu lösen, sind wohl [KI](#)-Tools nötig, die konventionell geschriebene Beweise zumindest halbautomatisch formalisieren. Daran wird auch gearbeitet, aber die Sache ist kompliziert. Wenn Sie das interessiert, dann schauen Sie sich vielleicht einmal das Video [The Shape of Math to Come](#) an.

Lösungen zu ausgewählten Aufgaben

Lösung 2.1. Die Liste hat vier Elemente. Und zwar die Zahlen 1, 2 und 5 sowie die Liste (23 42). Weder 23 noch 42 sind Elemente der Liste!

Kam Ihnen diese Frage kindisch vor und war Ihre Antwort richtig? Das freut mich. Leider zeigen die Erfahrungen mit der Mengenlehre im ersten Semester, dass viele Studierende Schwierigkeiten mit so etwas haben. Deshalb frage ich das an dieser Stelle vorsichtshalber noch einmal.

Lösung 2.2. Fünf! Wenn es sich um eine *Menge* handeln würde, wäre die Antwort *drei* richtig gewesen. Aber Listen entsprechen mathematisch eher *Tupeln*. Deshalb wäre es eigentlich auch besser, *Komponenten* statt *Elemente* zu sagen. Aber im üblichen Sprachgebrauch – z. B. auch im *SCHEME-Standard* – nennt man die Bestandteile einer Liste Elemente.

Lösung 3.1. Subtraktion wird in der Regel (nicht nur in SCHEME) *linksassoziativ* interpretiert. $(- 10 5 2)$ berechnet daher $(10 - 5) - 2 = 3$. (Ohne diese Konvention hätte man auch $10 - (5 - 2) = 7$ berechnen können.)

Lösung 3.3. Bei $(+)$ muss sinnvollerweise 0 herauskommen: das *neutrale Element* der Addition. Das wurde bereits in der Mathematik-Vorlesung besprochen, Stichwort *leere Summe*. Analog muss bei $(*)$ das Ergebnis 1 sein. Hier geht es um das *leere Produkt*.

Lösung 3.4. Das sähe so aus:

```
(fOne (fTwo 42 y))  
(fThree (fOne 12) (fOne (fOne 12)) (fTwo 42 23))  
(fTwo (fThree 1 2 (fTwo 8 (fOne 11))) (fTwo x y))
```

Lösung 3.5. Durch die Evaluationsregel (E5) ist die Reihenfolge der Berechnung „von innen nach außen“ fest vorgegeben. $3+4\cdot 5$ wird zu $(+ 3 (* 4 5))$ und $(3+4)\cdot 5$ wird zu $(* (+ 3 4) 5)$.

Siehe dazu auch [den Wikipedia-Artikel über die sogenannte polnische Notation](#). Eine Variante davon wird traditionell von Taschenrechnern des Herstellers [Hewlett-Packard](#) verwendet.

Lösung 3.6. `(define HAW '(H A W))`

Lösung 3.7. `(define plus +)`

Sie können nun so etwas wie `(plus 20 22)` eingeben. Man beachte den subtilen, aber wichtigen Unterschied zu `(define plus '+)`. Letzteres hätte bedeutet, dass `plus` ein Name für das *Symbol* `+` wird. Probieren Sie es aus:

```
(define auchplus '+)
auchplus
```

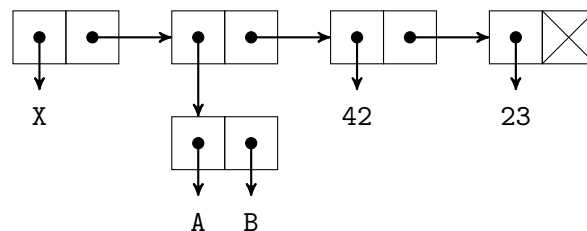
Lösung 4.3. Mit `caddr`. Ist Ihnen das zu aufwendig, so können Sie natürlich auch

```
(define third caddr)
```

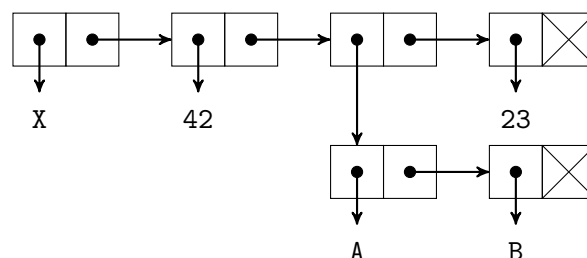
`first second etc.`

eingeben. (Siehe Aufgabe 3.7.) Das ist aber gar nicht nötig, weil in RACKET³ die Namen `first`, `second` und so weiter bis `fifteenth` bereits für die entsprechenden Funktionen definiert sind. Noch allgemeiner ist `list-ref`: Mit `(list-ref L n)` erhält man das n -te Element der Liste L .

Lösung 4.5. Die erste Liste sieht wie folgt aus. Achten Sie darauf, dass in der REPL zwischen A und B ein Punkt steht.

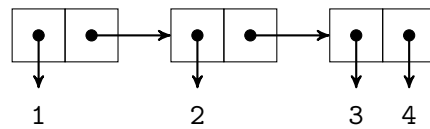


Die zweite Liste sieht so aus:



Lösung 4.6. Mit `(cons 1 (cons 2 (cons 3 4)))`. Wäre das letzte Argument `'()` statt `4` gewesen, so hätte sich die Liste `(1 2 3)` ergeben.

³Aber nicht in allen SCHEME-Implementationen.



SCHEME verwendet hier einen Mix aus der Listen- und der Paar-Darstellung. Man kann diese Darstellung so interpretieren, dass das Objekt so anfängt wie die Liste (1 2 3) und sich erst ganz am Ende von einer Liste unterscheidet – nämlich dort, wo der Punkt ist.

Lösung 4.7. Es gibt eine Fehlermeldung. Die Schreibweise ergibt keinen Sinn, weil rechts vom Punkt nur ein Objekt stehen kann.

Lösung 5.1. Mit sinnvoller Einrückung sollte es folgendermaßen aussehen:

```
(define square
  (lambda (n)
    (* n n)))
```

Hier wurde absichtlich `n` statt `x` verwendet, um zu demonstrieren, dass der Name willkürlich gewählt werden kann. (`square 3`) sollte nun 9 liefern.

Lösung 5.2. Zum Beispiel so:

```
(define gauss
  (lambda (k)
    (quotient (* k (add1 k)) 2)))
```

Hier wird `add1` von weiter oben verwendet, aber statt `(add1 k)` hätte man natürlich auch `(+ k 1)` schreiben können. Beachten Sie, dass `quotient` hier unproblematisch ist, weil das mit `*` berechnete Produkt garantiert **gerade** ist. (Warum?)

Lösung 5.3. Die äußere Funktion `(lambda (x) (+ x 1))` wird auf das Argument `((lambda (x) (+ x 2)) 3)` angewendet. Es muss also zunächst dieses Argument ausgewertet werden. Dafür erhält der Name `x` temporär den Wert 3 und es wird `(+ x 2)` berechnet. Das ergibt 5.

Den äußeren Funktionsaufruf kann man also durch `((lambda (x) (+ x 1)) 5)` ersetzen. Um das auch noch zu erledigen, erhält `x` nun den Wert 5 und `(+ x 1)` wird ausgewertet. Das ergibt 6 und diese Zahl wird ausgegeben.

Insbesondere sollte diese Aufgabe deutlich machen, dass die verschiedenen Stellen, an denen `x` vorkommt, nicht alle über „dasselbe“ `x` sprechen. Zwischendurch „ist“ `x` erst die Zahl 3 und dann die Zahl 5. Das „innere“ `x` spielt eine andere Rolle als das „äußere“ `x`. Nach der Auswertung ist `x` wieder das, was es vorher war. (Und eventuell undefiniert.)

Natürlich sollten Sie so etwas nicht schreiben, wenn Sie kein Skript verfassen. Besser lesbar wäre das hier gewesen:

```
((lambda (x) (+ x 1))
  ((lambda (y) (+ y 2)) 3))
```

Lösung 5.4. Wenn Sie z. B. `(strange 3)` aufrufen, dann wird während der Ausführung von `(list list)` der Variablenname `list` an den Wert 3 gebunden. Es soll also „(3 3)“ ausgeführt werden, was natürlich keinen Sinn ergibt.

Randnotiz: In COMMON LISP würde `strange` übrigens im Sinne der ursprünglichen Aufgabenstellung funktionieren. Der Grund dafür ist ein tiefgreifender Unterschied zwischen COMMON LISP und SCHEME, der im Artikel *Technical Issues of Separation in Function Cells and Value Cells* ausführlich diskutiert wird.

Lösung 5.5. Die folgenden drei Aufrufe würden beispielsweise funktionieren. Falls Sie selbst nicht darauf gekommen sind, dann überlegen Sie vielleicht mal vorher, was jeweils herauskommen sollte.⁴

```
(strange (lambda (x) x))
(strange (lambda (x) (eqv? x x)))
(strange list)
```

Auch `(strange strange)` funktioniert im Prinzip, führt aber zu einer **Endlosschleife**. Sie können so etwas in DRACKET beenden, indem Sie den roten Button STOP oben rechts anklicken.

Lösung 5.6. Am einfachsten geht das wohl so:

```
(define wrapper
  (lambda (k)
    (list (- k 1) k (+ k 1))))

(wrapper 8)
```

Lösung 5.7. Eine mögliche Lösung sieht so aus:

```
(define wrapper-factory
  (lambda (step)
    (lambda (k)
      (list (- k step) k (+ k step)))))

(wrapper-factory 1) 8
(wrapper-factory 3) 8
```

Lösung 5.8. Das sollte ungefähr so aussehen:

```
(define sum-from-to-cons
  (lambda (x-and-y)
    (- (gauss (cdr x-and-y))
```

⁴Zu `eqv?` siehe den Anfang von Abschnitt 6.

```
(gauss (- (car x-and-y) 1))))))  
  
(sum-from-to-cons (cons 3 6))
```

Lösung 6.1. Wir gehen davon aus, dass das Argument eine Zahl ist:

```
(define zero?  
  (lambda (num)  
    (= num 0)))
```

Lösung 6.2. Das sollte so aussehen:

```
(define even?  
  (lambda (n)  
    (= (modulo n 2) 0)))
```

Lösung 6.3. $(= a b c)$ ist genau dann wahr, wenn $a = b$ und $b = c$ (und damit wegen der **Transitivität** der Gleichheit auch $a = c$) gilt. $(< a b c)$ ist genau dann wahr, wenn $a < b$ und $b < c$ gilt. Für mehr als drei Argumente ist das Verhalten analog definiert.

Lösung 6.4. Natürlich hätte man das auch mit `modulo` machen können, aber es war so gemeint:

```
(define odd?  
  (lambda (n)  
    (not (even? n))))
```

Lösung 6.5. Für uns sind nur Symbole und ganze Zahlen Atome. Also:

```
(define atom?  
  (lambda (obj)  
    (or (integer? obj)  
        (symbol? obj))))
```

Lösung 6.6. $(\text{or } \dots)$ ist dann wahr, wenn mindestens eines der Argumente wahr ist. Wenn kein Argument vorhanden ist, muss die Antwort folglich `#f` sein. $(\text{and } \dots)$ ist dann falsch, wenn mindestens eines der Argumente falsch ist. Wenn kein Argument vorhanden ist, muss die Antwort folglich `#t` sein.

Lösung 6.7. Meine Version sieht so aus:

```
(define abs  
  (lambda (x)  
    (if (< x 0) (- x) x)))
```

Wie Sie sehen, darf natürlich auch alles in einer Zeile stehen. Funktioniert das auch noch korrekt, wenn man `<` durch `<=` ersetzt?

Falls Ihre Antwort anders aussieht, überprüfen Sie bitte, ob Sie bei (abs 0) die richtige Antwort bekommen!

Lösung 6.8. Das könnte man z. B. so machen:

```
(define max
  (lambda (a b)
    (if (> a b) a b)))
```

Lösung 6.9. Auch hier sollten Sie insbesondere prüfen, ob Ihr Code die richtige Antwort beim Aufruf (sub1 0) liefert.

```
(define sub1
  (lambda (n)
    (if (> n 0) (- n 1) n)))
```

Lösung 6.10. Mithilfe von Aufgabe 6.2 (oder unter Verwendung der „eingebauten“ Funktion even?) sieht das so aus:

```
(define collatz
  (lambda (n)
    (if (even? n)
        (quotient n 2)
        (+ 1 (* 3 n)))))
```

Man hätte natürlich auch mit Aufgabe 6.4 arbeiten können:

```
(define collatz
  (lambda (n)
    (if (odd? n)
        (+ 1 (* 3 n))
        (quotient n 2))))
```

Lösung 6.11. Eine mehr oder weniger direkte Umsetzung der Aufgabenstellung könnte so aussehen:

```
(define which
  (lambda (m n)
    (if (or (= m 0) (= n 0))
        'zero
        (if (or (and (> m 0) (> n 0))
                (and (< m 0) (< n 0)))
            'same
            'different))))
```

Sie hatten aber hoffentlich die Idee, es etwas eleganter zu lösen:

```
(define which
  (lambda (m n)
```

```
(if (= (* m n) 0)
    'zero
    (if (> (* m n) 0)
        'same
        'different))))
```

Lösung 6.12. Es handelt sich um einen Spezialoperator, weil nur eines der beiden hinteren Argumente ausgewertet wird. Da Auswertungen Zeit und Speicherplatz beanspruchen, wäre es nicht sinnvoll, das andere Argument, das gar nicht benötigt wird, auch noch auszuwerten. Außerdem kann es auch noch andere Gründe haben, auf eine Auswertung zu verzichten:

```
(if (= x 0)
    'not-possible
    (/ 1 x))
```

Würden hier immer alle Argumente ausgewertet werden, so gäbe es eine Fehlermeldung, wenn x den Wert null hat, obwohl der Code das gerade vermeiden will. Und es könnte natürlich noch schlimmer kommen:

```
(if (> 7 3)
    'system-works-as-expected
    (destroy-the-world))
```

Lösung 6.13. Die Implementation ist „faul“ in dem Sinne, dass sofort aufgehört wird, Argumente zu evaluieren, wenn die Antwort klar ist. and evaluiert also die Argumente der Reihe nach, bis eines falsch ist. Weil dann klar ist, dass die Antwort `#f` sein muss, müssen die restlichen Argumente nicht mehr betrachtet werden. Analog werden die Argumente von `or` nur evaluiert, bis zum ersten Mal ein wahres auftaucht. Wären `and` und `or` Funktionen, dann würde SCHEME alle Argumente evaluieren, bevor die Funktionen aufgerufen werden.

Wie gesagt machen das auch JAVA, PYTHON, C und viele andere Sprachen so.⁵ Der Hauptgrund dafür ist wie bei `if` (siehe Aufgabe 6.12) Optimierung: Warum sollte man Zeit und Speicherplatz für das Auswerten von Argumenten verschwenden, wenn diese gar nicht benötigt werden?

Lösung 6.15. Dass jeder Ausdruck einen Wert zurückgibt, stimmt zumindest in COMMON LISP. Aber auch da müsste man etwas genauer formulieren. Erstens können Operatoren, die einen bestimmten Ausführungskontext verlassen, natürlich keine Werte zurückgeben. Beispiele dafür wären `error`, `throw` oder `return-from`. Zweitens gibt es einen **Mechanismus**, mit dem Funktionen mehr als einen Wert zurückgeben können oder gar keinen. Aber auch Letzteres kann man semantisch als „Rückgabe“ interpretieren und `testen`. Prinzipiell gibt jedenfalls in COMMON LISP jede Form, die evaluiert wird, etwas zurück.

⁵Es gibt allerdings Ausnahmen wie **ADA** oder **VISUAL BASIC**.

In SCHEME gelten dieselben Einschränkungen wie im letzten Absatz. Dort kommt jedoch noch hinzu, dass anders als in COMMON LISP nicht jede syntaktisch korrekte Form ein Ausdruck (*expression*) ist. Es gibt auch Kategorien wie *Definition* (z. B. mittels `define`) oder *Deklaration*. Solche Formen geben nichts zurück, wenn sie evaluiert werden.

Lösung 6.16. Meine Lösung sieht so aus:

```
(define distinguish
  (lambda (x)
    (cond ((integer? x)
           (if (even? x) 'even 'odd))
          ((symbol? x) 'symbol)
          ((null? x) 'empty-list)
          (else 'no-idea))))
```

Beachten Sie die sinnvolle Einrückung, anhand derer man erkennt, dass der `if`-Ausdruck zum `integer?`-Test gehört. Wie bereits erwähnt sollte das der Editor automatisch richtig machen.

Man hätte es natürlich auch so schreiben können:

```
(define distinguish
  (lambda (x)
    (cond ((and (integer? x) (even? x)) 'even)
          ((and (integer? x) (odd? x)) 'odd)
          ((symbol? x) 'symbol)
          ((null? x) 'empty-list)
          (else 'no-idea))))
```

Aber abgesehen vom subjektiven Urteil, dass die erste Variante eleganter aussieht, hat die zweite den Nachteil, dass die Funktion `integer?` ggf. zweimal mit demselben Argument aufgerufen wird. Das wird in der ersten Version vermieden. (Aber wir wollten ja nicht über Effizienz sprechen ...)

Lösung 7.1. Weil das erste Argument von `set!` nicht ausgewertet wird. Siehe die Erläuterungen zu Spezialoperatoren in Abschnitt 3.

Lösung 7.2. `let` erzeugt Variablenbindungen. Aber wie wir in Abschnitt 5 gesehen haben, macht `lambda` das auch, und zwar für die Argumente. Daher hat

```
(let ((var 42))
  [...code...])
```

denselben Effekt wie das hier:

```
((lambda (var)
  [...code...])
```

```
)
42)
```

Sie können das am konkreten Beispiel `sum` ausprobieren.

```
(define sum
  (lambda (n)
    ((lambda (res)
      (do ((i 0 (+ i 1)))
        ((> i n))
        (set! res (+ res i))))
     res)
    0)))
```

Tatsächlich könnte man `let`, wenn es das nicht schon gäbe, in SCHEME selbst programmieren. Die einfachste Variante würde so aussehen (wobei die [Auslassungspunkte](#) dazugehören):

```
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

Das ist die am Ende von Abschnitt 4 erwähnte [Metaprogrammierung](#). In LISP-Sprachen kann man die Syntax der Sprache also nach Bedarf ändern!

Der [SCHEME-Standard](#) zeigt übrigens für eine ganze Reihe von Spezialoperatoren wie z. B. `let` (die man in SCHEME einfach *syntax* nennt), wie man sie selbst implementieren könnte, wenn sie nicht vorhanden wären.

Man beachte, dass das nicht mit dem [Präprozessor](#) von C bzw. C++ zu vergleichen ist. Dort kann zwar ebenfalls die Syntax bis zu einem gewissen Grad angepasst werden, aber das geschieht auf der Ebene des [Quelltextes](#) in einem separaten Arbeitsgang vor dem [Kompilieren](#) und in einer anderen Sprache. In LISP-Sprachen wird hingegen die Syntaxänderung in der Sprache selbst durchgeführt und arbeitet wie bereits erwähnt quasi direkt mit dem [Syntaxbaum](#).

Lösung 7.3. Man kann das so machen:

```
(define sum
  (lambda (n)
    (do ((i 0 (+ i 1))
        (res 0))
      ((> i n) res)
      (set! res (+ res i)))))
```

Lösung 7.4. Dieser Abschnitt hat hoffentlich klargemacht, dass eine Funktion dadurch charakterisiert ist, was sie zurückgibt. Das ist ihr entscheidendes Merkmal. `foo1` gibt $x+42$ zurück, wenn ihre Eingabe x ist. `foo2` gibt nichts zurück! Geben Sie

(foo1 1) und (foo2 1) im Interaktionsfenster ein, so sieht es auf den ersten Blick so aus, als machten beide Funktionen dasselbe. Wenn Sie genau hinschauen, sehen Sie jedoch, dass DRACKET unterschiedliche Farben verwendet. Im ersten Fall wird uns angezeigt, was die Funktion zurückgegeben hat, im zweiten Fall schreibt die Funktion selbst als Seiteneffekt die Zeichenkette 43 ins Interaktionsfenster. Aber es wird kein Rückgabewert angezeigt! (Rein technisch ist der Rückgabewert in RACKET der spezielle Wert `#<void>`.)

Der Unterschied wird vielleicht noch deutlicher, wenn Sie

```
(define x1 (foo1 1))
(define x2 (foo2 1))
```

eingeben und sich dann x1 und x2 anzeigen lassen. Oder Sie probieren das aus:

```
(+ 1 (foo1 1))
(+ 1 (foo2 1))
```

Diese Antwort ist deshalb so ausführlich, weil nach meiner Erfahrung manche Studierende, die wenig Erfahrung mit interaktiven Programmiersprachen haben, den grundlegenden Unterschied zwischen einem zurückgegebenen und einem nur angezeigten Objekt nicht verstehen. Ich hoffe, Sie gehören spätestens jetzt nicht mehr zu dieser Gruppe.

Lösung 8.1. Nein! Schauen Sie sich Aufgabe 2.1 noch einmal an.

Lösung 8.2. Ich hatte an so etwas gedacht:

```
(define num-ints
  (lambda (list)
    (cond ((null? list) 0)
          (else (+ (if (integer? (car list))
                        1 0)
                    (num-ints (cdr list)))))))
```

Lösung 8.3. Meine Lösung sieht so aus:

```
(define list-of-atoms?
  (lambda (list)
    (cond ((null? list) #t)
          ((atom? (car list))
           (list-of-atoms? (cdr list)))
          (else #f))))
```

Lösung 8.4. Ungefähr so:

```
(define sum
  (lambda (list)
    (cond ((null? list) 0)
```

```
(else (+ (car list)
         (sum (cdr list))))))
```

Und man kann so etwas natürlich auch mit `if` statt `cond` machen:

```
(define sum
  (lambda (list)
    (if (null? list)
        0
        (+ (car list) (sum (cdr list))))))
```

Lösung 8.5. Man kann die Funktion `sum` mehr oder weniger abschreiben und muss nur beachten, dass das leere Produkt 1 und nicht 0 ist.

```
(define prod
  (lambda (list)
    (cond ((null? list) 1)
          (else (* (car list)
                   (prod (cdr list))))))
```

Lösung 8.6. Der Unterschied kann folgendermaßen verdeutlicht werden:

```
> (define orig '(42 23 99))
> (eqv? orig (copy orig))
#f
> (eqv? orig (naive-copy orig))
#t
```

`orig` und `(naive-copy orig)` sind *identisch* in dem Sinne, dass es sich technisch um dasselbe Objekt im Speicher handelt. `(copy orig)` ist hingegen eine verkettete Liste mit denselben Elementen, liegt aber an einer anderen Stelle im Speicher.

Diese Unterscheidung wird dann relevant, wenn man *nicht funktional* programmiert. Im Folgenden ein Beispiel mit SCHEME-Prozeduren, die wir „offiziell“ nicht verwenden, deren Funktionsweise aber wohl durch ihren Namen klar sein sollte.⁶ Das wird allerdings in RACKET nicht funktionieren, weil diese Sprache in einigen Details vom *standardisierten* SCHEME abweicht. Sie können es ausprobieren, indem Sie die obigen Definitionen und den folgenden Code online in <https://try.scheme.org/> eingeben.

```
> (set-car! (copy orig) 41)
> orig
(42 23 99)
> (set-car! (naive-copy orig) 41)
> orig
(41 23 99)
```

⁶Beachten sie das Ausrufezeichen. Siehe Fußnote 44 auf Seite 25.

Bei der Verwendung von `copy` hat `set-car!` nur die Kopie modifiziert. Benutzt man stattdessen die „naive“ Version der Funktion, dann verändert man das Original, weil es gar keine „Kopie“ gibt.

Lösung 8.7. Man muss dafür in `cond` eine Bedingung hinzufügen:

```
(define invert-no-zeros
  (lambda (list)
    (cond ((null? list) '())
          ((= 0 (car list))
           (invert-no-zeros (cdr list)))
          (else (cons (- (car list))
                       (invert-no-zeros (cdr list)))))))
```

Lösung 8.8. So sieht meine Lösung aus:

```
(define remove-all
  (lambda (atom list)
    (cond ((null? list) '())
          ((eqv? (car list) atom)
           (remove-all atom (cdr list)))
          (else
           (cons (car list)
                 (remove-all atom (cdr list)))))))
```

Lösung 8.9. Abgesehen vom neuen Namen ist nur eine Änderung nötig:

```
(define remove-first
  (lambda (atom list)
    (cond ((null? list) '())
          ((eqv? (car list) atom)
           (copy (cdr list)))      ;; geänderte Zeile
          (else
           (cons (car list)
                 (remove-first atom (cdr list)))))))
```

Solange man rein **funktional** programmiert kann man statt `(copy (cdr list))` auch einfach `(cdr list)` schreiben. Man erhält dann allerdings ggf. etwas, was man *shared structure* nennt.

Lösung 8.10. Das sollte nach den vorherigen Aufgaben einfach sein.

```
(define member?
  (lambda (atom list)
    (cond ((null? list) #f)
          ((eqv? atom (car list)) #t)
          (else (member? atom (cdr list))))))
```

Lösung 8.11. Wie wäre es so?

```
(define unique
  (lambda (list)
    (cond ((null? list) list)
          ((member? (car list) (cdr list))
           (unique (cdr list)))
          (else (cons (car list) (unique (cdr list)))))))
```

Lösung 8.12. Bei Rekursion über das zweite Argument sieht es so aus:

```
(define union
  (lambda (x y)
    (cond ((null? y) x)
          ((member? (car y) x) (union x (cdr y)))
          (else (cons (car y) (union x (cdr y)))))))
```

Lösung 8.13. Meine Lösung sieht so aus:

```
(define map
  (lambda (fn list)
    (cond ((null? list) '())
          (else (cons (fn (car list))
                      (map fn (cdr list)))))))
```

Lösung 8.14. Man beachte, dass hier gemäß der Bemerkung am Ende der Aufgabenstellung der car von obj nicht überprüft wird, weil das für eine Liste irrelevant ist. *Jedes* Objekt (jeder S-Ausdruck) kann Element einer Liste sein.

```
(define list?
  (lambda (obj)
    (cond ((null? obj) #t)
          ((not (pair? obj)) #f)
          (else (list? (cdr obj))))))
```

Lösung 9.2. Unsere Definition von add1 verwendet +. Das würde in einem Teufelskreis enden: + ruft add1 auf, add1 ruft + auf und immer so weiter.

Lösung 9.3. Die Struktur ist dieselbe wie bei +:

```
(define *
  (lambda (a b)
    (cond ((zero? b) 0)
          (else (+ a (* a (sub1 b)))))))
```

Lösung 9.4. Mir schwebte eine Lösung vor, die die Struktur unserer Definition von < eins zu eins übernimmt.

```
(define =
  (lambda (a b)
```

```
(cond ((zero? b) (zero? a))
      ((zero? a) #f)
      (else (= (sub1 a) (sub1 b)))))
```

Lösung 9.5. Dabei würden wir ausnutzen, dass für zwei Zahlen a und b immer nur genau einer der drei Fälle $a < b$, $b < a$ und $a = b$ eintritt.

```
(define =
  (lambda (a b)
    (not (or (< a b) (< b a)))))
```

Natürlich arbeitet „unser“ = nicht wirklich „ohne Rekursion“, weil ja gegebenenfalls sogar zweimal $<$ aufgerufen wird.

Lösung 9.6. Das sieht dann ungefähr so aus:

```
(define > (lambda (a b) (< b a)))
(define <= (lambda (a b) (or (< a b) (= b a))))
(define >= (lambda (a b) (or (> a b) (= b a))))
```

Lösung 9.7. Man kann nun auf das Überprüfen von $(< a b)$ verzichten.

```
(define my-sub1
  (lambda (n)
    (if (> n 0) (sub1 n) n)))

(define -
  (lambda (a b)
    (cond ((zero? b) a)
          (else (my-sub1 (- a (my-sub1 b)))))))
```

Lösung 9.8. Am einfachsten ist es wohl so:

```
(define quotient
  (lambda (a b)
    (cond ((< a b) 0)
          (else (add1 (quotient (- a b) b)))))
```

Lösung 9.9. Die Funktion gerät in eine **Endlosschleife**. Das ist aber auch OK, weil **Division durch null nicht möglich** ist. Man könnte natürlich eine Warnung einbauen:

```
(define quotient
  (lambda (a b)
    (cond ((zero? b) 'are-you-kidding?)
          ((< a b) 0)
          (else (add1 (quotient (- a b) b))))))
```

Lösung 9.10. Man kann es rekursiv wie bei `quotient` erledigen.

```
(define modulo
  (lambda (a b)
    (cond ((zero? b) 'nope)
          ((< a b) a)
          (else (modulo (- a b) b)))))
```

Man kann aber auch ausnutzen, dass wir `quotient` bereits haben.

```
(define modulo
  (lambda (a b)
    (- a (* b (quotient a b)))))
```

Lösung 9.11. Ich hatte an so etwas gedacht:

```
(define even?
  (lambda (n)
    (cond ((zero? n) #t)
          (else (not (even? (sub1 n)))))))
```

Man könnte es auch noch umständlicher folgendermaßen machen:

```
(define even?
  (lambda (n)
    (cond ((zero? n) #t)
          (else (odd? (sub1 n)))))

(define odd?
  (lambda (n)
    (cond ((zero? n) #f)
          (else (even? (sub1 n)))))
```

Beide Versionen sind natürlich wegen ihrer Ineffizienz für die Praxis ungeeignet, aber für das Verständnis sind sie hilfreich. Insbesondere die zweite zeigt etwas, was wir bisher noch nicht gemacht haben, nämlich eine *Doppelrekursion*, bei der die Rekursion auf den ersten Blick gar nicht erkennbar ist, weil keine der beiden Funktionen sich selbst aufruft.

Lösung 9.12. Meine Lösung sieht folgendermaßen aus.

```
(define digit-sum
  (lambda (n)
    (cond ((zero? n) 0)
          (else (+ (modulo n 10)
                    (digit-sum (quotient n 10)))))))
```

Lösung 9.13. Im Wesentlichen sollte Ihre Lösung so aussehen:

```
(define expt
  (lambda (a b)
    (cond ((zero? b) 1)
          (else (* a (expt a (sub1 b)))))))
```

Lösung 9.14. Das sollte nach allen vorherigen Aufgaben hoffentlich einfach sein.

```
(define fact
  (lambda (n)
    (cond ((zero? n) 1)
          (else (* n (fact (sub1 n)))))))
```

Geben Sie z. B. mal `(fact 100)` ein. Sie werden sehen, dass SCHEME anders als beispielsweise JAVA keine Probleme mit *arithmetischem Überlauf* hat.

Lösung 9.15. In meiner Lösung erfolgt die Rekursion über die Liste, es wird jedoch gleichzeitig ein „Zeiger“ `pos` für die Position mitgeführt.

```
(define position-helper
  (lambda (atom list pos)
    (cond ((null? list) #f)
          ((eqv? atom (car list)) pos)
          (else (position-helper atom (cdr list)
                                  (add1 pos))))))

(define position
  (lambda (atom list)
    (position-helper atom list 0)))
```

Die Funktion fängt mit dem Zählen bei null an, weil das in der Informatik so üblich ist. Es sollte klar sein, was man ändern muss, wenn man lieber bei eins anfangen will.

Lösung 9.16. So sieht meine Lösung aus:

```
(define depth-helper
  (lambda (list m)
    (cond ((null? list) m)
          ((list? (car list))
           (depth-helper (cdr list)
                         (max m (add1 (depth (car list))))))
          (else (depth-helper (cdr list) m))))

(define depth
  (lambda (list)
    (depth-helper list 0)))
```

Das Argument *m* ist die maximale bisher „gesehene“ Tiefe. Dass `depth-helper` an einer Stelle `depth` statt sich selbst aufruft, ist eher eine stilistische Frage. Beides wäre OK.

Lösung 9.17. Das sollte ungefähr so aussehen:

```
(define tree-equal?
  (lambda (thing1 thing2)
    (cond ((eqv? thing1 thing2) #t)
          ((not (and (list? thing1) (list? thing2))) #f)
          ((or (null? thing1) (null? thing2)) #f)
          (else (and (tree-equal? (car thing1) (car thing2))
                     (tree-equal? (cdr thing1) (cdr thing2)))))))
```

Der dritte Fall kann nur eintreten, wenn nur eine von beiden Listen die leere ist, weil sonst bereits der erste Fall eingetreten wäre.

Lösung 9.18. Meine Lösung sieht folgendermaßen aus:

```
(define subst
  (lambda (new old tree)
    (cond ((eqv? tree old) new)
          ((not (list? tree)) tree)
          (else (map (lambda (thing)
                       (subst new old thing))
                     tree)))))
```

Lösung 9.19. Man kann beide Abbruchbedingungen mit einer Zeile erledigen.

```
(define fibo
  (lambda (n)
    (cond ((<= n 1) n)
          (else (+ (fibo (sub1 n))
                   (fibo (sub1 (sub1 n)))))))
```

Lösung 9.20. Das lässt sich recht einfach erledigen.

```
(define next-prime
  (lambda (n)
    (cond ((prime? n) n)
          (else (next-prime (add1 n)))))
```

Die Abbruchbedingung, von der weiter oben die Rede war, ist in diesem Fall der [Satz von Euklid](#). Der garantiert, dass wir irgendwann auf eine Primzahl treffen werden, wenn wir das Argument immer wieder um eins vergrößern.

Lösung 9.21. Auch das kann man sehr einfach lösen.

```
(define nth-prime
  (lambda (n)
    (cond ((= n 1) 2)
          (else (next-prime (add1 (nth-prime (sub1 n)))))))
```

Lösung 9.22. Meine Lösung sieht so aus:

```

(define prime-div
  (lambda (n k)
    (cond ((and (prime? k) (zero? (modulo n k))) k)
          (else (prime-div n (add1 k))))))

(define p-prod
  (lambda (n)
    (cond ((zero? n) 1)
          (else (* (p (sub1 n))
                    (p-prod (sub1 n))))))

(define p
  (lambda (n)
    (cond ((zero? n) 2)
          (else (prime-div (add1 (p-prod n)) 2)))))

```

Die Funktion, die aufgerufen wird, ist `p`. Die Hilfsfunktion `p-prod` berechnet das Produkt der Folgenwerte von p_0 bis inklusive p_{n-1} . `prime-div` berechnet den kleinsten Primteiler von n , der nicht kleiner als k ist.

Es ist natürlich extrem ineffizient, dass die Werte p_i für kleinere i ständig neu berechnet werden müssen. Falls Sie wissen wollen, wie man so etwas vermeiden kann, informieren Sie sich über die Technik der [Memoisation](#).

Lösung 9.23. Das Programmieren ist kein großes Problem, weil man die Definition mehr oder weniger wortwörtlich abschreiben kann.

```

(define a
  (lambda (n m)
    (cond ((zero? n) (add1 m))
          ((zero? m) (a (sub1 n) 1))
          (else (a (sub1 n) (a n (sub1 m)))))))

```

Interessanter ist die Frage, warum das klappt, warum also die Abbruchbedingung immer erreicht wird. Das ist auf den ersten Blick nicht offensichtlich. Eine mögliche Erklärung sieht folgendermaßen aus: Man stelle sich alle Paare aus $\mathbb{N} \times \mathbb{N}$ „lexikographisch sortiert“ vor. Das ist im Prinzip dieselbe Sortierung wie in Wörterbüchern, wobei hier die „Wörter“ immer aus zwei Zahlen bestehen. (n_1, m_1) kommt also in der Sortierung vor (n_2, m_2) , wenn entweder n_1 kleiner als n_2 ist oder wenn n_1 und n_2 gleich sind und m_1 kleiner als m_2 ist. In den Fällen, in denen die Funktion `a` sich selbst aufruft (auch in dem in der letzten Zeile mit zwei Aufrufen) ist das Paar von Argumenten im Aufruf immer eines, das vor (n, m) liegt.

Man muss sich jetzt nur noch überlegen, dass die meisten Paare aus $\mathbb{N} \times \mathbb{N}$ zwar (anders als natürliche Zahlen) unendlich viele „Vorgänger“ haben, dass aber eine Sequenz aus Paaren, bei denen jedes Paar lexikographisch vor dem vorherigen steht, immer endlich sein muss. Die Details will ich nicht aufschreiben, aber ich glaube, es wird Ihnen klar werden, wenn Sie ausgehend von einem Paar wie z. B.

(4,6) überlegen, welche Paare *vor* diesem Paar stehen und welche Wahlmöglichkeiten Ihnen verbleiben, nachdem Sie eines von denen ausgewählt haben.

Lösung 10.1. In der ersten Zeile steht kein einziger λ -Ausdruck. Bei der ersten Zeichenkette fehlen die Klammern. Bei der zweiten stehen zwischen λ und dem Punkt zwei Variablen statt einer. Bei der dritten steht dort gar keine Variable. Bei der vierten fehlen Klammern. Sowohl $(\lambda x_1.(x_1 x_1))$ als auch $((\lambda x_1.x_1) x_1)$ wären jedoch λ -Ausdrücke.

In der zweiten Zeile ist nur die dritte Zeichenkette ein λ -Ausdruck. Er ist durch zweimalige Anwendung von (L2) auf Variablen entstanden. Bei der ersten Zeichenkette fehlen die Klammern, bei der zweiten stehen zwischen den Klammern drei Variablen, aber für eine Applikation wären nur zwei λ -Ausdrücke erlaubt. Bei der vierten Zeichenkette ist bereits (x_2) kein λ -Ausdruck.

In der dritten Zeile stehen vier λ -Ausdrücke. Der erste entstand durch (L2) gefolgt von (L3). Der zweite entstand durch zweimalige Anwendung von (L3), der dritte durch Anwendung auf (L2) auf den vorherigen, und der vierte durch (L3) gefolgt von (L2).

Lösung 10.2. Hier mein Lösungsvorschlag mit Kommentaren:

```
(define var?
  (lambda (thing)
    ;; alle Symbole außer 'lambda sind Variablen
    (and (symbol? thing) (not (eqv? thing 'lambda)))))

(define var-list?
  ;; eine Variable mit Klammern drumherum?
  (lambda (thing)
    (and (list? thing)
         (= 1 (length thing))
         (var? (car thing)))))

(define lambda-expr?
  (lambda (thing)
    (cond ((var? thing) #t) ;; Variablen sind Lambda-Ausdrücke
          ;; keine Variable, dann muss es eine Liste sein
          ((not (list? thing)) #f)
          ((= 2 (length thing))
           ;; zwei Elemente, dann beide Lambda-Ausdrücke
           (and (lambda-expr? (first thing))
                 (lambda-expr? (second thing))))
          ((= 3 (length thing))
           ;; drei Elemente, dann 'lambda ...
           (and (eqv? (first thing) 'lambda)
                 ;; ... anschließend Variable in Klammern ...
                 (var-list? (second thing))
                 ;; ... und schließlich ein Lambda-Ausdruck
                 (lambda-expr? (third thing)))))
```

```
;; andere Längen sind nicht möglich
(else #f))))
```

Lösung 10.3. In der ersten Reihe ist die erste Zeichenkette eine Abkürzung für $(\lambda x_1.x_1)$ und die vierte eine für $(\lambda x_1.(x_1 x_1))$.

In der zweiten Reihe ist die erste Zeichenkette eine Abkürzung für $(x_1 x_2)$ und die zweite eine für $((x_1 x_2) x_3)$.

Lösung 10.4. Der besseren Übersichtlichkeit halber wird im Folgenden die Aufgabenstellung wiederholt. Unter jedem Ausdruck steht dessen ausführliche Form.

$$\begin{array}{cccc} \lambda x.y x z & \lambda x.\lambda y.\lambda z.w & (\lambda x.y)(\lambda x.y z) w & (\lambda x.x) x y \\ \lambda x.((y x) z) & \lambda x.(\lambda y.(\lambda z.w)) & ((\lambda x.y)(\lambda x.(y z))) w & ((\lambda x.x) x) y \end{array}$$

Lösung 10.5. Ein Lösungsvorschlag, der die Funktion `var-list?` aus der Lösung von Aufgabe 10.2 verwendet:

```
(define expand
  (lambda (thing)
    (cond ((or (not (list? thing)) (< (length thing) 3))
          ;; nichts zu tun
          thing)
          ((eqv? (first thing) 'lambda)
           ;; evtl. Abstraktion
           (cond ((not (and (> (length thing) 2)
                           (var-list? (second thing))))
                 ;; nein, doch nicht
                 thing)
                 (else (list 'lambda
                              (second thing)
                              (cond ((= (length (cddr thing)) 1)
                                    ;; nur ein Element hinter Var.
                                    (expand (first (cddr thing))))
                              (else
                               ;; mehrere Element
                               (expand (cddr thing)))))))
           ;; Applikation
           (else (expand (cons (list (first thing) (second thing))
                               (cddr thing)))))))
```

Lösung 11.2. Grundsätzlich sollten Sie sich in diesem Kapitel angewöhnen, Ihre Antwort selbst mit `DRLAMBDA` abzugleichen, bevor Sie die Lösung lesen. Bei dieser Aufgabe geht es darum, ob man die Klammerregeln verstanden hat und ob man die provisorische Definition richtig anwendet.

Der erste Ausdruck ist mit Klammern $((\lambda x.(x x)) w) w$ und aus dem Teilausdruck $(\lambda x.(x x)) w$ wird $(w w)$, so dass sich insgesamt $(w w) w$ ergibt, wobei man die Klammern auch weglassen kann: $w w w$.

Der zweite Ausdruck ist $(\lambda x.(x x))(w w)$. $(w w)$ spielt hier die Rolle des Ausdrucks B in der Definition und wird für x eingesetzt. Damit ergibt sich $((w w) (w w))$ bzw. kürzer $w w (w w)$.

Der dritte Ausdruck ist $\lambda x.(((x x) w) w)$. Das ist gar kein Redex und damit greift die Regel für die Reduktion nicht.

Lösung 11.4. Man erhält wieder $(\lambda x.(x x)) (\lambda x.(x x))$. Es ist also *nicht* so, dass man jeden λ -Ausdruck solange reduzieren kann, bis keine Redexe mehr da sind.

Dieser λ -Ausdruck ist das kleinste Beispiel dafür. Er wird in der Literatur häufig Ω genannt.

Lösung 11.5. Ich zeige Ihnen meine Lösung schrittweise mit Kommentaren. Zuerst habe ich ein paar Hilfsfunktionen sowie alternative Namen definiert, damit der Rest des Codes besser lesbar ist. `app` steht dabei für Applikation und `abstr` für Abstraktion. Die beiden Teile einer Applikation nennen wir hier *Funktion* (`fn`) und *Argument* (`arg`). Und wie in SCHEME nennen wir den dritten Teil einer Abstraktion ihren *Rumpf* (`body`).

```
(define app?
  (lambda (expr)
    (and (list? expr) (= 2 (length expr)))))
(define app-fn car)
(define app-arg cadr)
(define build-app list)

(define abstr?
  (lambda (expr)
    (and (list? expr) (= 3 (length expr)))))
(define abstr-var caadr)
(define abstr-body caddr)
(define build-abstr
  (lambda (var body) (list 'lambda (list var) body)))

(define redex?
  (lambda (expr)
    (and (app? expr) (abstr? (app-fn expr)))))
```

Der obige Code setzt voraus, dass das Argument `expr` den Test `lambda-expr?` bereits erfolgreich bestanden hat. Das passiert weiter unten.

Als Nächstes kommt eine Funktion, die einen β -Reduktionsschritt vornimmt und dabei darauf achtet, den „richtigen“ Redex (im Sinne der Standardreihenfolge) auszuwählen. Ist kein Redex vorhanden, gibt diese Funktion `#f` zurück, um zu signalisieren, dass sich nichts geändert hat.

```
(define reduce-once
  (lambda (expr)
    (cond ((or (var? expr)
```

```

        (not (lambda-expr? expr))) #f)
    ((redex? expr)
     (subst (app-arg expr) (abstr-var (app-fn expr))
            (abstr-body (app-fn expr))))
    ((app? expr)
     (let ((new-fn (reduce-once (app-fn expr))))
       (cond (new-fn (build-app new-fn (app-arg expr)))
             (else
              (let ((new-arg (reduce-once (app-arg expr))))
                (cond (new-arg
                      (build-app (app-fn expr) new-arg))
                      (else #f)))))))
    (else ;; must be an abstraction
     (let ((new-body (reduce-once (abstr-body expr))))
       (cond (new-body
              (build-abstr (abstr-var expr) new-body))
             (else #f))))))

```

Die eigentliche Reduktion in Standardreihenfolge erledigt nun diese Funktion, die so lange `reduce-once` anwendet, bis sich nichts mehr ändert (genauer: bis die Antwort `#f` von `reduce-once` kommt).

```

(define normal-reduction
  (lambda (expr)
    (let ((new-expr (reduce-once expr)))
      (cond (new-expr (normal-reduction new-expr))
            (else expr)))))

```

Bei einer Eingabe wie in Aufgabe 11.4 würde diese Funktion in eine [Endlosschleife](#) geraten. So etwas ließe sich nur dadurch vermeiden, dass man sich alle Zwischenergebnisse merkt und nach jedem Schritt prüft, ob ein Ausdruck schon einmal vorkam.

Sinnvollerweise sollte man seinen Code auch immer testen. Damit Sie ausprobieren können, ob Ihre Lösung korrekt arbeitet, können Sie das Folgende verwenden:

```

(define test-cases
  '(((lambda (x) y) x) y)
    ((lambda (x) y) y) y)
    ((x ((lambda (x) y) y)) (x y))
    ((lambda (x) x) y) y)
    ((lambda (x) x) x) x)
    ((lambda (x) (x x)) w) (w w))
    ((lambda (x) (x y x)) v) (v y v))
    (((lambda (x) (x x)) w) w) ((w w) w))
    ((lambda (x) (x x)) (w w)) ((w w) (w w)))
    ((lambda (x) (((x x) w) w)) (lambda (x) (((x x) w) w)))
    ((lambda (x) (x y)) ((lambda (z) ((z z) z)) w)) (((w w) w) y))
    (((lambda (x) (lambda (y) (y x))) w) v) (v w))

```

```

((((lambda (x) (lambda (y) z)) w) w) z)
(((lambda (x) (x x)) ((lambda (w) (w w)) v)) ((v v) (v v)))
((((lambda (x) (lambda (y) x)) v) w) v)))

(define test-reduction
  (lambda (reduce-fn)
    (do ((cases test-cases (cdr cases)))
      ((null? cases))
      (let ((case (first cases)))
        (unless (equal? (reduce-fn (first case))
                        (second case))
          (printf "~A: Expected ~A, but got ~A.~%"
                  (first case) (second case)
                  (reduce-fn (first case)))))))

(test-reduction normal-reduction)

```

Dabei setzen Sie für `normal-reduction` den Namen Ihrer Funktion ein. Wenn die Funktion einfach ohne Meldung durchläuft, ist alles in Ordnung.

`test-cases` ist eine Sammlung der Beispiele und Aufgaben aus diesem Abschnitt. `equal?` ist eine SCHEME-Funktion die für S-Ausdrücke dasselbe macht wie die Funktion `tree-equal` aus Aufgabe 9.17. `test-reduction` verwendet die SCHEME-Syntax `unless` sowie die Racket-Funktion `printf`, die wir bisher nicht besprochen haben und die außerhalb dieser Testfunktion nicht weiter verwendet werden.

Lösung 12.2. Es reicht, die Funktion `subst` durch diese zu ersetzen:

```

(define new-subst
  (lambda (new old tree)
    (cond ((eqv? tree old) new)
          ((or (not (list? tree))
               (and (abstr? tree)
                    (eqv? (abstr-var tree) old))) ;;<--
           tree)
          (else (map (lambda (thing)
                       (new-subst new old thing))
                     tree)))))

```

Neu ist hier nur der markierte Test. Natürlich muss `reduce-once` jetzt `new-subst` statt `subst` aufrufen.

Lösung 12.3. Wenn ich mich nicht vertippt habe, sieht es so aus:

A	$F[A]$	$B[A]$
$x\ z\ (\lambda x.(x\ y))$	$\{x, y, z\}$	$\{x\}$
$z\ (\lambda y.(\lambda x.(x\ y)))$	$\{z\}$	$\{x, y\}$
$(\lambda y.(x\ y))\ (\lambda x.(x\ y))$	$\{x, y\}$	$\{x, y\}$

Lösung 12.4. Eigentlich muss man nur die Definition abschreiben.

```

(define free-vars
  (lambda (expr)
    (cond ((var? expr) (list expr))
          ((app? expr) (union (free-vars (app-fn expr))
                              (free-vars (app-arg expr))))
          (else (remove-all (abstr-var expr)
                              (free-vars (abstr-body expr)))))))

(define bound-vars
  (lambda (expr)
    (cond ((var? expr) '())
          ((app? expr) (union (bound-vars (app-fn expr))
                              (bound-vars (app-arg expr))))
          (else (union (list (abstr-var expr))
                        (bound-vars (abstr-body expr)))))))

```

union stammt aus Aufgabe 8.12 und remove-all aus Aufgabe 8.8.

Lösung 12.6. Die neue Substitutionsfunktion sieht so aus:

```

(define subst-var
  (lambda (new var term)
    (cond ((eqv? term var) new)
          ((var? term) term)
          ((app? term)
           (build-app (subst-var new var (app-fn term))
                       (subst-var new var (app-arg term))))
          (else ;; must be an abstraction
           (cond ((or (eqv? var (abstr-var term))
                      (not (member? var (free-vars term))))
                  ;; nothing to do
                  term)
                 ((not (member (abstr-var term) (free-vars new)))
                  ;; no variable capture possible
                  (build-abstr (abstr-var term)
                               (subst-var new var
                                           (abstr-body term))))
                 (else ;; we must rename
                  (let ((new-var
                        (fresh-var (abstr-var term)
                                   (union (free-vars new)
                                           (free-vars
                                            (abstr-body term))))
                        1)))
                    (build-abstr
                     new-var
                     (subst-var
                      new var
                      ;; first introduce fresh var in body
                      (subst-var new-var

```

```
(abstr-var term)
(abstr-body term)))))))))
```

Lösung 12.7. Meine Lösung sieht so aus:

```
(define alpha-equiv?-helper
  (lambda (expr1 expr2 vars1 vars2)
    (cond ((and (var? expr1) (var? expr2))
      (let ((pos1 (position expr1 vars1))
            (pos2 (position expr2 vars2)))
        (cond ((or pos1 pos2) (equiv? pos1 pos2))
              (else (equiv? expr1 expr2)))))
      ((and (app? expr1) (app? expr2))
        (and (alpha-equiv?-helper (app-fn expr1) (app-fn expr2)
                                   vars1 vars2)
              (alpha-equiv?-helper (app-arg expr1) (app-arg expr2)
                                   vars1 vars2)))
      ((and (abstr? expr1) (abstr? expr2))
        (alpha-equiv?-helper (abstr-body expr1) (abstr-body expr2)
                              (cons (abstr-var expr1) vars1)
                              (cons (abstr-var expr2) vars2)))
      (else #f))))

(define alpha-equiv?
  (lambda (expr1 expr2)
    (alpha-equiv?-helper expr1 expr2 '() '())))
```

Die im Hinweis angesprochene „Buchführung“ wird durch die Variablen `vars1` und `vars2` übernommen, die als [Stack](#) angelegt sind: Jede Abstraktion legt eine neue Variable auf den Stack, die Vorrang vor Bindungen „weiter außen“ hat. Variablen sind α -äquivalent, wenn sie entweder beide nicht im Stack auftauchen und identisch sind oder wenn sie an derselben Position im Stack gefunden werden.

Getestet habe ich so:

```
(define alpha-test-cases
  '((x x #t)
    (x y #f)
    (x (y z) #f)
    ((x y) (x y) #t)
    ((x y) (y x) #f)
    ((x (y z)) ((x y) z) #f)
    ((lambda (x) x) (lambda (x) x) #t)
    ((lambda (x) x) (lambda (x) y) #f)
    ((lambda (x) x) (lambda (y) y) #t)
    ((lambda (x) (lambda (x) x)) (lambda (y) (lambda (y) y))) #t)
    ((lambda (x) (lambda (x) x)) (lambda (y) (lambda (x) x))) #t)
    ((lambda (x) (lambda (y) x)) (lambda (y) (lambda (x) y))) #t)
    ((lambda (x) (lambda (y) x)) (lambda (a) (lambda (b) a))) #t)
```

```

((lambda (x) (y x)) (lambda (y) (y y)) #f)
((lambda (x) (z x)) (lambda (y) (z y)) #t)))

(define test-alpha
  (lambda ()
    (do ((cases alpha-test-cases (cdr cases)))
        ((null? cases))
        (let* ((case (first cases))
                (result (alpha-eqv? (first case) (second case))))
          (unless (eqv? result (third case))
            (printf "Compared ~A to ~A and got ~A instead of ~A.~%"
                     (first case) (second case)
                     result (third case)))))))

```

Siehe die Lösung von Aufgabe 11.5 zur Erläuterung des Testcodes. `let*` ist eine Variante von `let` (siehe Aufgabe 7.2), in der mehrere Variablen *nacheinander* gebunden werden. (In `let` werden sie *gleichzeitig* gebunden.)

Lösung 13.1. Die zweite in der ersten Zeile.

Lösung 13.2. Das sind v , w v und $\lambda z.w$ v .

Lösung 13.4. Das sind z , s z , s (s z) und s s z . Man beachte, dass nur die ersten drei Abstraktionen Church-Numerale sind und dass sich das dritte und das vierte Ergebnis unterscheiden. Das sollte die Rolle der Klammerung in Church-Numeralen verdeutlichen.

Lösung 13.5. Die Normalform des ersten Ausdrucks ist x y . Der zweite Ausdruck ist bereits in Normalform.

Lösung 13.7. Sie haben es hoffentlich mit Papier und Bleistift nachvollzogen oder zumindest in `DR LAMBDA`: 3 f liefert die dreimalige Anwendung von f als Abstraktion, d. h. statt $f(f(f\ x))$ erhält man $\lambda x.f(f(f\ x))$ – bzw. den dazu α -äquivalenten Ausdruck $\lambda n.f(f(f\ n))$.

Analog liefert 0 f die Abstraktion $\lambda n.n$, also die Identität I .

Lösung 13.10. $\lambda x\ y.x$ `SUCC` y . Nach den vorherigen Überlegungen war das hoffentlich einfach.

Lösung 13.11. $\lambda x\ y.x$ (`PLUS` y) 0 . Die Idee dahinter ist natürlich, dass man $x \cdot y$ berechnet, wenn man mit 0 anfängt und x -mal die Zahl y addiert. Die Klammern sind hier wesentlich, damit (`PLUS` y) die „Funktion“ ist, die x -mal ausgeführt wird.

Lösung 13.13. Wir betrachten dafür den λ -Ausdruck $\lambda a.3$ (4 a). Den Rumpf können wir aufgrund der Wirkungsweise von 3 (siehe Aufgabe 13.7) auch als

$$\lambda x.(4\ a)\ ((4\ a)\ ((4\ a)\ x))$$

schreiben. Lässt man die entbehrlichen Klammern weg, so hat man den Ausdruck $\lambda x.4\ a\ (4\ a\ (4\ a\ x))$, woraus durch Reduktion⁷ offenbar nacheinander

$\lambda x.4\ a\ (4\ a\ (a\ (a\ (a\ x))))$,

$\lambda x.4\ a\ (a\ (a\ (a\ (a\ (a\ (a\ x))))))$ und schließlich

$\lambda x.a\ (a\ (a\ (a\ (a\ (a\ (a\ (a\ (a\ (a\ x))))))))$

wird. Fügt man das anfangs weggelassene λa wieder hinzu, so hat man (bis auf α -Äquivalenz) das Church-Numeral 12.

Lösung 13.14. F ist α -äquivalent zum Church-Numeral 0. Und DRLAMBDA wird auch 0 anzeigen, wenn Sie F eingeben (weil sich das System für einen der beiden Werte entscheiden muss und Numerale bei ihm Vorrang vor Abkürzungen haben).

Diese Übereinstimmung zeigt unter anderem, dass der klassische Lambda-Kalkül keine Typen kennt und dass λ -Ausdrücke auch keine ihnen innewohnende Bedeutung haben. Die bekommen sie nur dadurch, wie sie verwendet werden. Man kann mit $\lambda x\ y.y$ rechnen und dann verhält sich dieser λ -Ausdruck wie die Zahl null. Man kann damit aber auch die Dinge machen, die wir gleich besprechen werden, und dann verhält sich der Ausdruck wie ein Wahrheitswert.

Lösung 13.15. Das sollte eine Routineaufgabe für Sie gewesen sein.

	λ -Ausdruck	β -Normalform
AND T T	$(\lambda x\ y.x\ y\ (\lambda x\ y.y))\ (\lambda x\ y.x)\ (\lambda x\ y.x)$	$(\lambda x\ y.x)\ T$
AND T F	$(\lambda x\ y.x\ y\ (\lambda x\ y.y))\ (\lambda x\ y.x)\ (\lambda x\ y.y)$	$(\lambda x\ y.y)\ F$
AND F T	$(\lambda x\ y.x\ y\ (\lambda x\ y.y))\ (\lambda x\ y.y)\ (\lambda x\ y.x)$	$(\lambda x\ y.y)\ F$
AND F F	$(\lambda x\ y.x\ y\ (\lambda x\ y.y))\ (\lambda x\ y.y)\ (\lambda x\ y.y)$	$(\lambda x\ y.y)\ F$

Lösung 13.18. Ich hatte an $\lambda f.(\lambda x\ y.f\ y\ x)$ gedacht. Probieren Sie aus, was passiert, wenn man diese Abstraktion auf T bzw. F anwendet.

Lösung 13.19. Das geht mit $\lambda x.(Z\ x)\ 2\ 3$. Der „Witz“ dabei ist, dass der Wahrheitswert, den $(Z\ x)$ liefert, selbst eine Abstraktion ist, die sich von den beiden Argumenten 2 bzw. 3 das „richtige“ herauspicks.

Lösung 13.20. Ich dachte an so etwas:

```
((Z n) (lambda () (do-this (* n n)))
      (lambda () (do-that (+ n 3))))
```

Die beiden `do`-Ausdrücke werden jeweils mit `lambda` „eingewickelt“ um ihre Ausführung gewissermaßen zu verzögern. Man nennt solche Konstrukte auch scherzhaft *thunks*.

Lösung 13.21. Addition ist wiederholtes Anwenden von SUCC, Subtraktion ist sinnvollerweise wiederholtes Anwenden von PRED. Man muss nur auf die Reihenfolge achten. Es sollte im Prinzip so aussehen: $\lambda a\ b.b\ \text{PRED}\ a$.

⁷Wir reduzieren hier absichtlich nicht in Standardreihenfolge, weil dadurch hoffentlich klarer wird, was geschieht.

Lösung 13.22. Zwei (natürliche) Zahlen a und b sind genau dann gleich, wenn die Differenzen $a - b$ und $b - a$ beide verschwinden. Und das gilt zum Glück auch für unsere Version der Subtraktion. Also klappt es so:

$$\lambda a b. \text{AND } (Z \text{ (MINUS } a b)) (Z \text{ (MINUS } b a))$$

Lösung 13.23. Man fängt am besten innen mit der Abstraktion $\lambda g. \lambda h. h (g f)$ an, die ich für diese Lösung mit V abkürze. Der Rumpf von V (also quasi der „Rückgabewert“) ist eine Abstraktion, die mit $\lambda h. h$ anfängt. Dahinter steht $(g f)$, d. h., das Argument g von V wird als „Funktion“ verwendet, die auf f angewendet wird.

Das Argument von V ist $\lambda u. n$ – eine **konstante „Funktion“**, die ihr Argument, in diesem Fall f , ignoriert. Die Anwendung $V (\lambda u. n)$ wird daher zu $\lambda h. h n$.

Was passiert, wenn man V auf dieses Ergebnis $\lambda h. h n$ erneut anwendet? Nun ist $\lambda h. h n$ die „Funktion“, die auf f angewendet wird. Das Resultat dieser Anwendung ist $f n$, das Resultat der Anwendung von V also $\lambda h. h (f n)$.

Wendet man darauf wieder V an, so ergibt sich $\lambda h. h (f (f n))$ und so geht es weiter. (Überzeugen Sie sich selbst davon!) In PRED wird V m -mal auf $\lambda u. n$ angewendet, so dass sich eine $m - 1$ -malige Anwendung von f auf n ergibt. I ist nur dazu da, zum Schluss das h zu entfernen.

Es ist noch zu klären, was passiert, wenn m das Numeral 0 ist. Aber das bekommen Sie sicher selbst hin.

Lösung 13.24. Ich würde es so machen:

$$\text{FACT} = \lambda r n. (Z n) 1 (\text{TIMES } n (r (\text{PRED } n)))$$

Lösung 13.25. Im Prinzip wortwörtlich übernommen:

$$\text{QUOTIENT} = \lambda r a b. (Z (\text{MINUS } (\text{SUCC } a) b)) 0 (\text{SUCC } (r (b \text{ PRED } a) b))$$

Der Ausdruck $Z (\text{MINUS } (\text{SUCC } a) b)$ realisiert den Test, der in der SCHEME-Version $(< a b)$ hieß.

Lösung 13.26. Am Anfang von Abschnitt 12 wurde der Unterschied zwischen dem Lambda-Kalkül und dem Evaluationsmodell von SCHEME besprochen. Ruft man beispielsweise $(Y f)$ mit einer SCHEME-Funktion f auf, so wird die Funktion $(\text{lambda } (x) (f (x x)))$, die wir zur Abkürzung g nennen, auf g angewendet, d. h., $(g g)$ muss evaluiert werden. Das liefert den Funktionsaufruf $(f (g g))$. Um den auszuführen, wird zuerst das Argument evaluiert. Das ist jedoch wieder $(g g)$ und damit kommt es zu einer **Endlosschleife**.

Lösung 13.27. Das müsste diese Form haben:

```
(define sum
  (lambda (r)
```

```
(lambda (n)
  (if (zero? n) 0 (+ n (r (sub1 n))))))
```

Aufrufen kann man dann z. B. so: ((Y sum) 10).

Lösung 13.28. Das sollte in etwa so aussehen:

```
(define Y
  (lambda (y)
    ((lambda (x)
      (y (lambda (v) ((x x) v))))
     (lambda (x)
      (y (lambda (v) ((x x) v)))))))

((Y sum) 10)
```

Der Ausdruck (x x), dessentwegen es in Aufgabe 13.26 nicht geklappt hat, wird hier durch den verzögerten Aufruf (lambda (v) ((x x) v)) ersetzt.

Lösung 14.1. Das ist (add1 (add1 (add1 (add1 (add1 zero))))).

Lösung 14.2. Das ist einfach zero.

Lösung 14.3. Nein. Atome fangen immer mit einem Hochstrich an. 'pups ist ein Atom.

Lösung 14.4. 'r2d2 ist kein Atom, weil das Zeichen 2 weder ein Buchstabe noch ein Bindestrich ist. Die anderen beiden Ausdrücke sind Atome. Sie können das auch selbst überprüfen, indem Sie die entsprechenden Zeichenketten einfach in PIE eingeben.

Lösung 14.5. Ja. Wie weiter oben erklärt wurde, ist die Darstellung 2 nur eine „Komfortfunktion“ von PIE. 2 ist der Nat-Ausdruck (add1 (add1 zero)).

Lösung 14.6. In beiden Fällen ist die Antwort *nein*. Aber offiziell wissen Sie das noch nicht! Bisher haben wir nur Regeln dafür kennengelernt, wann zwei Nat- oder Atom-Ausdrücke gleich sind. Wir kennen aber weder Regeln dafür, wenn solche Ausdrücke *nicht* gleich sind, noch stand irgendwo, dass die obigen Regeln die einzigen sind, aus denen sich Gleichheit schließen lässt.

Mehr dazu in den folgenden Abschnitten.

Lösung 14.7. Sie können es ausprobieren. Sie werden jeweils die Fehlermeldung *bad syntax* sehen. Das bedeutet, dass es in PIE weder negative Zahlen noch Fließkommazahlen gibt. Im Sinne der durch PIE realisierten Typentheorie handelt es sich also gar nicht um Ausdrücke. Das heißt aber nicht, dass man solche Zahlen in Typentheorien nicht behandeln kann. Es heißt nur, dass die Typentheorie von PIE eine sehr einfache ist, die lediglich für Lehrzwecke entwickelt wurde.

Lösung 15.1. Ihre Eingaben sollten folgendermaßen aussehen. Damit können Sie dann auch überprüfen, ob das herauskommt, was Sie erwarteten.

```

(car
  (the (Pair Nat Nat)
    (cons 23 42)))
(cdr
  (the (Pair (Pair Nat Nat) Atom)
    (cons
      (the (Pair Nat Nat) (cons 42 23))
      'quark)))
(cdr
  (car
    (the (Pair (Pair Nat Nat) Atom)
      (cons
        (the (Pair Nat Nat) (cons 42 23))
        'quark))))
(the (Pair Nat Nat)
  (cons
    (car (the (Pair Nat Nat) (cons 23 42)))
    (cdr (the (Pair Nat Nat) (cons 23 42))))))

```

Lösung 15.2. Nein. Auch `Nat` und `Atom` sind Typ-Konstruktoren. Es sind wie `zero` Konstruktoren ohne Argumente.

Lösung 15.3. Die Ausdrücke sind jeweils nicht identisch (bezüglich ihres Typs), weil sie bereits in Normalform vorliegen und diese Normalformen unterschiedlich sind. (Dabei ist zu beachten, dass 3 für `(add1 (add1 (add1 zero)))` steht.)

Lösung 15.4. Die Normalformen `(add1 zero)` und `(add1 (add1 zero))` sind offenbar verschieden. Also ist die Antwort *nein*.

Lösung 15.5. Das sollte wie folgt aussehen:

```

(car (the (Pair U U)
  (cons Nat Atom)))
(the (Pair U U)
  (cons
    (cdr (the (Pair U U)
      (cons Nat Atom)))
    (car (the (Pair U U)
      (cons Nat Atom))))))

```

Im ersten Fall erhält man den Typ `Nat`, im zweiten `(cons Atom Nat)`. Letzteres ist *kein* Typ, sondern ein Ausdruck vom Typ `(Pair U U)`.

Lösung 15.6. Nach unserem bisherigen Wissensstand müsste `(Pair Nat Atom)` der Typ sein. Wir werden jedoch noch sehen, dass auch [andere Typen](#) infrage kommen können. (Erinnern Sie sich an die Frage, warum PIE bei solchen Eingaben nicht automatisch den Typ angibt.)

Lösung 15.7. Nein, das war eine Fangfrage. `'(3 4)` gibt es in PIE gar nicht. Wie hinter der Beschreibung des Typs `Atom` erklärt wurde, tauchen Hochstriche in

Pie nur als erstes Zeichen von Atomen auf. Bringen Sie nicht PIE und SCHEME durcheinander!

Abgesehen davon sollte durch die bisherigen Ausführungen klar geworden sein, dass es ohne Angabe eines Typs ohnehin keinen Sinn ergibt, zu fragen, ob X „derselbe Ausdruck“ wie Y ist. Siehe dazu das Urteil (J3).

Lösung 16.1. Beim Test auf Gleichheit scheint der Variablenname keine Rolle zu spielen. Und tatsächlich ist es so, dass α -Äquivalenz im Sinne von Aufgabe 12.7 nichts an der Gleichheit zweier Funktionen ändert. Mit anderen Worten: Sind die Funktionen f_1 und f_2 gleich und ist f_3 α -äquivalent zu f_2 , dann sind auch f_1 und f_3 gleich.

Wir wissen damit aber noch *nicht*, wann zwei Funktionen von T_1 nach T_2 derselbe Ausdruck vom Typ $(\rightarrow T_1 T_2)$ sind. Das kommt [später](#).

Lösung 16.2. Diese Frage ist etwas gemein, weil Sie sie zu diesem Zeitpunkt noch gar nicht beantworten können. Wichtig ist, dass Sie hoffentlich darüber nachgedacht haben und dass Ihnen klar geworden ist, dass man den Typ nicht in der Form $(\rightarrow \dots)$ darstellen kann. Der Typ von `car` ist offenbar „flexibel“ und passt sich dem Typ des Arguments an.

Wir [werden noch lernen](#), wie wir ähnliche Funktionen – allerdings mit einer etwas anderen Syntax – selbst schreiben können.

Lösung 16.3. Ja, die `claim`-Deklaration wäre nur etwas länglicher geworden:

```
(claim cadr
  (-> (Pair
      (Pair Nat Nat)
      (Pair Nat Nat))
      Nat))
```

Grundsätzlich ist es so, dass man komplett ohne `define` (und `claim`) auskommen könnte, weil damit lediglich Abkürzungen eingeführt werden. Man könnte stattdessen immer die Ausdrücke eingeben, die abgekürzt werden. Es würde nur sehr viel Tipparbeit nach sich ziehen ...

Lösung 16.4. Der erste Eliminator sieht so aus:

```
(claim car-APair (-> APair Atom))

(define car-APair
  (lambda (p)
    (elim-APair p (lambda (one two) one))))
```

Die Funktion, die `elim-APair` übergeben wird (und deren Typ wir offenbar nicht explizit angeben müssen), ignoriert einfach ihr zweites Argument. Wie die Funktion `cdr-APair` aussehen muss, sollte nun wohl klar sein.

Wir haben hier gelernt, dass man `car` und `cdr` im Prinzip nicht bräuchte, wenn man einen Eliminator „`elim-Pair`“ hätte, der wie `elim-APair` arbeitet, aber für alle Paare funktioniert. Wir werden [später](#) sehen, wie man so etwas programmieren könnte.

Lösung 17.1. Im Gegensatz zum Beispiel `positive?` wird hier das Argument nicht ignoriert.

```
(claim sub1 (-> Nat Nat))

(define sub1
  (lambda (n)
    (which-Nat n
      0
      (lambda (n-1) n-1))))
```

Lösung 17.2. Das war auch nicht schwer, oder? Wir brauchen diese Funktion allerdings [später](#) noch.

```
(claim switch (-> Nat Nat))

(define switch
  (lambda (n)
    (which-Nat n 1 (lambda (n-1) 0))))
```

Lösung 17.3. Das war hoffentlich nur eine Fingerübung.

```
(define which-Nat
  (lambda (target base step)
    (cond ((zero? target) base)
          (else (step (sub1 target)))))
```

Lösung 17.4. In SCHEME hätte man übrigens in der Definition von `add3` die letzten beiden Zeilen (also die [anonyme Funktion](#)) durch `add1` ersetzen können. In PIE geht das nicht, weil `add1` ein „eingebauter“ [Konstruktor](#) und keine Funktion ist. Aber das hier ist möglich:

```
(claim add-one (-> Nat Nat))
(define add-one
  (lambda (n) (add1 n)))

(claim add3 (-> Nat Nat))
(define add3
  (lambda (n) (iter-Nat n 3 add-one)))
```

Bei mir sieht das dann so aus:

$$(\text{add3 } 0) \rightarrow (\text{iter-Nat } 0 \ 3 \ \text{add-one}) \rightarrow 3$$

⁷Ich zeige das in erster Linie, um den folgenden Teil etwas abzukürzen.

```
(add3 1) → (iter-Nat 1 3 add-one)
          → (add-one (iter-Nat 0 3 add-one))
          → (add-one 3) → 4
(add3 2) → (iter-Nat 2 3 add-one)
          → (add-one (iter-Nat 1 3 add-one))
          → (add-one 4) → 5
```

add3 ist offenbar schlicht und einfach eine Funktion, die 3 zu einer natürlichen Zahl addiert.

Lösung 17.5. Man vergleiche diesen Code mit dem von add3:

```
(claim + (-> Nat Nat Nat))    ;; add3: (-> Nat Nat)

(define +
  (lambda (k n)                ;; add3: (n)
    (iter-Nat n
      k                          ;; add3: 3
      (lambda (step-1)
        (add1 step-1))))))
```

Lösung 17.6. Das sieht bei mir so aus:

```
(claim * (-> Nat Nat Nat))

(define *
  (lambda (k n)
    (iter-Nat n
      0
      (lambda (step-1)
        (+ k step-1))))))
```

Lösung 17.7. Mit denselben Buchstaben wie in Aufgabe 9.7:

```
(claim - (-> Nat Nat Nat))

(define -
  (lambda (a b)
    (iter-Nat b
      a
      (lambda (step-1)
        (sub1 step-1))))))
```

Lösung 17.8. Ich hatte mir das so vorgestellt:

```
(claim parity (-> Nat Nat))

(define parity
```

```
(lambda (n)
  (iter-Nat n 0 switch)))
```

Lösung 17.9. Man kann fast die Lösung von Aufgabe 17.3 kopieren.

```
(define iter-Nat
  (lambda (target base step)
    (cond ((zero? target) base)
          (else (step (iter-Nat (sub1 target) base step))))))
```

Man hätte es mit speziellen RACKET-Funktionen (die nicht zum SCHEME-Standard gehören), aber auch so schreiben können:

```
(define iter-Nat
  (lambda (target base step)
    (for/fold ((result base))
              ((n (range target)))
              (step result))))
```

(`range n`) liefert die Liste der Zahlen von 0 bis $n - 1$. Und `for/fold` entspricht eher den `for-Schleifen`, die Sie aus konventionellen Sprachen kennen: Am Anfang wird `result` auf den Wert `base` gesetzt, `n` durchläuft der Reihe nach die Zahlen in (`range target`) und in jedem Schritt wird der Wert in `result` in (`step result`) geändert.

Lösung 17.10. Das Prinzip ist dasselbe wie bei `sum`:

```
(claim fact (-> Nat Nat))

(define fact
  (lambda (n)
    (rec-Nat n
      1
      (lambda (n-1 step-1)
        (* (add1 n-1) step-1)))))
```

Lösung 17.11. Der Rekursionschritt muss einfach einen Teil der Information, die `rec-Nat` liefert, ignorieren.

```
(claim my-which-Nat (-> Nat Nat (-> Nat Nat) Nat))

(define my-which-Nat
  (lambda (target base step)
    (rec-Nat target
      base
      (lambda (n-1 step-1)
        (step n-1)))))
```

Lösung 17.12. Das gleiche Prinzip wie in Aufgabe 17.11, nur dass diesmal die andere Hälfte der Information ignoriert wird.

```
(claim my-iter-Nat (-> Nat Nat (-> Nat Nat) Nat))

(define my-iter-Nat
  (lambda (target base step)
    (rec-Nat target
      base
      (lambda (n-1 step-1)
        (step step-1))))))
```

Lösung 17.13. Die Variable `n` in der letzten Zeile wird durch das `lambda` gebunden. Es ist also immer dieselbe Zahl, die in jedem Schritt addiert wird, während in der Version mit `rec-Nat` der erste Summand in jedem Schritt ein anderer ist. Die Funktion Ihres Kommilitonen addiert also n -mal die Zahl n zu null, d. h., sie berechnet dasselbe wie $(* n n)$ aus Aufgabe 17.6.

Lösung 17.14. Meine Lösung sieht so aus:

```
(claim rec-Nat-helper (-> Nat Nat (-> Nat Nat Nat)
                      (Pair Nat Nat)))

(define rec-Nat-helper
  (lambda (target base step)
    (iter-Nat target
      (the (Pair Nat Nat) (cons 0 base))
      (lambda (step-1)
        (cons (add1 (car step-1))
              (step (car step-1) (cdr step-1)))))))

(claim my-rec-Nat (-> Nat Nat (-> Nat Nat Nat) Nat))

(define my-rec-Nat
  (lambda (target base step)
    (cdr (rec-Nat-helper target base step))))
```

Die eigentlich Arbeit wird von `rec-Nat-helper` erledigt. Der weitergegebene Wert ist dabei nicht einfach eine Zahl, sondern ein Paar von Zahlen, wobei die erste Zahl den Stand der Rekursionsvariablen konserviert und die zweite das eigentliche Resultat ist. `my-rec-Nat` extrahiert diese Komponente dann nur noch mit `cdr`. Die Funktion `step` agiert dabei genau wie bei `rec-Nat`. Sie muss von den Paaren nichts „wissen“.

Lösung 18.1. Der im Hinweis angesprochene Ergebnistyp heißt hier `X`.

```
(claim elim-Pair
  (Pi ((T1 U) (T2 U) (X U))
    (-> (Pair T1 T2) (-> T1 T2 X) X)))
```

```
(define elim-Pair
  (lambda (T1 T2 X p fn)
    (fn (car p) (cdr p))))

(claim car-Pair
  (Pi ((T1 U) (T2 U))
    (-> (Pair T1 T2) T1)))

(define car-Pair
  (lambda (T1 T2 p)
    (elim-Pair T1 T2 T1 p (lambda (one two) one))))

(car-Pair Nat Atom (cons 2 'two))
```

Lösung 18.2. Das sollte in etwa so aussehen:

```
(claim length (Pi ((T U)) (-> (List T) Nat)))

(define length
  (lambda (T list)
    (rec-List list
      0
      (lambda (first rest previous)
        (add1 previous)))))

(length Nat (:: 23 (:: 42 (:: 99 nil))))
```

Lösung 18.3. Hier meine Lösung:

```
(claim append (Pi ((T U)) (-> T (List T) (List T))))

(define append
  (lambda (T X list)
    (rec-List list
      (:: X nil)
      (lambda (first rest previous)
        (:: first previous)))))
```

Für die LISP-Experten: Ja, ich weiß, dass der Name `append` normalerweise für eine andere Funktionalität steht ...

Lösung 18.4. Die hätte so ausgesehen: `(lambda (lst) Nat)`. `rec-List` ist quasi für die einfachen Fälle zuständig, in denen diese Funktion konstant ist.

Lösung 18.5. So klappt es zum Beispiel:

```
(claim repeat (Pi ((T U) (n Nat) (x T)) (Vec T n)))

(define repeat
```

```
(lambda (T n x)
  (ind-Nat n
    (lambda (k) (Vec T k))
    vecnil
    (lambda (n-1 step-1)
      (vec:: x step-1))))))
```

Lösung 19.1. Bei plus verläuft die Rekursion über das erste Argument k , bei + über das zweite n .

Lösung 19.3. Weil das Argument $(\text{add1 } n)$ nicht zero ist, tritt in rec-Nat der zweite Fall ein. Aus dem Term wird also der folgende, wobei nebenbei auch noch $(\text{add2 } n)$ vereinfacht wurde.

```
(+ (rec-Nat n
  (add1 (add1 n))
  (lambda (n-1 step-1)
    (+ step-1 (add1 n-1))))
  (add1 n))
```

Beachten Sie das grüne +. Ab jetzt zeigt die farblich markierte Stelle immer, wie es weitergeht. Es wird nun in die Definition von + eingesetzt:

```
(iter-Nat (add1 n)
  (rec-Nat n
    (add1 (add1 n))
    (lambda (n-1 step-1)
      (+ step-1 (add1 n-1))))
  (lambda (step-1)
    (add1 step-1)))
```

Jetzt haben wir außen iter-Nat mit $(\text{add1 } n)$ und können einen Iterationsschritt durchführen.

```
(add1 (iter-Nat n
  (rec-Nat n
    (add1 (add1 n))
    (lambda (n-1 step-1)
      (+ step-1 (add1 n-1))))
  (lambda (step-1)
    (add1 step-1))))
```

Weiter innen taucht noch ein + auf, das wir wieder in die Definition einsetzen:

```
(add1 (iter-Nat n
  (rec-Nat n
    (add1 (add1 n))
    (lambda (n-1 step-1)
      (iter-Nat (add1 n-1)
```

```

      step-1
      (lambda (step-1)
        (add1 step-1))))
    (lambda (step-1)
      (add1 step-1)))

```

Und da haben wir ein letztes Mal ein `iter-Nat`, das sich weiter vereinfachen lässt.

```

(add1 (iter-Nat n
  (rec-Nat n
    (add1 (add1 n))
    (lambda (n-1 step-1)
      (add1 (iter-Nat n-1
        step-1
        (lambda (step-1)
          (add1 step-1))))))
    (lambda (step-1)
      (add1 step-1))))

```

Das Ergebnis können Sie in PIE überprüfen, indem Sie den Ausdruck aus der Aufgabenstellung in `(the (-> Nat Nat) (lambda (n) ...))` einsetzen. Sie können aber auch Schritt für Schritt vorgehen und Ihre Schritte mit

```
(check-same (-> Nat Nat) (lambda (n) ...) (lambda (n) ...))
```

vergleichen.

Ich habe mich übrigens beim Lösen mehrfach verheddert. Aber ich habe nicht aufgegeben, bis die Lösung schließlich richtig war. Und Sie?

Lösung 20.1. Das wäre dann eine [Aussageform](#) mit der [Individuenvariable](#) `n`.

Lösung 20.2. Was muss herauskommen, wenn `n` der Ausdruck `zero` ist? Der Ausdruck muss den Type `(= Nat (add2 0) ((+ 2) 0))` haben. Wenn man das vereinfacht,⁸ wird daraus `(= Nat 2 2)`. Der Konstruktor für diesen Typ ist `same`, also ist die Antwort `(same 2)`.

Lösung 20.4. `x` und `y` sind `(add2 n-1)` und `((+ 2) n-1)` und `add1` spielt die Rolle von `f`.

Lösung 20.6. Zum Beispiel so:

```
(= Nat (+ a (+ b c)) (+ (+ a b) c))
```

Oder auch so:

```
(= Nat ((+ x) ((+ y) z)) ((+ ((+ x) y)) z))
```

Lösung 20.7. Das erste TODO muss laut PIE den Typ

⁸Sie können es auch einfach im Interaktionsfenster eingeben.

```
(= Nat
  (iter-Nat b
    (the Nat a)
    (lambda (step-1) (add1 step-1))))
(iter-Nat b
  (the Nat a)
  (lambda (step-1) (add1 step-1))))
```

haben. Das entspricht genau dem Code der Funktion `+`. Die gesuchte Ersetzung ist also einfach `(same (+ a b))`. Beim zweiten TODO sieht man durch die von PIE vorgenommene Vereinfachung der Ausdrücke, dass in beiden Fällen einfach `add1` hinzukommt. Wir können also wie eben unsere Hilfsfunktion `incr` verwenden.

```
(define +-assoc
  (lambda (a b c)
    (ind-Nat c
      (lambda (c)
        (= Nat (+ a (+ b c)) (+ (+ a b) c)))
      (same (+ a b))
      (lambda (c-1 almost) (cong almost incr)))))
```

Lösung 20.8. So sieht meine Version aus:

```
(claim zero-+
  (Pi ((n Nat))
    (= Nat (+ 0 n) n)))

(define zero-+
  (lambda (n)
    (ind-Nat n
      (lambda (n) (= Nat (+ 0 n) n))
      (same 0)
      (lambda (n-1 almost) (cong almost incr)))))
```

Lösung 20.9. Wird es langweilig? Es kommen noch schwierigere Beweise ...

```
(claim add1-+
  (Pi ((a Nat) (b Nat))
    (= Nat (+ (add1 a) b) (add1 (+ a b)))))

(define add1-+
  (lambda (a b)
    (ind-Nat b
      (lambda (b) (= Nat (+ (add1 a) b) (add1 (+ a b)))))
    (same (add1 a))
    (lambda (n-1 almost) (cong almost incr)))))
```

Lösung 20.10. x ist hier `(add1 (+ a b))`, y ist `(add1 (+ b a))` und z ist der Ausdruck `(+ (add1 b) a)`.

Lösung 20.11. Den ersten Schritt erreicht man einfach durch Eingabe im Interaktionsfenster:

```
(Pi ((n Nat)) (= Nat (add1 (add1 (* n 2))) (* (add1 n) 2)))
```

Wenn man scharf genug hinschaut, erkennt man in den `iter-Nat`-Ausdrücken die Addition – was auch logisch ist, wenn man sich die Definition von `*` anschaut. Der Rest wird hoffentlich durch die folgende Tabelle klar. In der habe ich statt `add1` zur Abkürzung `1+` geschrieben⁹ und mit `...` ist jeweils die Zeile darüber gemeint.

zu zeigen	$(1+ (1+ (+ n n)))$	$(1+ (+ (1+ n) n))$
<code>(add1-+ n n)</code>	<code>(+ (1+ n) n)</code>	<code>(1+ (+ n n))</code>
<code>(cong ... incr)</code>	<code>(1+ (+ (1+ n) n))</code>	<code>(1+ (1+ (+ n n)))</code>
<code>(symm ...)</code>	<code>(1+ (1+ (+ n n)))</code>	<code>(1+ (+ (1+ n) n))</code>

Damit wird die Lösung ganz einfach:

```
(define twice
  (lambda (n)
    (symm (cong (add1-+ n n) incr))))
```

Lösung 20.12. Man muss sich erst einmal an die etwas schräge „Denkweise“ von `replace` gewöhnen. Es sollte jedenfalls so aussehen:

```
(claim my-trans
  (Pi ((T U) (from T) (bridge T) (to T))
    (-> (= T from bridge) (= T bridge to) (= T from to))))

(define my-trans
  (lambda (T from bridge to E1 E2)
    (replace E2
      (lambda (x) (= T from x))
      (replace E1
        (lambda (x) (= T from x))
        (same from))))))

(claim my-cong
  (Pi ((T U) (S U) (from T) (to T))
    (E (= T from to)) (f (-> T S)))
    (= S (f from) (f to))))

(define my-cong
  (lambda (T S from to E f)
    (replace E
      (lambda (x) (= S (f from) (f x)))
      (same (f from))))))
```

Lösung 21.1. Es ist ganz einfach:

⁹Der Name der entsprechenden Funktion in COMMON LISP.

```
(define mp
  (lambda (T1 T2 A->B A)
    (A->B A)))
```

Diese Aufgabe sollte nur noch einmal das demonstrieren, was im Absatz davor erklärt wurde.

Lösung 21.2. Ja, der gesuchte Typ ist $(\text{Pair } T_1 T_2)$.

Lösung 21.3. Hier sind Aussage und Beweis.

```
(claim and-comm
  (Pi ((T1 U) (T2 U)) (-> (Pair T1 T2) (Pair T2 T1))))

(define and-comm
  (lambda (T1 T2 P1)
    (cons (cdr P1) (car P1))))
```

Sie können die Funktion nun ausprobieren:

```
> (and-comm Nat Atom (the (Pair Nat Atom) (cons 3 'three)))
(the (Pair Atom Nat)
  (cons 'three 3))
```

Aber das ist eigentlich nicht nötig. Die Tatsache, dass PIE die Definition von `and-comm` akzeptiert hat, ist bereits der Beweis, um den es geht!

Lösung 21.4. Im Prinzip ist das nur Jonglieren mit Paaren wie in Abschnitt 4.

```
(claim and-asso
  (Pi ((T1 U) (T2 U) (T3 U))
    (-> (Pair (Pair T1 T2) T3) (Pair T1 (Pair T2 T3)))))

(define and-asso
  (lambda (T1 T2 T3 P1)
    (cons (car (car P1)) (cons (cdr (car P1)) (cdr P1)))))
```

Lösung 21.5. Ein Beweis könnte so aussehen:

```
(claim and-impl-or
  (Pi ((T1 U) (T2 U)) (-> (Pair T1 T2) (Either T1 T2))))

(define and-impl-or
  (lambda (T1 T2 P)
    (left (car P))))
```

Man hätte statt `car` aber auch `cdr` und statt `left` dann `right` nehmen können.

Lösung 21.6. Man muss einfach [links und rechts vertauschen](#):

```
(claim or-comm
  (Pi ((T1 U) (T2 U)) (-> (Either T1 T2) (Either T2 T1))))

(define or-comm
  (lambda (T1 T2 E)
    (ind-Either E
      (lambda (E) (Either T2 T1))
      (lambda (L) (right L))
      (lambda (R) (left R)))))
```

Lösung 21.7. Das ist etwas aufwendiger als Aufgabe 21.6, weil man drei Fälle unterscheiden muss, die gewissermaßen verschachtelt sind, und weil man darauf achten muss, dass das Ergebnis den richtigen Typ hat.

```
(claim or-asso
  (Pi ((T1 U) (T2 U) (T3 U))
    (-> (Either (Either T1 T2) T3) (Either T1 (Either T2 T3)))))

(define or-asso
  (lambda (T1 T2 T3 E)
    (ind-Either E
      (lambda (E) (Either T1 (Either T2 T3)))
      (lambda (L) ; T1 oder T2 wahr
        (ind-Either L
          (lambda (E) (Either T1 (Either T2 T3)))
          (lambda (LL) (left LL)) ; T1 wahr
          (lambda (LR) (right (left LR))))) ; T2 wahr
      (lambda (R)
        (right (right R))))) ; T3 wahr
```

Lösung 21.8. Nach Definition von `ind-Either` muss das zweite Argument in diesem Fall als Ergebnis etwas vom Typ `U` liefern. Das **Universum** `U` hat aber selbst *nicht* den Typ `U`. In PIE wäre so eine Definition prinzipiell nicht möglich. Siehe auch Fußnote 6 auf Seite 69.

Lösung 21.9. Das hier würde klappen:

```
(claim payload-info
  (Pi ((E (Either Nat Atom))) (= (Either Nat Atom) E E)))

(define payload-info
  (lambda (E)
    (ind-Either E
      (lambda (E) (= (Either Nat Atom) E E))
      (lambda (L) (same (left L)))
      (lambda (R) (same (right R)))))
```

Je nach Argument haben die sich ergebenden Ausdrücke tatsächlich unterschiedliche (abhängige) Typen.

```
> (payload-info (left 23))
(the (= (Either Nat Atom) (left 23) (left 23))
  (same (left 23)))
> (payload-info (right 'quark))
(the (= (Either Nat Atom) (right 'quark) (right 'quark))
  (same (right 'quark)))
```

Lösung 21.10. Steht der Typ T für die Aussage A , dann steht $(\rightarrow T \text{ Absurd})$ für $\neg A$ und $(\rightarrow (\rightarrow T \text{ Absurd}) \text{ Absurd})$ für $\neg\neg A$. Also muss es so aussehen:

```
(claim double-neg
  (Pi ((T U))
    ( $\rightarrow$  T ( $\rightarrow$  ( $\rightarrow$  T Absurd) Absurd))))

(define double-neg
  (lambda (T x)
    (lambda (f) (f x))))
```

Die innere Funktion (letzte Zeile) muss einen Ausdruck vom Typ Absurd zurückgeben. Eigentlich geht das nicht. Aber sie erwartet als Argument eine Funktion f vom Typ $(\rightarrow T \text{ Absurd})$. Setzt sie in diese den Beweis x für T ein, so erhält sie den gewünschten Ausdruck.

Natürlich gibt es so ein f nicht, wenn es einen Beweis für T gibt. Aber für das System ist es egal, solange der Ausdruck korrekt typisiert ist.

Lösung 21.11. Das sieht wie folgt aus:

```
(claim impl-as-or
  (Pi ((T1 U) (T2 U))
    ( $\rightarrow$  (Either T2 ( $\rightarrow$  T1 Absurd)) ( $\rightarrow$  T1 T2))))

(define impl-as-or
  (lambda (T1 T2 E)
    (ind-Either E
      (lambda (E) ( $\rightarrow$  T1 T2))
      (lambda (L) (lambda (x) L))
      (lambda (R) (lambda (x) (ind-Absurd (R x) T2))))))
```

In beiden Fällen (left oder right) müssen wir eine Funktion vom Typ $(\rightarrow T1 T2)$ zurückgeben. Das Argument vom Typ $T1$ heißt x . Der „Witz“ im zweiten Fall (letzte Zeile) ist, dass wir gar nicht sagen, *was* wir zurückgeben. `ind-Absurd` garantiert lediglich, dass es sich um einen Ausdruck vom Typ $T2$ handelt. Und mehr braucht das System nicht, um zufrieden zu sein.

Lösung 21.12. Hier ist meine Lösung:

```
(claim indirect*
  (Pi ((T1 U) (T2 U))
```

```

      (-> (-> T1 T2) (-> (-> T2 Absurd) (-> T1 Absurd))))))

(define indirect*
  (lambda (T1 T2 f)
    (lambda (g)
      (lambda (x)
        (g (f x)))))))

```

Das Argument `f` macht aus einem Beweis von `T1` einen für `T2`. Und `g` macht aus einem Beweis von `T2` einen Absurd-Ausdruck. **Komponiert** man die beiden Funktionen, erhält man wie gewünscht eine, die `T1`-Ausdrücke zu Absurd-Ausdrücken macht.

Lösung 21.13. (`Even 10`) ist der folgende Typ:

```
(Sigma ((k Nat)) (= Nat 10 (double k)))
```

Die einzige Möglichkeit, ein Paar dieses Typs zu erhalten, sieht so aus:

```
(the (Even 10) (cons 5 (same 10)))
```

Lösung 21.14. Bis auf `add1` sieht das genauso aus.

```

(claim Odd (-> Nat U))

(define Odd
  (lambda (n)
    (Sigma ((k Nat))
      (= Nat n (add1 (double k))))))

```

Lösung 21.15. Das sieht sehr ähnlich aus. Man muss nur darauf achten, dass aus $n = 2k + 1$ die Gleichung $n + 1 = 2(k + 1)$ folgt, d. h., dass das `k` durch `(add1 k)` ersetzt werden muss.

```

(claim odd-succ (Pi ((n Nat))
  (-> (Odd n) (Even (add1 n)))))

(define odd-succ
  (lambda (n S)
    (cons (add1 (car S))
      (cong (cdr S) incr))))

```

In diesem Fall wird also aus

```
(cons k (= Nat n (add1 (double k))))
```

die Gleichung

```
(cons (add1 k) (= Nat (add1 n) (add1 (add1 (double k))))).
```

Wir wollen jedoch `(Even (add1 n))` haben und das sieht so aus:

```
(cons (add1 k) (= Nat (add1 n) (double (add1 k))))).
```

Die ganze Sache funktioniert, weil wir `double` so definiert haben, dass das System ohne Hilfe erkennt, dass die beiden Terme gleich sind.

Lösung 21.16. Ich habe es so gemacht:

```
(claim even-or-odd (Pi ((n Nat))
                      (Either (Even n) (Odd n))))

(define even-or-odd
  (lambda (n)
    (ind-Nat n
      (lambda (n) (Either (Even n) (Odd n)))
      (left (cons 0 (same 0)))
      (lambda (n-1 E-1)
        (ind-Either E-1
          (lambda (E) (Either (Even (add1 n-1)) (Odd (add1 n-1))))
          (lambda (L) (right (even-succ n-1 L)))
          (lambda (R) (left (odd-succ n-1 R))))))))
```

Lösung 21.17. Funktionen – also Ausdrücke, die mit dem Konstruktor `lambda` beginnen – binden Variablen. Das war ohnehin schon klar. Aber auch die **Typ-Konstrukturen** `Pi` und `Sigma` können Variablen binden.

Lösung 21.18. Ich habe das so gemacht:

```
(claim indirect
  (Pi ((T1 U) (T2 U))
    (-> (-> (-> T2 Absurd) (-> T1 Absurd)) (-> T1 T2))))

(define indirect
  (lambda (T1 T2 f)
    (lambda (x)
      (ind-Either (excl-middle T2)
        (lambda (E) T2)
        (lambda (L) L)
        (lambda (R) (ind-Absurd ((f R) x) T2))))))
```

Lösung 21.19. Das geht folgendermaßen:

```
(claim impl-as-or*
  (Pi ((T1 U) (T2 U))
    (-> (-> T1 T2) (Either T2 (-> T1 Absurd)))))

(define impl-as-or*
  (lambda (T1 T2 f)
    (ind-Either (excl-middle T1)
      (lambda (E) (Either T2 (-> T1 Absurd))
```

```
(lambda (L) (left (f L)))  
(lambda (R) (right R))))
```

Index

Die Begriffe am Anfang, für die es keine sinnvolle alphabetische Sortierung gibt, werden in der Reihenfolge aufgeführt, in der sie im Skript erscheinen.

Modus ponens, 100

$'$, 5

$+$, 6

$-$, 7

$*$, 7

$()$, 11

$=$, 19

$<$, 19

$>$, 19

$\leq$, 19

$\geq$, 19

$;$, 25

$\leftarrow$, 49

$\leftarrow$, 71

$\rightarrow$, 72

$::$, 80

$=$, 90

$\Rightarrow$, 100

$\wedge$, 100

$\vee$, 100

$\perp$, 102

$\top$, 102

abhängiger Typ, *siehe* Typ,
abhängiger

abs, 20

Abstraktion, 40

Absurd, 102

Ackermannfunktion, 37

add1, 15, 64

AGDA, 109

Allquantor, 92

α -äquivalent, 51

AND, 56

and, 19

anonyme Funktion, 15

Anwendung, 40, 72

Applikation, *siehe* Anwendung

Argument, 14

Assoziativität, 94, 100

Atom, 5, 65

Atom, 64

atom?, 20

Ausdruck, 6, 64

Aussagenlogik, 99

$A[x \leftarrow B]$, 49, 71

$B[A]$, 48

bedingte Anweisung, 18

β -Normalform, *siehe* Normalform

β -Redex, *siehe* Redex

β -Reduktionsschritt, *siehe*
Reduktionsschritt

Beweis, 90

Beweisassistent, 1, 63, 109

body, *siehe* Rumpf

boolean, 18

bound, *siehe* gebunden

- car, 9, 68
- cdr, 9, 68
- check-same, 66
- CHURCH, ALONZO, 39
- Church-Kodierung, 52
- Church-Numeral, 54
- claim, 67
- closure*, *siehe* Funktionsabschluss
- COMMON LISP, 3
- cond, 22
- cong, 93
- cons, 9, 67, 103
- Curry-Howard-Korrespondenz, 89
- Currying, 18, 54, 60, 74, 82

- Dedekind-Peano-Axiome, 53
- define, 8, 67
- Definition, rekursive, 13
- Definitionsfenster, 4
- Deklaration, 66
- dependent*, *siehe* abhängig
- Disjunktion, 100
- do, 25
- DRLAMBDA, 44
- DRRACKET, 4

- Either, 100
- Element, 5, 80
- Eliminator, 68
- else, 22
- endrekursiv, 27
- Entscheidungsproblem, 39
- eq?, 19
- eqv?, 19
- Evaluation, 6
- even?, 19
- ex falso quodlibet*, 103
- expression*, *siehe* Ausdruck
- expt, 7

- F, 56
- #f, 18
- $F[A]$, 48
- Fakultät, 34, 60, 79
- Fibonacci-Folge, 36
- first, 116
- First-Class-Objekt, 14, 71

- free*, *siehe* frei
- frei, *siehe* Variable, freie
- Funktion, 6, 43, 72
 - totale, 75
- funktionale Programmierung, 23
- Funktionsabschluss, 17

- ganze Zahl, *siehe* Zahl, ganze
- gaußsche Summenformel, 15, 60, 96, 111
- gebunden, *siehe* Variable, gebundene

- Hochstrich, 5, 64
- Homoikonizität, 14

- I, 54
- Identität, 44, 54
- if, 20
- Implikation, 100
- ind-Absurd, 103
- ind-Either, 101
- ind-List, 85
- ind-Nat, 86
- Induktion, 53, 94, 111
- integer*, *siehe* Zahl, ganze
- integer?, 19
- Interaktionsfenster, 4
- Internalisierung, 91
- intuitionistische Logik, *siehe* Logik, intuitionistische
- iter-Nat, 77
- Iteration, 37, 78

- judgment*, *siehe* Urteil
- Junktoren, 19, 99

- KI, 1, 113
- Klammerregeln, 41
- klassische Logik, *siehe* Logik, klassische
- Kommentar, 25
- Kommutativität, 95, 100
- Komponente, 67
- Kongruenz, 93
- Konjunktion, 100
- konstante Funktion, 44
- Konstruktor, 64
- Kontradiktion, 102

- lambda, 14, 72
- Lambda-Ausdruck, 40
- λ -Ausdruck, *siehe* Lambda-Ausdruck
- Lambda-Kalkül, 39
- LEAN, 1, 63, 84, 97, 109
- leere Liste, 11
- leere Summe, 115
- left, 100
- Leibniz-Gesetz, 93
- length, 27
- let, 25
- linksassoziativ, 41, 115
- LISP, 3
- List, 80
- list, 11
- list-ref, 116
- list?, 31
- Liste, 5, 12, 80
- Listener, 4
- Logik, 99
 - intuitionistische, 106
 - klassische, 106
- map, 30
- max, 20
- Metaprogrammierung, 14
- MINUS, 59
- modulo, 7
- motive*, *siehe* Zieltypfunktion
- Nachfolger, 53
- Name, 6, 67
- Nat, 64, 113
- natürliche Zahl, *siehe* Zahl,
 - natürliche
- Negation, 102
- neutrales Element, 115
- nil, 80
- Normalform, 52, 70
- NOT, 57
- not, 19
- null?, 19
- odd?, 19
- Ω , 135
- Operatorrangfolge, 8
- OR, 57
- or, 19
- Paar, 9, 67
- PAIR, 58
- Pair, 67
- pair?, 30
- parametrisierter Typ, *siehe* Typ,
 - parametrisierter
- Pattern Matching*, 110
- Peano-Axiome, *siehe*
 - Dedekind-Peano-Axiome
- Φ , 58
- Pi, 82
- PIE, 63
- PLUS, 56
- polnische Notation, 116
- Prädikat, 19
- Prädikatenlogik, 99
- PRED, 58
- Primzahltest, 34
- Priorität, 41, 112
- Projektion, *siehe* Komponente
- Prozedur, 6
- quote, 8, 65
- quotient, 7
- RACKET, 4
- rec-List, 81
- rec-Nat, 79
- Redex, 43
- Reduktion, 45
 - in Standardreihenfolge, 45
- Reduktionsschritt, 44, 47, 50
- reflexiv, *siehe* Relation, reflexive
- rein funktional, 24
- Rekursion, 26, 59, 75
- rekursive Definition, *siehe* Definition,
 - rekursive
- Relation
 - reflexive, 112
 - symmetrische, 95
 - transitive, 96
- REPL, 4
- replace, 97
- right, 100
- ROCQ, 109

- Rumpf, 14
- RUSSELL, BERTRAND, 63
- S-Ausdruck, 12
- same, 91
- Satz
 - von Church-Rosser, 52
 - von Euklid, 37
- SCHEME, 3
- second, 116
- Seiteneffekt, 24
- selbst-evaluierend, 6
- Semikolon, 25
- Sigma, 103
- S_n , 53
- sole, 102
- Spezialoperator, 8
- sub1, 20
- SUCC, 55
- Summenformel, gaußsche, *siehe*
 - gaußsche Summenformel
- Symbol, 4
- symbol?, 19
- symm, 95
- symmetrisch, *siehe* Relation,
 - symmetrische
- syntaktischer Zucker, *siehe* Zucker,
 - syntaktischer
- Syntaxbaum, 14
- T, 56
- #t, 18
- Tautologie, 100
- Term, *siehe* Ausdruck
- ternärer Operator, 21
- the, 65
- TIMES, 56
- TODO, 92, 106
- total, *siehe* Funktion, totale
- trans, 96
- transitiv, *siehe* Relation, transitive
- Trivial, 102
- Turingmaschine, 39
- Typ, 64
 - abhängiger, 84
 - parametrisierter, 83
- Typ-Konstruktor, 69
- Typentheorie, 63
- U, 69
- Universum, 69
- Urteil, 65, 69
- value*, *siehe* Wert
- Variable, 39, 71
 - freie, 48, 72, 105
 - gebundene, 48, 72, 105
- Variablenbindung, 16
- Variablenfang, 48
- Vec, 84
- vec::, 84
- vecnil, 84
- Verifikation, 63, 90, 109
- verkettete Liste, 11
- Verzweigung, 18
- vollständige Induktion, *siehe*
 - Induktion
- Wahrheitswert, 18, 56, 102
- Wert, 6
- which-Nat, 76
- write, 26
- Y, 59
- y combinator*, 59
- Z, 57
- Z3, 109
- Zahl
 - ganze, 4
 - natürliche, 31, 53, 64
- Zeiger, 11
- zero, 64
- zero?, 19
- Zucker, syntaktischer, 18, 64, 74, 110
- Zuweisung, 25