Prof. Dr. Friedrich Steimann

Modul 63618

Objekt-funktionale Programmierung

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Lektion 1: Voraussetzungen

1 Programmierung

1.1 Computer

Zwar ist die Idee der Programmierung unabhängig von Computern, aber sobald man über Programmierung in mehr als nur einem rein theoretischen Kontext nachdenkt, kommt man um die Gegebenheiten eines Computers nicht herum. Dies gilt selbst dann, wenn man die Programmierung in mathematische Modelle einbettet, denn wenn diese Modelle keine Entsprechung in der Hardware eines Computers haben, dann wird es schnell esoterisch. Und Programmierung ist nun mal eine hochgradig geerdete Disziplin, bei der es nicht zuletzt um die effiziente Ausführung von komplexen Programmen mit begrenzten Ressourcen geht.

Von-Neumann-Architektur

Wenn nun in diesem Lehrtext von einem **Computer** die Rede ist, meine ich eine Maschine mit einer **Von-Neumann-Architektur**, also mit einem **Prozessor** und einem **Speicher**, in dem **Programme** und **Daten** abgelegt sind. Auch wenn sich die Programmierung heute längst weit von dieser technischen Grundlage entfernt haben will, so scheint sie doch immer wieder durch und spätestens, wenn eine *Abstraktion*, die eine Programmiersprache mit sich bringt, unerwartete Ergebnisse liefert, ist es zum Verständnis notwendig, zu wissen, wie diese Abstraktion in die Gegebenheiten eines Computers übersetzt wird.

Und so sollten Sie nie vergessen: Letztlich wird alles auf Bitstrings im Speicher abgebildet und somit vom gleichen „Material“ (auch wenn Daten immateriell erscheinen, werden Sie doch durch ihre Speicherung physisch und somit, in Descartes‘ Sinn, zu einer *Res extensa*). Egal, welche Unterscheidungen wir noch treffen werden: Darin ist sich alles gleich! Die Kunst sicherer Programmiersprachen ist, diese Gleichheit aufzuheben, also verschiedene Dinge trotz gleicher Repräsentationen stets sicher zu unterscheiden.

1.2 Programme

Ein **Programm** überführt Eingaben in Ausgaben. Es kann als eine **Berechnungsvorschrift**, auch **Algorithmus** genannt, verstanden werden, die festlegt, wie die Ausgaben aus den Eingaben bestimmt werden. Dabei können die Eingaben eines Programms auch einem **Gedächtnis** entstammen; es handelt sich dann im Wesentlichen um Eingaben, die durch frühere Ausgaben (und damit auch durch frühere Eingaben) des Programms bestimmt und die im Gedächtnis gespeichert sind. Zudem können sich bei der Verwendung eines Programms Ein- und Ausgabe

abwechseln, so dass Eingaben von früheren Ausgaben informiert sein können. Das ist typischerweise bei interaktiven Systemen der Fall (und damit bei praktisch allen Systemen, die wir heute direkt benutzen). Für die theoretische Betrachtung der Programmierung ist jedoch die einfachste Auffassung von einem Programm, eben die obengenannte der Überführung von Eingaben in Ausgaben, in aller Regel ausreichend. Sie ist es insbesondere in dieser ersten Lektion.

1.3 Daten

Alle Daten, die ein Computer verarbeitet, werden durch endliche Folgen von Bits, auch **Bitstrings** genannt, repräsentiert. Dabei kann jedes Bit einen von zwei möglichen Werten annehmen, die wahlweise entweder als 1 oder 0 (Ziffern des *Dualsystems*) oder als wahr oder falsch (**Wahrheitswerte**) interpretiert werden. Eine mögliche Interpretation von Bitstrings (also Folgen von Bits) ist die als Zahlen, mit denen gerechnet werden kann (daher auch der Name „Computer“); eine andere ist die als Zeichenketten, mit denen beliebige Texte dargestellt und verändert werden können. Nicht zuletzt sind Bitstrings auch als Bilder interpretierbar; dies ist beispielsweise die Grundlage der Darstellung von Zeichen auf einem Bildschirm oder Drucker. Vor diesem Hintergrund kann der Speicher eines Computers als eine endliche, zeitlich veränderliche Abbildung von Speicheradressen, oder Namen, auf Bitstrings modelliert werden und vom Computer durchführbare Operationen als Abbildungen von Bitstrings auf Bitstrings. Es ist das Ziel von Programmiersprachen, von dieser Sichtweise auf einem für eine Anwendungsentwicklung (Programmierung) sinnvollen Niveau zu abstrahieren.

1.4 Programmiersprachen

Programmiersprachen sind formale Sprachen, in denen Programme, also Berechnungsvorschriften, so formuliert werden können, dass diese auf einem Computer ausführbar sind. Programmiersprachen haben dabei stets eine Brückenfunktion: Sie erlauben die Formulierung von Programmen in einer abstrakten, von der Hardware mehr oder weniger losgelösten Form. Dabei unterscheiden sich Programmiersprachen zum Teil erheblich darin, mit welchen Sprachkonstrukten sie von der Maschine abstrahieren. Diese Unterschiede sind zentral, da sie bestimmen, welche Mittel dem Programmierer zur Problemlösung zur Verfügung steht. Die Wahl der Programmiersprache sollte daher nicht zuletzt vom zu lösenden Problem abhängen.

Programmiersprachen sind exakte Sprachen, d. h., jedes in einer Programmiersprache formulierte Programm hat genau eine Bedeutung. Damit ist zum einen dem Computer keine Wahl gelassen, wie er ein Programm interpretiert, zum anderen kann der Programmierer nicht erwarten, dass der Computer sie schon verstehen wird — er ist vielmehr dazu gezwungen, das zu programmierende Problem bzw. seine Lösung vollständig und korrekt zu beschreiben. Umso mehr sollte er sich gewahr sein, dass der Programmiersprache eine wichtige Rolle zukommt: Die Sprache sollte es möglichst einfach machen, fehlerfreie Programme zu schreiben.

Ressourceneffizienz

Daneben gewinnt heute aber wieder ein anderes Kriterium für die Wahl einer Programmiersprache an Bedeutung: Wie ressourceneffizient ist die Ausführung darin geschriebener Programme? Sprachen, die hier glänzen, genügen oftmals weniger den ästhetischen Ansprüchen von Programmierern und fristen u. a. deswegen derzeit ein Schattendasein, aber die zunehmende Softwaresteuerung von allem und jedem verschiebt zwangsläufig die Priorität in Richtung Ressourceneffizienz, also dahin, wo sie ganz am Anfang der Programmierung schon einmal lag.

1.5 Definition von Programmiersprachen

Die Definition von Programmiersprachen umfasst zwei wesentliche Teile: die Definition ihrer Syntax und die Definition ihrer Semantik. Dabei regelt die Syntax die Konstruktion von Programmen als Sätze der Programmiersprache und die Semantik die Bedeutung dieser Sätze. Auch wenn die Semantik letztlich entscheidend ist, spielt die Syntax bei der Wahl einer Programmiersprache in der Praxis eine wichtige Rolle.

1.5.1 Warum ist Syntax so wichtig?

Texte sind linear

Sprache, egal ob gesprochen oder geschrieben, hat an ihrer Oberfläche eine lineare Form. Bei der gesprochenen Sprache äußert sich diese Form als Lautfolge, bei der geschriebenen als Zeichenfolge. Die in einer Sprache geäußerten Inhalte sind hingegen nicht unbedingt linear; dies äußert sich an der Sprachoberfläche in der Wiederholung von Namen sowie allgemein der Verwendung von Verweisen (deiktischen Ausdrücken), die selbst Teile der linearen Folge sind, dem Inhalt aber die Form eines gerichteten *Graphen* (also von Knoten, die über gerichtete Kanten verbunden sind) geben.

Programme sind Graphen

Nun sind Programme ebenfalls Graphen: So müssen Variablen in einem Programm mehrfach vorkommen, wenn sie einen Nutzen haben sollen, und *Unterprogramme*[1] werden in der Regel an mehreren Stellen aufgerufen. Sieht man einmal von solchen Verweisen ab, ist die Struktur von Programmen hierarchisch, hat also Baumform: Programme haben Unterprogramme, die wiederum Unterprogramme haben können; Kontrollstrukturen wie Verzweigungen und Wiederholungen können beliebig tief geschachtelt werden; etc. Diese hierarchische Struktur wird durch die Syntax der Programmiersprache ausgedrückt, die damit maßgeblich dafür ist, wie sich die Struktur in der linearen Form eines Programmtexts ausdrückt.

gute Editierbarkeit zählt

Die lineare Form von Programmen, die durch die Syntax der verwendeten Programmiersprachen bestimmt ist, ermöglicht es uns, Programme auf die gleiche Art zu schreiben wie andere Texte auch. Dies betrifft nicht nur die verwendeten Werkzeuge (es gibt Menschen, die Programme in Word schreiben!), sondern auch für unsere Schreibtechniken: Nicht zuletzt gehört das Verschieben und Kopieren von bereits Geschriebenem zum

[1] Ich verwende hier absichtlich den antiquiert wirkenden Begriff des Unterprogramms. Die Begriffe *Prozedur* und *Funktion* werden dann an jeweils passender Stelle eingeführt.

Schreiben wie das Setzen von Zeichen selbst. Gerade die Möglichkeiten, ein Programm ohne Rücksicht auf seine unterliegende Baumstruktur zu editieren, erklärt vermutlich auch, warum wir noch immer nicht mit Grapheditoren (den sog. projektionalen oder Strukturediteren; Googles Blockly ist ein gutes Beispiel hierfür) programmieren, obwohl uns diese eigentlich viel Arbeit und Zeit ersparen sollten.

Sozialisation mit der Syntax

Wir schreiben also Programme als linearen Text, obwohl ihre Struktur hierarchisch und ihr Inhalt ein Graph ist. Damit wir (und ein Compiler!) die Hierarchie in der linearen Darstellung erkennen, verwendet die Syntax von Programmiersprachen neben Satzeichen (wie Trennzeichen und Klammern) auch Schlüsselwörter, die zur besseren Orientierung durch einen Editor mit Sprachkenntnissen auch noch besonders hervorgehoben werden. Einrückungen im in Zeilen umgebrochenen Programmtext heben die hierarchische Struktur auch optisch hervor, indem sie der linearen Folge eine zweite Dimension auferlegen. Weil aber all diese Hinweise auf die Struktur von Programmen in der linearen Oberfläche für unser Programmverstehen so zentral sind, entwickeln wir nicht nur Vorlieben für die Syntax einer Sprache, sondern sozialisieren uns auch mit ihr. Eine fremde Syntax, obwohl erlernbar, macht uns das Leben da nur unnötig (wie wir meinen) schwer.

Letztlich sollten Sie sich aber von der Syntax lösen, denn was eine Sprache auszeichnet sind immer deren Konstrukte. Und so verwendet diese Lehrveranstaltung gleich mehrere Sprachen, wobei jede Sprache mindestens ein Prinzip der objekt-funktionalen Programmierung durch eigene Sprachkonstrukte besonders eindrücklich präsentiert. Klammern Sie sich also nicht an eine Ihnen bekannte (vertraute) Syntax, sondern entwickeln Sie einen Blick für die dahinterliegenden Konstrukte!

1.5.2 Semantik

Mit der Semantik wird allen syntaktisch wohlgeformten Programmen einer Sprache eine Bedeutung beigemessen. Man unterscheidet dabei zwischen statischer und dynamischer Semantik:

- Mit der **statischen Semantik** verbunden sind Regeln, nach denen u. a. die Querverweise in einem Programm (also beispielsweise die Verwendung von Namen wie die von Variablen, die an anderer Stelle im Programm vereinbart worden sind) aufgelöst werden. Gelingt diese sog. **semantische Analyse** nicht, wird ein Programm von einem *Compiler* mit einem *Übersetzungsfehler* zurückgewiesen oder von einem *Interpreter* mit einem *Laufzeitfehler* abgebrochen.
- Die **dynamische Semantik** bestimmt, was bei der Ausführung eines (ausweislich der semantischen Analyse wohlgeformten) Programms passiert. Für die meisten gebräuchlichen Programmiersprachen ist sie nur durch ihren *Compiler* oder *Interpreter* wirklich festgelegt, selbst wenn sich Referenzhandbücher redlich um eine vollständige Beschreibung der Semantik bemühen. Lediglich im akademischen Bereich findet man formale Beschreibungen der Semantik von meistens eher kleinen Sprachen; Abschnitt 3.4 wird Beispiele hierfür zeigen.

1.6 Programmierparadigmen

Es gehört zu den großen Entdeckungen der Informatik, dass es auf der einen Seite grundverschiedene **Arten der Programmierung** gibt (so verschieden, dass die eine mit der anderen kaum etwas gemeinsam zu haben scheint), auf der anderen Seite aber mit allen diesen Arten exakt die gleichen Probleme gelöst werden können (wenn auch auf höchst unterschiedliche Art und Weise). Diese grundverschiedenen Arten der Programmierung bezeichnet man auch als **Programmierparadigmen**. Da mit *Programmiersprachen* in der Regel eine Art der Programmierung vorgegeben wird, werden sie Paradigmen zugeordnet.

imperative und deklarative Programmierung

Bei den Programmierparadigmen unterscheidet man zunächst zwischen der **imperativen** und der **deklarativen Programmierung**. Auf hoher Flugebene geht es bei der imperativen Programmierung darum, zu beschreiben, wie ein Problem gelöst wird (wobei das Problem selbst nur durch seine Lösung beschrieben wird), und bei der deklarativen Programmierung darum, zu beschreiben, was das Problem ist (wobei die Lösung aus der Problembeschreibung automatisch abgeleitet wird). Auf den ersten Blick scheint die deklarative Programmierung attraktiver, aber tatsächlich ist es häufig einfacher, eine Lösung genau zu beschreiben als ein Problem so, dass seine Lösung eindeutig bestimmt ist. Zudem muss man bei der deklarativen Programmierung wissen, welcher Mechanismen sie sich bedient, um das Problem so aufzuschreiben („zu deklarieren"), dass es auf Basis dieser Beschreibung auch automatisch gelöst werden kann. Die deklarative Programmierung ist also keineswegs generell die bessere; gleichwohl ist ihr eine gewisse Erhabenheit schwer abzustreiten.

funktionale und logische Programmierung

Die deklarative Programmierung wird häufig weiter in die *funktionale* und die *logische Programmierung* unterteilt. Erstere ist für diesen Lehrtext zentral; letztere ist eher Anwendungen der (klassischen) KI zugeordnet und spielt hier keine Rolle. Allerdings wird die logische Programmierung auch manchmal mit der *relationalen Programmierung* gleichgesetzt (wobei mathematisch gesehen Relationen und Funktionen eng verwandt sind, so dass man die relationale Programmierung auch mit der funktionalen in Verbindung bringen kann). Ein Beispiel für eine relationale Sprache, die viele von Ihnen vermutlich kennen, ist *SQL*: In ihr können Anfragen an eine relationale Datenbank deklarativ formuliert werden. Allerdings werden von SQL auch Datenbank-Updates unterstützt, die von Haus aus einen imperativen Charakter haben (`insert`, `delete` etc.). Wie man schon an diesem Beispiel sieht, können Sprachen auch mehreren Paradigmen zugeordnet werden.

objektorientierte Programmierung

Die *objektorientierte Programmierung* wird üblicherweise der imperativen Programmierung zugerechnet, kann jedoch auch als eigenständiges Paradigma angesehen werden. Wie der Titel dieses Lehrtextes nahelegt, wird sie häufig mit der funktionalen Programmierung kombiniert, weswegen ich auf eine Darstellung der objektorientierten Programmierung als eigenständiges Paradigma verzichte (sie ist eher eine Wanderin zwischen den Welten).

2 Imperative Programmierung

Achtung: Den Inhalt dieses Kapitels setze ich als bekannt voraus. Es ist deswegen nur schwach mit Beispielen illustriert. Dass ich es trotzdem hier anführe, soll lediglich der Einordnung des Inhalts in einen für die nachfolgenden Kapitel nützlichen Rahmen dienen.

Wie der Name nahelegt, besteht ein imperatives Programm aus einer Folge von **Befehlen**, auch **Anweisungen** genannt (commands oder statements im Englischen). Die Befehle oder Anweisungen eines Programms werden in einer durch das Programm selbst, u. a. durch die Verwendung sog. *Kontrollflussanweisungen*, festgelegten Reihenfolge abgearbeitet. Ein Programm ist beendet, wenn die letzte Anweisung der Folge abgearbeitet wurde. Man spricht dann von **Terminierung**, wobei zu beachten ist, dass Programme nicht terminieren müssen. Zudem können Programme vorzeitig beendet werden, entweder weil bei der Ausführung einer Anweisung ein Fehler aufgetreten ist oder weil die Ausführung von außen abgebrochen wird.

2.1 Die Elemente imperativer Programmierung

2.1.1 Kontrollflussanweisungen

Neben der Sequenz (Ausführung von Anweisungen in der Reihenfolge, in der sie nacheinander im Programmtext stehen) bestimmen Verzweigungen, Wiederholungen („Schleifen“) und Unterprogrammaufrufe den Programmablauf. Zusammen machen sie die sog. **Kontrollstrukturen** aus. Während die **Sequenz** durch den linearen Programmtext implizit gegeben ist, werden die anderen Kontrollstrukturen durch entsprechende Kontrollflussanweisungen umgesetzt. Klassische Beispiele für Kontrollflussanweisungen, die Sie in fast allen imperativen Programmiersprachen finden, sind if-then-else, while-do, repeat-until sowie der Unterprogrammaufruf.

WIKIPEDIA

strukturierte Programmierung

Verzweigungen und **Wiederholungen** können durch bedingte Sprünge zu bestimmten Stellen eines Programms realisiert werden; die Sprunganweisung selbst ist jedoch seit mehr als einem halben Jahrhundert verpönt und wird nur noch selten verwendet. Stattdessen werden in der sog. **strukturierten Programmierung** Verzweigungen und Wiederholungen zu Blöcken zusammengefasst, die, als Ganzes betrachtet, Elemente einer Sequenz sind, deren unmittelbar zuvor und danach ausgeführte Anweisung im Programmtext also auch unmittelbar davor bzw. danach stehen, so dass der Kontrollfluss insgesamt besser mit der Linearität des Programmtextes korrespondiert. Unterprogramme werden ebenfalls als Blöcke aufgefasst; sie sind jedoch nicht in die Sequenz der Anweisungen eingebettet. An ihrer Stelle stehen in einer Sequenz **Unterprogrammaufrufe**, die den Programmfluss zum Unterprogramm verzweigen und danach wieder an die Stelle des Aufrufs zurückkehren lassen. Durch solche Aufrufe kann ein Unterprogramm, anders als eine Verzweigung oder Wiederholung, an mehreren Stellen eines Programms auftauchen; es wird an diesen Stellen durch seinen Aufruf (eine Art von *Verweis*) vertreten. Man kann sich einen Unterprogrammaufruf auch wie eine textuelle Ersetzung des Aufrufs (Verweises) durch den Text des Unterprogramms vorstellen; allerdings ist dem eine wichtige Grenze gesetzt.

WIKIPEDIA

rekursiver Aufruf

Die Tatsache, dass ein Unterprogrammaufruf an seiner Stelle das aufgerufene Unterprogramm nur vertritt, ermöglicht ein in seiner Mächtigkeit nicht zu unterschätzendes Konstrukt: den Aufruf eines Unterprogramms durch sich selbst. Man nennt einen solchen Aufruf **rekursiv**, und zwar **direkt rekursiv**, wenn der Aufruf im Text desselben Unterprogramms erfolgt, und **indirekt rekursiv**, wenn er in einem anderen, durch es aufgerufenem Unterprogramm (und damit immer noch im Rahmen seiner eigenen Ausführung) erfolgt. Eine Ersetzung des Aufrufs durch den Text des Unterprogramms ist nun nicht mehr (im direkten Fall gar nicht und im indirekten nur noch bis zu einem gewissen Grad) möglich, da dieser Text ja den Aufruf enthält, man also in einen unendlichen Ersetzungsprozess geriete.

Rekursion vs. Wiederholung

Wie man leicht einsieht, ist der rekursive Unterprogrammaufruf mit der Wiederholung verwandt. Der wesentliche Unterschied zur Wiederholung in Form einer Schleife ist, dass die Rekursion an jeder beliebigen Stelle, ja sogar an mehreren Stellen des zu wiederholenden Anweisungsblocks stattfinden kann, wodurch eine Schachtelungsstruktur entsteht, die bei einer Schleife nicht möglich ist. Der rekursive Aufruf ist somit allgemeiner als die Schleife. Tatsächlich simuliert eine sog. **Endrekursion**, also der Aufruf eines Unterprogramms als dessen letzte Anweisung, eine Schleife (vgl. dazu Abschnitt 13.8.2); das Umgekehrte, nämlich dass eine Schleife eine Endrekursion ersetzen kann, ist die Basis eines wichtigen Optimierungsverfahrens bei der Übersetzung von Programmen in Maschinencode.

Parallelität und Nebenläufigkeit

Neben der sequentiellen Ausführung von Programmen (Ausführung einer Anweisung nach der anderen, ggf. mit Verzweigung, Wiederholung und Unterprogrammaufruf) gibt es auch eine **parallele** und eine **nebenläufige Ausführung**. Bei diesen werden, wie ihre Namen nahelegen, Programmteile gleichzeitig bzw. ineinander verzahnt ausgeführt. Dies kann in bestimmten Kontexten günstig sein, kommt aber um den Preis dessen, dass man aus dem Programmtext nicht mehr ablesen kann, in welcher Reihenfolge die Anweisungen eines Programms ausgeführt werden. Das ist immer dann bedeutsam, wenn Anweisungen Effekte haben, von denen andere Anweisungen oder deren Effekte abhängen. Das Ergebnis eines Programms ist dann ohne weitere Maßnahmen nicht mehr genau vorhersehbar, was in vielen Kontexten als Problem angesehen werden muss. In solchen Kontexten ist diese Art der Programmierung schwierig und mit Vorsicht zu genießen, nicht zuletzt, weil sich aus Parallelität oder Nebenläufigkeit resultierende Programmierfehler nur schwer reproduzieren, eingrenzen und damit auch beseitigen lassen.

2.1.2 Variablen und Wertzuweisung

Neben den Kontrollflussanweisungen, die den Programmablauf steuern und die in der einen oder anderen Form auch in anderen Programmierparadigmen vorkommen, ist für die imperative Programmierung die **Wertzuweisung** das zentrale Element. Das Ziel der Zuweisung eines Werts ist stets eine **Variable**; die Variable hat nach der Zuweisung (in der Folge der Programmausführung) solange den ihr zugewiesenen Wert, bis ihr ein neuer Wert zugewiesen wird.

Variablen als Speicher, Zuweisung als Kopie

Während in der Mathematik (wie auch in der funktionalen Programmierung; s. Kapitel 3) eine Variable als **Platzhalter** für einen Wert steht, durch den, sobald er bekannt ist, die Variable ersetzt werden kann (die sog. **Substitution**; s. Kapitel 3), ist in der imperativen Programmierung eine Variable ein **Speicher** für einen Wert.[2] Tatsächlich kann man sich in der imperativen Programmierung Variablen[3] als Namen für Speicherzellen vorstellen, in denen Daten im Zuge ihrer Verarbeitung temporär abgelegt werden. Entsprechend werden dann auch bei einer Zuweisung von einer Variable an eine andere die bezeichneten Daten von einer Speicherstelle in eine andere kopiert.

die problematische Seite der imperativen Programmierung

Die Funktion von Variablen als Speicher für Daten oder Werte ermöglicht die mehrmalige Wertzuweisung an eine Variable innerhalb desselben Programmabschnitts. Tatsächlich ist diese Möglichkeit das Charakteristikum der imperativen Programmierung schlechthin: Das Wesen der meisten imperativen Programme besteht schlicht darin, den Inhalt von Variablen solange zu verändern, bis das gewünschte Ergebnis erreicht wurde. Ein geradezu ikonisches Beispiel hierfür ist die Zuweisung

```
1 x := x + 1
```

die den Wert der Variable `x` um `1` erhöht. Unter keiner deklarativen Interpretation kann dieser Ausdruck wahr sein: Selbst wenn man `:=` als „wird definiert als“ liest, erstreckt sich der Geltungsbereich der Definition doch auch auf die rechte Seite und es gibt schlicht keinen Wert, den man für `x` einsetzen könnte, ohne dass die Definition widersprüchlich wäre.

Genau dies ist nun der Ansatzpunkt für die größte Kritik an der imperativen Programmierung: Aufgrund von mehrfachen Wertzuweisungen kann eine Variable an verschiedenen Stellen eines Programmtextes, im Rahmen von Wiederholungen sogar an derselben Stelle, für verschiedene Werte stehen, wodurch Programme schwerer zu verstehen sind. Ein Indiz dafür, dass dies wirklich so ist, ist die Bedeutung, die das Anzeigen von Variablenwerten beim Debuggen von Programmen hat — wer ertappte sich nicht immer wieder beim Inspizieren von Variablenwerten zur Laufzeit, um einem Programmierfehler auf die Schliche zu kommen? Man kann sich leicht vorstellen, wie sich die Probleme weiter verschärfen, wenn die Reihenfolge der Zuweisungen an eine Variable im Kontext der parallelen oder nebenläufigen Programmierung nicht vorhersagbar ist. Vom Überschreiben von Variablenwerten (durch mehrfache Zuweisungen) wird daher in solchen Kontexten immer wieder abgeraten. Zugleich ist die mangelnde Überschreibbarkeit von Variablenwerten in „puren“ funktionalen Sprachen eines der am häufigsten angeführten Argumente für ihre Überlegenheit.

---

[2] Der Unterschied mag zunächst nicht besonders groß und zudem wie ein rein technischer erscheinen (s. dazu auch Abschnitt 3.4.3), er ist aber, wie wir gleich sehen werden, eine Voraussetzung für die Abgrenzung der imperativen von der deklarativen Programmierung.

[3] Ich verwende in diesem Text das Wort „Variable“ nicht deadjektivisch, also als substantiviertes Adjektiv (wie etwa „Veränderliche“). Deswegen schreibe ich „zwei Variablen“ und nicht „zwei Variable“ sowie „der Wert einer Variable“ und nicht „der Wert einer Variablen“.

2.1.3 Ausdrücke

Während auf der linken Seite einer Wertzuweisung immer eine Variable steht, können auf der rechten Seite, der Seite, die den zugewiesenen Wert bestimmt, beliebige **Ausdrücke** stehen. Ausdrücke sind:

1. Literale,
2. Variablen,
3. Funktionsaufrufe,
4. durch Operationen oder Relationen verknüpfte (Teil)-Ausdrücke sowie
5. in manchen Sprachen auch Wertzuweisungen.

Ein **Literal** ist die textuelle Repräsentation eines Werts in der Syntax der Programmiersprache, also beispielsweise `10` als Repräsentant der Zahl Zehn oder `"abc"` als Repräsentant der entsprechenden Zeichenkette. Ein **Funktionsaufruf** entspricht im Wesentlichen einem *Unterprogrammaufruf* mit der Besonderheit, dass der Aufruf selbst für einen Wert steht, nämlich den Wert, der von dem aufgerufenen Unterprogramm als Ergebnis berechnet und zurückgegeben wird. Bei einem Funktionsaufruf werden dem Unterprogramm in der Regel Werte, die sog. **tatsächlichen Parameter** (engl. actual parameters[4]), mitgegeben, die an der Aufrufstelle in Form von Teilausdrücken des Funktionsaufrufs angegeben werden und die im Unterprogramm in Form der sog. **formalen Parameter** (engl. formal parameters), also lokaler Variablen, vorliegen. **Operationen** entsprechen im Wesentlichen mit der Sprache fest vorgegebenen Funktionen wie Addition, Multiplikation oder die Konkatenation von Zeichenketten, für deren Namen häufig mathematische Symbole wie `+`, `*` etc. verwendet werden und die nicht nur vor, sondern auch zwischen und hinter ihren Operanden stehen können (wie beispielsweise in `x + y`). **Relationen** kann man sich als spezielle Funktionen vorstellen, die stets einen Wahrheitswert (`true` oder `false`) zum Wert haben: Beispiele aus so gut wie allen Programmiersprachen sind `=` (oder `==`) für Gleichheit und `>` für die Größer-Relation.

Schachtelung von Ausdrücken

Ausdrücke können beliebig tief geschachtelt (aus Teilausdrücken zusammengesetzt) sein und somit eine erhebliche Komplexität annehmen. In der imperativen Programmierung gelten komplexe Ausdrücke jedoch häufig als schlechter Stil; als besser gilt dann, wenn Teilausdrücke herausgezogen und durch Variablen ersetzt werden, denen zuvor die Werte der Teilausdrücke zugewiesen wurden, so dass der Gesamtausdruck dann nur noch die (Werte dieser) Variablen verbindet. Ein optimierender Compiler wird erkennen,

[4] Der im Englischen gebrauchte Term „actual parameter“ wird auch manchmal mit „*aktueller Parameter*“ ins Deutsche übersetzt, was gelegentlich als falsch verhöhnt wird, aber die Sache trotzdem trifft, zumindest, wenn man die zeitliche Dimension, die mit der Programmausführung einhergeht, in Betracht zieht: Es gibt nämlich in der Regel viele tatsächliche Parameter zu einem Methodenaufruf, von denen — zu jedem Zeitpunkt — höchstens einer der aktuelle ist. Wir werden später (in Abschnitt 3.3.1) die Unterscheidung von formalem und tatsächlichem Parameter zugunsten der (gleichwertigen) Unterscheidung von Parameter und Argument aufgeben.

dass die Variablen nur diesem einen Zweck dienen, und sie bei der Übersetzung in Maschinencode wieder eliminieren.

Boolesche Ausdrücke

Eine bestimmte Form von Ausdrücken, die sog. **Booleschen Ausdrücke**, mit den Werten `true` und `false` sind zentraler Bestandteil von *Kontrollflussanweisungen*, wo sie in den Bedingungen von Verzweigungen und Wiederholungen vorkommen. So ist beispielsweise der Vergleich `x > 10` ein Boolescher Ausdruck, der die Bedingung für eine Verzweigung oder das Abbruchkriterium für eine Wiederholung darstellen kann. Neben Relation wie `>` oder `=` kommen in Booleschen Ausdrücken auch logische **Junktoren** wie `not` oder `and` vor.

beförderbare Ausdrücke

In manchen Sprachen können bestimmte Ausdrücke die Rolle von Anweisungen einnehmen. Klassische Beispiele hierfür sind Funktionsaufrufe sowie Wertzuweisungsausdrücke (falls vorgesehen); sind sie nicht Teil eines umfassenden Ausdrucks (stehen also allein da), dann gelten sie als Anweisungen (wobei bei Funktionsaufrufen der Wert der Funktion verworfen wird). Je nach Sprache kann dies zusätzliche Syntax erfordern, wie beispielsweise ein hintangestelltes Semikolon in Sprachen mit C-artiger Syntax: Dort wird aus dem Zuweisungsausdruck `i = 1` die Anweisung

```
2 i = 1;
```

Man spricht dann von der **Beförderung eines Ausdrucks zu einer Anweisung**.

Ausdrücke sind universell

Praktisch alle imperativen Programmiersprachen enthalten Ausdrücke als Teilsprache. Dies ist insofern interessant, als Ausdrücke zugleich die Grundlage funktionaler Sprachen sind — in manchen dieser Sprachen sind sie sogar die einzige syntaktische Kategorie. Trotzdem sind die imperativen Sprachen nicht allgemeiner als die funktionalen Sprachen, umfassen diese also insbesondere nicht; mehr dazu in Kapitel 3.

2.1.4 Strukturierte Daten

Die Sequenz, die Wiederholung sowie Ausdrücke über primitive Werte (wie ganze Zahlen, Wahrheitswerte oder Zeichen) und die Wertzuweisung an Variablen reichen aus, um jede Berechnung durchzuführen, die überhaupt durchgeführt werden kann (die Klasse der berechenbaren Probleme; als Beweis mag hier die Mächtigkeit der While-Sprache dienen). Allerdings sind die Programme, die nur aus diesen einfachen Bausteinen bestehen, nicht nur lang und komplex, sondern verlangen auch umständliche Codierungen der Daten, die sie verarbeiten sollen. Sie sind somit schwer zu schreiben und zu lesen. Beides wirkt sich aber gleichermaßen negativ auf die Programmierung aus.

Programme werden oft deutlich einfacher, wenn man die Daten, die sie verarbeiten, in eine für die Verarbeitung besonders geeignete Form bringt. Dabei umfassen Daten neben den Ein- und Ausgaben auch alle Zwischenergebnisse, die in der Programmierung nicht selten wesentlich komplexer als die Ein- und Ausgaben sind. Diese Erkenntnis spiegelt sich im Begriffspaar „Algorithmen und Datenstrukturen“ wider, das in der praktischen Informatik so gut wie unauf-

lösbar ist. So bieten denn auch praktisch alle Programmiersprachen die Möglichkeit, in Programmen strukturierte Daten zu erzeugen, die einer gegebenen Problemstellung besser gerecht werden als ihre unstrukturierten Komponenten. Dabei sind der Strukturierung von Daten ebenso wenig Grenzen gesetzt wie der Steuerung des Programmablaufs durch die Verwendung von *Kontrollflussanweisungen*.

Für die Erzeugung strukturierter Daten braucht man zwei Dinge:

1. gegebene, **primitive Daten** wie beispielsweise Zahlen, die Wahrheitswerte `true` und `false` oder Zeichen wie `a`, `b`, etc. sowie
2. **Konstruktoren**, die aus einfacheren Daten zusammengesetzte, strukturierte Daten erzeugen.

Konstruktoren

Ein klassischer Konstruktor ist das **Array**, das eine theoretisch unbegrenzte Anzahl von (häufig gleichförmigen) Daten zu einer Struktur zusammenfasst, in der jedes einzelne Datum, oder jedes **Element** des Arrays, über einen Index (meistens eine Ganzzahl, in jedem Fall aber durch einen Ausdruck gegeben) identifizierbar ist. Eine gängige Schreibweise für den Zugriff auf ein Element eines Arrays ist `a[i]`, wobei `a` für einen Ausdruck steht, der zu einem Array auswertet (z. B. eine Variable, die ein Array enthält) und `i` für einen Ausdruck, der zu einem Index auswertet (z. B. ebenfalls eine Variable). Ein anderer klassischer Konstruktor ist der **Record**, der mehrere individuell benannte Datenelemente (die meistens verschiedener Natur sind), seine **Felder**, zusammenfasst, wobei die einzelnen Felder dann über ihre Namen identifizierbar sind (die Namen sind im Gegensatz zu den Indizes eines Arrays in den meisten Programmiersprachen selbst keine Werte und können somit auch nicht durch Ausdrücke angegeben werden, sondern müssen, wie Variablennamen auch, direkt, oder *literal*, im Programm stehen). Eine übliche Schreibweise für den Zugriff auf ein Feld eines Records ist `r.f`, wobei `r` ein Ausdruck ist, der zu einem Record auswertet, und `f` der Name des Feldes.

Komponenten als Variablen

Strukturierte Daten werden wie primitive Daten in Variablen gespeichert. Sofern die Komponenten eines strukturierten Datums selbst veränderlich sind, kann man diese ebenfalls als Variablen auffassen. Ein Array mit n Elementen entspricht dann n Variablen, die allerdings nicht individuell benannt, sondern indiziert sind. Ein Record mit n Feldern entspricht ebenfalls n Variablen, die jedoch durch die Namen der Felder explizit benannt sind.

Werte sind unveränderlich

Wenn man bei einem Wert in Form einer zusammengesetzten Datenstruktur eine Komponente durch eine andere ersetzt, dann erhält man dadurch einen anderen Wert. Dies liegt in der Natur von Werten an sich begründet: Sie können nicht geändert, sondern nur ersetzt werden. Dies mag im Einzelfall gegen die Intuition sein: Wenn man beispielsweise einer Person eine Adresse zuordnet und bei dieser Adresse die Hausnummer durch eine andere ersetzt, dann hat man scheinbar die Adresse geändert. Tatsächlich hat man aber die alte Adresse durch eine neue ersetzt (wobei sich alte und neue nur in der Hausnummer unterscheiden). Noch unintuitiver wird es, wenn die Adresse selbst eine Komponente

eines eine Person repräsentierenden Werts ist: Dann wird mit der Hausnummer die ganze Person durch eine andere ersetzt. Dies ergibt sich aus der Zeit- oder Zustandslosigkeit von Werten: Aus einer Zwei kann keine Drei werden, selbst wenn man dafür nur ein Bit in der binären Repräsentation der beiden Zahlen kippen muss. Diese Sichtweise auf Daten wird durch die Objektorientierung (in Lektion 2) aufgehoben.

Strukturtiefe und Zugriffspfad

Wie das obige Beispiel von Person und Adresse zeigt, können die Komponenten strukturierter Daten selbst strukturiert sein. Die Schachtelungstiefe einer solchen *Datenstruktur*[5] ist grundsätzlich beliebig. Mit der Tiefe der Schachtelung steigt auch die Anzahl der Selektoren (`[i]` bei Arrays und `.f` bei Records), die man hintereinander setzten muss, um auf einen elementare Komponente zuzugreifen. Man spricht bei solchen Ketten von Selektoren auch von einem **Zugriffspfad**.

Teil-Ganzes-Beziehung

Daten, die über Konstruktoren wie die eines Arrays oder Records gebildet werden, sind Ganze, deren Teile ihre Komponenten sind. So ist ein Array ein Ganzes, das seine Elemente als Teile enthält; gleiches gilt für einen Record und seine Felder. Insbesondere hängt die Existenz der Teile (im Speicher) an der Existenz des Ganzen — entfernt man das Ganze aus dem Speicher, sind auch seine Teile weg. Solche Teil-Ganzes-Beziehungen auf Datenebene spiegeln Kompositionen (Zusammensetzungen) der Realität, die durch die Daten repräsentiert wird, auf natürliche Weise wider. Interessanterweise kennt die Objektorientierung solche Kompositionen eher nicht: Hier überleben die Teile regelmäßig das Ganze.

rekursive Daten

Zwar können Daten geschachtelt werden (ein Array von Records beispielsweise), aber Daten können sich nicht selbst enthalten. Der Grund hierfür ist ein ganz praktischer: Wenn sie sich selbst enthielten, wäre die Struktur unendlich groß (so wie der Blick in zwei gegenüberliegende Spiegel unendlich tief ist) und passte somit nicht in den endlichen Speicher eines Computers.

dynamische Daten

Eine mildere Form dieses Problems stellen dynamische Datenstrukturen dar, also strukturierte Daten, die im Laufe eines Programms unvorhersehbar wachsen und auch wieder schrumpfen können. Klassische Beispiele hierfür sind Bäume, also strukturierte Daten, die aus Knoten besteht, die entweder terminal sind (die sog. Blätter) oder eine Anzahl von Kindern halten, die selbst wieder Knoten sind.[6] Solche Datenstrukturen sind dynamisch, weil man sie durch das Ersetzen von Blättern durch Eltern jederzeit erweitern und durch den umgekehrten Vorgang auch wieder schrumpfen lassen kann, wobei die Größe nur durch den verfügbaren

[5] Wenn ich hier (in dieser Lektion) von „Datenstruktur“ spreche, dann meine ich „strukturierten Daten“. Tatsächlich wird der Begriff aber häufig mit dem Begriff des *Datentyps* (Kapitel 26 in Lektion 4) gleichgesetzt. Datentypen, oder *Typen* allgemein, dienen u. a. der Organisation des Speichers, indem sie die Größe (den Speicherplatzbedarf) der Daten, die von ihrem Typ sind, festlegen. Wir beschäftigen uns hier aber noch nicht mit Typen, selbst wenn die meisten imperativen Programmiersprachen typisiert sind, ihre Variablen also fest mit einem Typ verbunden sind, der die Werte bestimmt, die sie aufnehmen können (für die sie stehen).

[6] Setzt man hier „Datenstruktur“ mit „Datentyp“ gleich, dann hat man es mit einem rekursiven Datentyp zu tun. Solche behandeln wir dann ebenfalls in Lektion 4.

Speicher begrenzt ist. Voraussetzung hierfür ist jedoch, dass für die Anlage einer solchen Datenstruktur nicht von Anfang an der maximal benötigte Speicherplatz reserviert werden muss, denn der ist in der Regel vorab nicht bekannt.

2.1.5 Verweise oder Zeiger

Die Lösung der Realisierungsprobleme von rekursiven und dynamischen Datenstrukturen liegt in der Verwendung von **Verweisen**, besser unter dem Namen **Zeiger** (engl. pointer) bekannt.[7] Wie der Name nahelegt, zeigt ein Verweis, oder verweist ein Zeiger, lediglich auf ein Datum und ist es nicht selbst. Dies ist in etwa vergleichbar mit dem Aufruf eines *Unterprogramms*, der nicht das Unterprogramm selbst ist, der es aber in seinem Kontext vollständig ersetzt (vgl. Abschnitt 2.1.1). Mittels eines Verweises kann ein Datum auf sich selbst verweisen, ohne dass es sich deswegen selbst enthalten müsste (wiederum ganz wie bei einem rekursiven Unterprogrammaufruf).

Dereferenzierung und Indirektion

Ein Zeiger verweist jedoch nicht nur auf ein Datum, sondern ist auch selbst eines: Ein Zeiger ist ein Wert. Zeiger können daher, genau wie andere Werte auch, Variablen zugewiesen werden. Der eigentlich interessierende Wert, der durch die Variable vertreten werden soll, ist jedoch meistens das Datum, auf das der der Variable zugewiesene Zeiger zeigt. Um auf diesen Wert zuzugreifen, wird der Zeiger (der Wert der Variable) **dereferenziert**. Da man bei einer Dereferenzierung erst auf den Zeiger und erst darüber dann auf das gezeigte Datum zugreift, nennt man den Zugriff über Zeiger **indirekt**. Diese Form der **Indirektion** ist sehr mächtig: Wie man leicht einsieht, ändert sich nämlich mit dem Zeiger auch dasjenige Datum, auf das indirekt zugegriffen wird. Der Zeiger steuert also gewissermaßen den Zugriff. Dies ist in etwa vergleichbar mit dem Zugriff auf die Elemente eines Arrays über einen Index (und tatsächlich sind Zeiger in Programmiersprachen wie C Indizes in den dem Programm insgesamt zur Verfügung stehenden Speicher).

Herkunft von Zeigern

Wie aber kommt es zu Zeigern? In der imperativen Programmierung wird ein Zeiger durch die Umkehrung der Dereferenzierung, den sog. **Adress-Operator**, gewonnen. Dieser Operator wird auf Variablen angewendet (inkl. Komponentenvariablen; s. Abschnitt 2.1.4) und liefert einen Zeiger auf den Inhalt der Variable (genauer: die Speicherstelle, an der der Inhalt steht) zurück. Die Dereferenzierung des so gewonnenen Zeigers liefert dann wieder den Inhalt der Variable.

rekursive und dynamische Datenstrukturen

Mithilfe von Zeigern lassen sich nun rekursive und dynamische Datenstrukturen aufbauen. Dazu wird der Zeiger meistens im Feld eines Records gespeichert, der neben dem Zeiger auch noch „Nutzlast“ enthält. So ist der Knoten eines Baums typischerweise ein Record mit einem Feld für die Nutzlast des Knotens und einem Feld mit einem Zeiger auf den Elternknoten oder einem Array von Zeigern auf die Kindknoten (ggf. auch beides).

[7] Ebenfalls gebräuchlich ist der Begriff der *Referenz*; je nach Kontext kann eine Referenz aber auch mehr sein als ein Pointer.

keine Teile, keine Ganzen

Durch die Verwendung von Zeigern wird nicht nur das Problem der Selbstbezüglichkeit von Daten und deren Strukturen gelöst — zugleich wird damit auch die Teil-Ganzes-Beziehung aufgehoben. Es ist nämlich nur noch der Zeiger (Bestand)teil einer Datenstruktur, also in ihr enthalten — das Datum, auf das er verweist, ist es nicht. Dadurch können die Elemente einer dynamischen Datenstruktur wie der eines Baumes alle dieselbe, endliche Größe haben: Ein Kindknoten ist nicht mehr in seinem Elternknoten enthalten, vielmehr sind beide eigenständige Knoten, die sich über Verweise gegenseitig referenzieren können. Dies ist ein sehr mächtiges Konstrukt, das allerdings auch seinen Preis hat.

Der Null-Zeiger

Gerade um dynamische Datenstrukturen zu realisieren, muss eine Variable (ein Feld), die (das) einen Zeiger enthält, auch leer bleiben können. Eine leere Zeigervariable kann dann natürlich nicht dereferenziert werden, weil sie auf nichts zeigt. Bei der Dereferenzierung einer Zeigervariable, die theoretisch leer sein kann, wäre also stets sicherzustellen, dass sie nicht leer ist.

Nun haben Variablen in der imperativen Programmierung immer einen Wert und so wurde mit der Einführung von Zeigern in die imperative Programmiersprache Algol auch der Null-Zeiger eingeführt, mit dem Zeigervariablen automatisch initialisiert werden. Hat eine Zeigervariable zwischenzeitlich einen anderen Wert erhalten und soll wieder „leer“ werden, dann wird ihr der Null-Zeiger zugewiesen. Zwar ist die Zeigervariable damit nicht wirklich leer, aber eine Dereferenzierung wäre in den meisten Fällen dennoch die Manifestation eines Programmierfehlers. So kommt es, dass in den meisten Programmiersprachen die Dereferenzierung eines Null-Zeigers genau den gleichen Status hat wie eine Division durch 0 — sie endet in einem Programmabbruch.[8] Tony Hoare, der für sich reklamiert, den Null-Zeiger eingeführt zu haben, hat dies mittlerweile seinen **Billion-dollar mistake** genannt. Leider ist er dabei schuldig geblieben, wie er den Fehler hätte vermeiden können (denn sonst wäre es kein Fehler gewesen); eine mögliche Lösung, die er vielleicht im Sinn hatte, werden wir in Abschnitt 26.3 kennenlernen.

WIKIPEDIA

2.1.6 Call-by-value und Call-by-reference

Sowohl Wertzuweisungen als auch Unterprogrammaufrufe sind für die imperative Programmierung zentral. Aus der Verwendung beider ergibt sich jedoch ein Problem, wenn man Werte, die in Unterprogrammen berechnet und dort vielleicht in Variablen gespeichert werden, in den aufrufenden Hauptprogrammen verwenden möchte. Die einfachste Lösung, hierfür *globale Variablen* zu verwenden, ist aus softwaretechnischen Gründen verpönt; sie steht zudem für eine

[8] Tatsächlich ist das bei Divisionen durch 0 gar nicht mehr unbedingt der Fall, denn viele Programmiersprachen haben einen besonderen Zahlwert `NaN` (für not a number) eingeführt, der das Ergebnis einer solchen Division ist und der in gewisser Weise der Bedeutung des Null-Zeigers von „kein Zeiger“ entspricht. Jedoch wird dieser Wert durch nachfolgende Berechnungen hindurch propagiert, wodurch sein Ursprung verschleiert wird. Genauso könnte die Dereferenzierung des Null-Zeigers zu einem Null-Wert führen, jedoch mit dem gleichen Problem: Das ist nur dann eine gute Idee, wenn dadurch kein nachfolgender Programmfehler provoziert wird.

Asymmetrie zwischen den Eingaben und Ausgaben eines Unterprogramms, sofern für die Eingaben *formale Parameter* des Unterprogramms, also für das Unterprogramm *lokale Variablen*, herangezogen werden.

Bei der Parameterübergabe von einem Hauptprogramm an ein Unterprogramm wird in der Regel der tatsächliche Parameter, ein Wert in den entsprechenden formalen Parameter, eine Variable, kopiert. Diese Art der Übergabe wird **Call-by-value** genannt. Ändert man den Wert dieser Variable durch eine Zuweisung, dann ändert dies nichts am tatsächlichen Parameter — es wird ja nur eine Kopie überschrieben. Dies lässt sich allerdings ändern, wenn an der Stelle des tatsächlichen Parameters ein Verweis auf eine Variable angegeben wird: Dann wird dieser Verweis in den aktuellen Parameter kopiert und kann im Unterprogram dereferenziert werden, womit dann auch ein Schreiben in die Variable, die für den aktuellen Parameter steht, möglich ist. Somit kann das Unterprogramm den Wert einer Variable des es aufrufenden Hauptprogramms ändern, ohne diese Variable direkt zu verwenden. Diese Art der Parameterübergabe wird **Call-by-reference** genannt.

2.1.7 Aliasing und Sharing

Wenn mehrere Variablen auf denselben Speicherbereich verweisen, spricht man von **Aliasing**. Die Intuition hinter dieser Namensgebung ist, dass Variablen **Namen** für Werte sind: Tatsächlich haben ja die Variablen eines Programms die Form von Namen, womit sie die Werte, für die sie stehen (ihren Inhalt) benennen. Wenn also *derselbe* Wert durch mehrere Variablen benannt wird, hat der Wert auch mehrere Namen; man spricht dann von Aliasen.

Man beachte eine kleine Spitzfindigkeit im vorangegangenen Absatz: Wenn dort von *demselben Wert* die Rede ist, dann ist damit nicht nur der gleiche Wert gemeint — zwei Variablen können nämlich den gleichen Wert haben, ohne dass deswegen Aliasing vorliegen muss (nämlich wenn die gleichen Werte in verschiedenen, durch die Variablen jeweils bezeichneten Speicherbereichen liegen). Gemeint ist hier vielmehr, dass sich die Variablen denselben Speicherbereich teilen, was bedeutet, dass ihre Werte nicht unabhängig, sondern vollständig „synchronisiert“ sind. Sprich: Wenn man den Wert der einen Variablen ändert, ändert sich zugleich der Wert der anderen Variable. So ist beim *Call-by-reference* der formale Parameter ein Alias; aber auch bei der Zuweisung einer Variable, die eine Zeiger enthält, an eine andere entsteht ein Alias. Aliasing ist deswegen vor allem in der Objektorientierung ein Thema; es liegt aber auch vor, wenn den Variablen vom Compiler derselbe Speicherbereich zugeordnet wird (wie es beispielsweise bei *varianten Records* der Fall ist; s. Abschnitt 26.3).

Aliasing als Problem

Während Aliasing ein zentrales Konzept insbesondere der objektorientierten Programmierung ist, ist es nicht ausschließlich positiv konnotiert: *Versehentliches Aliasing*, also die gemeinsame Belegung desselben Speichers durch zwei Variablen, die nichts miteinander zu tun haben, kann die Quelle schwer aufzudeckender Programmierfehler sein. Außerdem erschwert es die Programmanalyse und damit letztlich auch die *Verifikation von Programmen* (Zusicherung bestimmter erwünschter Eigenschaften mit formalen Mitteln) erheblich.

Sharing

Beim Aliasing handelt es sich um eine spezielle Form des Teilens von Daten durch mehrere Teile eines Programms, auch **Sharing** genannt. Anders als bei globalen Variablen, die auch für das Teilen von Daten verwendet werden, haben Aliase typischerweise die Form von lokalen Variablen. Das Sharing mit Aliasen ist damit weniger offensichtlich und entsprechend tückischer. Dies gilt insbesondere dann, wenn der mit den Aliasen verbundene Speicherbereich wieder freigegeben wird, ohne dass gleichzeitig (oder vorher) alle Aliase abgeräumt werden.

2.2 Speicherverwaltung

Wie bereits in Abschnitt 2.1.2 angedeutet abstrahieren in der imperativen Programmierung Variablen vom Speicher eines Computers. Eine Variablenname kann somit als ein Etikett für einen Speicherbereich angesehen werden, der den Wert der Variable aufnimmt. Allerdings ist es im Allgemeinen nicht so, dass ein Variablenname des Programmtexts immer auf die gleiche Stelle im Speicher verweist.

2.2.1 Stack und Heap

Bei allen Parallelen zwischen rekursiven Unterprogrammaufrufen und rekursiven Datenstrukturen gibt es einen entscheidenden Unterschied: der Ort im Speicher, an dem Daten abgelegt werden. Dazu muss man etwas weiter ausholen.

globale und lokale Variablen; Scopes

In praktisch allen aktuellen imperativen Programmiersprachen seit Algol 60 werden Variablen nicht **global**, sondern **lokal** zu dem sie einführenden und verwendenden *Unterprogramm* angelegt. Konkret bedeutet dies, dass die Variablen auch nur innerhalb des Unterprogramms verwendet werden können. „Innerhalb“ bezieht sich dabei auf den Programmtext, den sog. **lexikalischen** oder **statischen Scope** einer Variable, nicht auf den Programmablauf (das wäre dann der *dynamische Scope*). Man beachte, dass der Text eines Unterprogramms je nach Programmiersprache auch weitere Unterprogramme enthalten kann; Variablen der umgebenden Unterprogramme sind dann auch in den enthaltenen Unterprogrammen verwendbar, also für diese quasi global. Auch das ergibt sich aus dem Programmtext und nicht dem Programmablauf.

Stack und Stack frames

Interessant wird es, wenn ein Unterprogramm sich selbst aufruft. Hieran wird nämlich deutlich, dass der statische Scope eine starke dynamische Komponente hat: Obwohl der Programmtext derselbe ist und die Variablen somit gleich heißen, haben diese Variablen für jeden Aufruf jeweils ihre eigenen Werte. Insbesondere überschreiben Zuweisungen an Variablen in einem rekursiven Aufruf nicht die Werte der gleichen (gleichnamigen) Variablen im Kontext der Aufrufstelle, deren Werte somit nach der Rückkehr vom rekursiven Aufruf fortbestehen. Möglich wird dies dadurch, dass die lokalen Variablen eines Unterprogramms auf einem **Stack** angelegt werden, wobei bei jedem Unterprogrammaufruf ein neuer sog. **Stack frame** auf den Stack gelegt wird, der alle lokalen Variablen enthält (und der bei Abschluss des Unterprogramms wieder abgeräumt wird). Damit das in allen Situationen funktioniert, ist es notwendig, dass der Speicherbedarf der Werte der Variablen schon

beim Aufruf des Unterprogramms bekannt ist. Bei rekursiven und dynamischen Datenstrukturen (Abschnitt 2.1.4), deren Wachsen und Schrumpfen in der Regel nicht im Gleichschritt mit Unterprogrammaufrufen erfolgt, ist dies jedoch nicht der Fall.

Heap

Für die Speicherung von rekursiven und dynamischen Datenstrukturen ist ein vom Stack losgelöster Speicher erforderlich. Dieser ist durch den sog. **Heap** gegeben, der, ähnlich wie der Stack, dynamisch organisiert, aber davon unabhängig ist. Daten, die im Heap gespeichert werden, sind ausschließlich über *Zeiger* zugreifbar; da Zeiger selbst eine feste Größe haben, können sie auf dem Stack oder im Heap (dort in der Regel als Komponenten von rekursiven oder dynamischen Datenstrukturen) gespeichert sein. Zeiger auf dem Stack sind Werte lokaler Variablen; sie dienen typischerweise als Einstiege in rekursive oder dynamische (verzeigerte) Datenstrukturen.

Unterschiede der Speicherbelegung und -freigabe

Stack und Heap sind also zwei voneinander getrennte Speicherbereiche, die im Laufe eines Programms mit den Werten von Variablen belegt werden, wobei der hierfür benötige Speicherplatz auch wieder freigegeben werden kann. Beide unterscheiden sich allerdings grundlegend darin, wie die Speicherbelegung und -freigabe erfolgt: Beim Stack erfolgt dies im Gleichschritt mit dem Aufruf und der Beendigung von Unterprogrammen, beim Heap hingegen davon losgelöst. Dies bedeutet insbesondere, dass beim Stack die Speicherbereinigung automatisch (implizit durch den Programmablauf gesteuert) erfolgt, beim Heap hingegen nicht. Hier ist die Speicherfreigabe ein von der -belegung unabhängiger, expliziter Prozess, was eine erhebliche Herausforderung darstellt. Je nach Programmiersprache geschieht die Freigabe mittels einer sog. automatischen **Speicherbereinigung** (engl. *garbage collection*) oder muss von der Programmiererin ins Programm geschrieben werden. Ist das eine oder das andere fehlerhaft, entstehen sog. **Speicherlecks** (belegter Speicher wird nicht mehr freigegeben, was zu unnötigem Speichermangel bis hin zu einem dadurch bedingten Programmabbruch führen kann).

2.2.2 Zustand

Die aktuelle Belegung sämtlicher Variablen eines laufenden Programms, einschließlich derer auf Stack und Heap, zusammen mit der Stelle, an der sich die Programmausführung gerade befindet (inkl. *Call stack*), machen seinen **Zustand** aus. Wertzuweisungen ändern somit den Zustand des Programms; er stellt das Gedächtnis eines Programms dar. Tatsächlich kann ein Programm unterbrochen werden, indem man seinen Zustand speichert und diesen zu einem späteren Zeitpunkt wiederherstellt, um das Programm fortzusetzen. Das passiert u. a., wenn man einen Computer in einen Ruhezustand („Hibernate“) versetzt oder wenn auf einem Gerät mit beschränkten Ressourcen (wie einem Mobilgerät) zu einem anderen Programm gewechselt wird.

2.3 Das Wesen imperativer Programmierung

Programme als Zustandsautomaten

Ein imperatives Programm ändert die Werte seiner Variablen so lange, bis darin die gewünschten Ausgaben enthalten sind. Es tut dies mithilfe von

Anweisungen oder Befehlen, nämlich mit seinen Kontrollflussanweisungen und Zuweisungen, die über Variablen, genauer die Änderung ihrer Werte, in Wechselwirkung zueinander stehen. Ein Programm ist damit ein **Zustandsautomat** (engl. state machine), dessen Zustände durch seine Variablenbelegungen und dessen Zustandsübergänge durch seine Anweisungen bestimmt werden. Da der Speicher jedes Computers, der Programme ausführt, endlich ist, ist es sogar ein **endlicher Zustandsautomat** (engl. finite state machine, FSM). Dass eine Programmiersprache rekursive Unterprogrammaufrufe erlaubt, ändert daran nur theoretisch etwas: Tatsächlich entspricht der dafür verwendete Stack auch nur einer Variable, deren Inhalt durch die Größe des Speichers beschränkt ist.

Neben- oder Seiteneffekte

Nun werden Zustandsautomaten mit der Zahl ihrer Zustände (repräsentiert durch die Variablen eines Programms) schnell unübersichtlich und so bieten so gut wie alle imperativen Programmiersprachen die Möglichkeit, den Gesamtzustand eines Programms in viele Teilzustände aufzuteilen. Dazu gehört insbesondere die Einführung von *Unterprogrammen* mit *lokalen Variablen*; diese können jeweils als eigenständige Automaten aufgefasst werden. Allerdings erlauben imperative Programmiersprachen typischerweise, in Unterprogrammen auch die Werte von Variablen, die im Verhältnis zu ihnen global sind (also einschließlich lokaler Variablen umfassender Unterprogramme), zu verändern. Diese sog. **Neben-** oder **Seiteneffekte** (engl. side effects)[9] sind typisch für die imperative Programmierung und werden häufig kritisiert, da sie der Idee der Aufteilung eines globalen Automaten in mehrere, unabhängig zu betrachtende Teilautomaten entgegenstehen. In einfachen imperativen Programmiersprachen sind Nebeneffekte jedoch die einzige Möglichkeit, zu bewirken, dass sich ein Unterprogramm zwischen zwei Aufrufen etwas merkt, sein Verhalten also von vorherigen Aufrufen abhängt. Dies ändert sich spätestens mit der Einführung der objektorientierten Programmierung, bei der bestimmte Unterprogramme, die sog. *Methoden*, Objekten zugeordnet sind, deren Zustand sie ändern können und von deren Zustand ihre Ausführung abhängen kann.

die zentrale Rolle von Variablen

Wenn man sich eines unbedingt merken sollte, dann das, dass Variablen in der imperativen Programmierung eine andere Funktion haben als in der deklarativen und insbesondere der funktionalen Programmierung (und im Übrigen auch in der Mathematik): An ihnen — und nicht an der Existenz von Anweisungen oder Befehlen — manifestiert sich die Idee des Zustands eines Programms und dass der Fortschritt eines Programms hin zur Lösung eines Problems (Ausgabe) aus Zustandswechseln besteht. Diese charakteristische Rolle, die den Variablen der imperativen Programmierung zukommt, wird durch die oftmals gleiche Einführung von Variablen in anderen Programmierparadigmen (und zwar sowohl in Lehrtexten als auch in Programmiersprachen in Gestalt von syntaktisch gleich daherkommenden Variablendeklarationen) kaschiert.

[9] Umgangssprachlich entspricht „side effect“ dem Wort „Nebenwirkung“, aber dieser Begriff ist in der Informatik kein technischer.

3 Funktionale Programmierung

Wie bereits in Kapitel 1 festgestellt überführt ein Programm Eingaben in Ausgaben. Mathematisch gesehen ist ein Programm also eine *Abbildung*, und zwar von einem *Definitionsbereich*, der die Menge möglicher Eingaben umfasst, in einen *Wertebereich*, die Menge der möglichen Ausgaben. Dabei können Ein- und Ausgaben zusammengesetzt sein, also jeweils mehrere Einzelwerte umfassen; diese Zusammensetzung spiegelt sich dann auch im Definitions- und Wertebereich wider. Sofern die Ausgaben eines Programms eindeutig durch seine Eingaben bestimmt sind, also eine Eingabe zu höchstens einer Ausgabe führt, ist die Abbildung — und somit das Programm, das die *Berechnungsvorschrift* dafür angibt — eine **Funktion**.[10]

In der funktionalen Programmierung sind nicht nur alle Programme Funktionen, sie werden auch als solche dargestellt. Das führt zu der Frage, wie Funktionen überhaupt dargestellt werden.

3.1 Funktionen in der Mathematik

In der Mathematik sind zwei Darstellungen von Funktionen gebräuchlich: die Darstellung als **Funktionsgleichung** und die Darstellung als eine Menge von geordneten Paaren, auch als **Graph** bezeichnet. Beide Darstellungen sind in gewissen Grenzen gegeneinander austauschbar: So bezeichnen die Funktionsgleichung $x + y = 1$ und der Graph (die Paarmenge) $\{ (x, y) \mid x + y = 1 \}$ dieselbe Funktion. Die Paare sind geordnet, weil man bei Funktionen zwischen einer **unabhängigen** und einer **abhängigen Variable** unterscheidet: Die unabhängige Variable, häufig x genannt, nimmt die erste Stelle des Paares ein und die abhängige, häufig y, die zweite. Die unabhängige Variable steht für die *Eingabe* in eine Funktion, die abhängige für die dazugehörige *Ausgabe*. In der Mathematik nennt man die Eingaben in eine Funktion ihre **Argumente** und die Ausgaben ihre **(Funktions-)Werte**.

implizite Funktionsgleichungen

Bei der im obigen Beispiel verwendeten Funktionsgleichung $x + y = 1$ ist nicht ohne weiteres klar, welches die abhängige und welches die unabhängige Variable ist — dies ergibt sich erst aus der genannten Namenskonvention. Man nennt solche Funktionsgleichungen auch **implizit**; sie geben nämlich nicht explizit an, wie man den (abhängigen) Funktionswert zu einem (unabhängigen) Argumentwert berechnet. Stattdessen würde man üblicherweise, und algebraisch äquivalent, $y = 1 - x$ schreiben: Hier ergibt sich unmittelbar, wie man einen Funktionswert y zu einem Argumentwert x berechnet.

Funktionen als Zuordnungsvorschriften

Um klarzustellen, dass x das Argument (die unabhängige Variable) der Funktion ist und nicht y, schreibt man auch $f(x) = 1 - x$; die Lesart „f von

[10] Im mathematischen Diskurs sind die Begriffe Abbildung und Funktion im Wesentlichen gleichbedeutend. Ich bezeichne hier jedoch jede gerichtete Relation als Abbildung, also auch solche, die ein Element des Definitionsbereichs auf mehrere Elemente des Wertebereichs abbilden.

x ist gleich ...“ legt dabei nahe, dass es sich bei $f(x) = 1 - x$ um eine Vorschrift, die **Zuordnungsvorschrift** für die Berechnung des Werts einer Funktion f in Abhängigkeit von x, handelt.[11] Anstelle von $1 - x$ auf der rechten Seite der Funktionsgleichung können natürlich beliebige (arithmetische) Ausdrücke über die Argumentvariable x stehen, ohne dass dies etwas am Wesen von f als dem einer Funktion ändern würde.

Funktionsanwendung per Variablensubstitution

In einer Zuordnungsvorschrift wie $f(x) = 1 - x$ steht die Variable x für beliebige Elemente des Definitionsbereichs der Funktion. Als Gleichung gilt $f(x) = 1 - x$ also für jeden Wert aus dem Definitionsbereich. Wenn man beispielsweise für x den Wert 1 einsetzt, ergibt sich zunächst rein syntaktisch $f(1) = 1 - 1$ und weiter, durch Ausrechnen der rechten Seite, $f(1) = 0$; das Paar $(1,0)$ ist somit gemäß Funktionsgleichung Element der Funktion (oder ein Punkt des durch die Funktionsgleichung bestimmten Graphen). Die Ersetzung der Variable x in der Funktionsgleichung der Funktion f durch einen Wert nennt man auch die **Anwendung der Funktion** f auf den Wert; den Vorgang der Ersetzung nennt man auch **(Variablen-)Substitution**.

Sprechweise bei Substitution

Der Sprachgebrauch bezüglich der Substitution verdient eine Klärung. Im Deutschen sagt man in der Regel, dass man eine Variable durch einen Wert subsituiert (oder allgemein einen Ausdruck durch einen Ausdruck). Dabei kann man „substituiert“ getrost durch „ersetzt“ ersetzen; die Präposition „durch“ macht jeweils klar, was an die Stelle des anderen tritt. Im Englischen sagt man im mathematischen Kontext hingegen stets „substitute a value for a variable“; „substitute a variable with a value” ist ungebräuchlich, auch wenn die Reihenfolge der beiden Objekte von „substitute“ hier (zumindest Deutschsprachigen) intuitiver erscheinen mag.

Grundlage der Substitution

Man kann also per Substitution von x in der Funktionsgleichung durch einen Wert ausdrücken, für welches Argument man den Funktionswert von $f(x)$ berechnen möchte: So steht $f(1)$ für den Wert von $f(x)$ für $x = 1$. Dass dies funktioniert leitet sich aus der Erkenntnis ab, dass man in der Mathematik in Ausdrücken Gleiches durch Gleiches ersetzen kann, ohne dass sich am Inhalt des Ausdrucks etwas ändert. Wenn also $x = 1$ gilt, dann kann man in der Funktionsgleichung von f jedes Vorkommen von x durch 1 ersetzen, ohne dass dies etwas am Gehalt des Ausdrucks „$f(x)$ für $x = 1$“ ändern würde. Dieses für die Mathematik fundamentale Prinzip wird auch als **Identitätsprinzip** oder **Leibniz-Gesetz** bezeichnet[12]; es liefert die Grundlage der Substitution. Wenn es Ihnen wie eine Trivialität vorkommt, dann sollten Sie bedenken, dass hier eine semantische Gleichheit (nämlich $x = 1$; dabei sind x und 1 offensichtlich syntaktisch verschieden) eine syntaktische Operation begründet (die textuelle Ersetzung), ohne dass dies wiederum etwas an der Semantik des so manipulierten

Link

[11] Die gleiche Zuordnungsvorschrift wird durch den Ausdruck $x \mapsto 1 - x$ (gelesen als „x wird auf $1 - x$ abgebildet“ gelesen) gegeben; hierbei wird sie jedoch nicht mit dem Namen „f“ versehen, weswegen man bei dem Ausdruck auch von einer *anonymen Funktion* spricht. Mehr dazu in Abschnitt 3.3.1.

[12] In seiner Umkehrung definiert dieses Prinzip Gleichheit: Wenn man von zwei Ausdrücken den einen durch den anderen in jedem Kontext ersetzen kann, ohne dass dies etwas am Wert des Gesamtausdrucks ändert, sind die Ausdrücke gleich.

Ausdrucks ändern würde. Dabei funktioniert die syntaktische Ersetzung ganz ohne Verstand (und insbesondere auch ohne das, was wir gemeinhin unter Rechnen verstehen).

Benennung von Funktionen

Der Buchstabe „f“ in $f(x)$ scheint zwar generisch, ist aber tatsächlich der Name der Funktion (genauso, wie „x“ und „y“ die Namen von Variablen sind). Die Benennung von Funktionen ist insbesondere dann von Vorteil, wenn man verschiedene Funktionen von derselben Variable x im selben Kontext darstellen und verwenden möchte: So definieren $f(x) = 1 - x$ und $g(x) = x$ zwei verschiedene Funktionen f und g. Zudem macht die Benennung von Funktionen deutlich, dass es sich hierbei um eigenständige mathematische Objekte handelt, auf die man — durch Nennung ihres Namens — Bezug nehmen kann. So tragen denn auch viele wichtige Funktionen standardisierte Namen wie beispielsweise „sin“ oder „max“.

partielle und totale Funktionen

Eine Funktion, die nicht alle ihre Argumente (Eingaben) auf Werte (Ausgaben) abbildet, nennt man **partiell**, alle anderen **total**. Die Funktion $f(x) = 1/x$ ist ein klassisches Beispiel für eine partielle Funktion: $f(0)$ ist undefiniert. Da die Behandlung partieller Funktionen häufig Fallunterscheidungen verlangt und damit verzweigt, versucht man, diese Funktionen durch entsprechende Einschränkung ihres Definitionsbereichs zu totalen zu machen, im Beispiel von $1/x$ dadurch, dass man 0 aus ihrem Definitionsbereich ausschließt. Die Deklaration ihres Definitions- und Wertebereichs gehören daher zur Definition einer Funktion dazu. Man schreibt üblicherweise für die Deklaration einer Funktion $f\colon A \longrightarrow B$ und meint damit, dass f alle Werte aus A auf Werte aus B abbildet. Für den Graph von f, $\{(x, y) \mid y = f(x)\}$, gilt damit, dass er eine Teilmenge von $A \times B$ ist (wobei $A \times B$ das *kartesische Produkt* der Mengen A und B bezeichnet).

Relationen

Eine partielle Funktion bildet also jedes Element aus ihrem Definitionsbereich auf höchstens ein und eine totale auf genau ein Element des Wertebereichs ab. Abbildungen aller einzelnen Elemente des Definitionsbereichs auf eine beliebige Anzahl Elemente des Wertbereichs nennt man **Relationen**[13]; da „eine beliebige Anzahl“ auch „kein“ und „ein“ umfasst, sind Funktionen, und zwar sowohl partielle als auch totale, immer auch Relationen.[14] Ein Beispiel für eine Relation ist durch den Graph $\{(x, y) \mid x^2 + y^2 = 1\}$ gegeben; er bildet beispielsweise $x = 0$ auf $y = 1$ und $y = -1$ ab. Man beachte, dass die durch die Funktionsgleichung $f(x) = \{\sqrt{x}, -\sqrt{x}\}$ dargestellte Abbildung von der vorgenannten Relation verschieden ist, da sie einen Wert auf eine Menge von Werten (und nicht mehrere Werte) abbildet. Kategorientheoretisch betrachtet sind aber Relationen und Funktionen, die auf Mengen abbilden, isomorph.

[13] s. dazu Fußnote 10

[14] Man vergleiche dazu die Auffassung von Relationen als speziellen Funktionen in Abschnitt 2.1.3 — beide sind gültig (und gebräuchlich).

3.2 Funktionen in der Programmierung

Wie in der Mathematik gibt es auch in der Programmierung zwei Darstellungsweisen von Funktionen: die als Graphen und die als Funktionsgleichungen. Allerdings sind beide Formen deutlich eingeschränkt: Graphen sind immer endlich (eine endliche Menge von Paaren) und werden zumeist in Form von Tabellen oder Mengen von Kanten (Zeigern), die einen Ausgangsknoten mit einem Zielknoten verbinden, dargestellt, und Funktionsgleichungen haben immer die Form von *Berechnungsvorschriften*, sind also insbesondere nicht implizit. Letzteres deckt sich durchaus mit der imperativen Programmierung (Abschnitt 2): Nicht nur ein Programm, sondern auch alle seine Unterprogramme lassen sich als Berechnungsvorschriften auffassen, die Eingaben eindeutig auf Ausgaben abbilden und somit das Wesen von Funktionen haben. Die bloße Verfügbarkeit von Funktionen reicht also nicht aus, um die funktionale Programmierung von der imperativen abzugrenzen.

partiell ist die Regel

Im Gegensatz zur Mathematik sind in der Informatik Funktionen im allgemeinen *partiell*: Es ist damit zu rechnen, dass ein Algorithmus (ein Programm) für bestimmte Eingaben keine Ausgaben liefert, sei es, weil ungültige Eingaben zwar akzeptiert, aber ignoriert werden (um ein undefiniertes Ergebnis zu vermeiden), sei es, weil sich der nächste Schritt nicht durchführen lässt (Abbruch mit einem Fehler) oder das Programm in eine Endlosschleife gerät, also nicht terminiert. Anstatt jedoch die Menge der Eingaben so zu beschränken, dass unerwünschte Ausgaben (keine oder Abbruch mit einem Fehler) nicht auftreten können (die Nichtterminierung lässt sich nicht grundsätzlich ausschließen; das *Halteproblem*), ist es häufig einfacher, den Zielbereich um einen zusätzlichen Wert, häufig „undefiniert“ genannt und als $\perp$ notiert, zu erweitern und diesen Wert den Argumenten, für die kein nächster Schritt in der Berechnung des Funktionswerts möglich ist, ersatzweise zuzuordnen. Aus der Programmierung kennen Sie diesen speziellen Wert vielleicht als `undefined`; durch seine Verwendung lassen sich partielle Funktionen, die sicher terminieren, die aber für manche Eingaben keine Ausgaben liefern, zu *totalen* erweitern. Um Fallunterscheidungen insbesondere bei der *Komposition* (Verkettung) von Funktionen zu vermeiden, müssen jedoch nicht nur die Wertebereiche, sondern auch die Definitionsbereiche von Funktionen um den zusätzlichen Wert erweitert werden.[15]

WIKIPEDIA

keine Relationen in der funktionalen Programmierung

Funktionen können eine Eingabe nicht auf mehrere Ausgaben abbilden: Sie geben entweder einen oder keinen Wert zurück. In der imperativen Programmierung kann man davon abweichen, indem man einer Funktion mehre zusätzliche formale Parameter spendiert, die über Call-by-value mehrer Ausgaben aufnehmen, in der reinen funktionalen Programmierung jedoch nicht. Will man hier mehrere Ausgaben, so muss man sie in einem Container (wie im obigen Beispiel von der Menge $\{\sqrt{x}, -\sqrt{x}\}$) zusammenfassen. Die Ausgabe eines leeren Containers, die dann als „kein Ergebnis“ interpretiert werden müsste, kann dann den Fall „undefiniert“ ersetzen. Liefert ein Programm bei verschiedenen Programmläufen auf dieselben Eingaben (inkl. desselben Inhalts eines eventuellen

[15] Man vergleiche dazu auch die Diskussion der Dereferenzierung von `null` sowie die Einführung von `NaN` in Abschnitt 2.1.5; tatsächlich lässt sich das Dereferenzieren eines Zeigers wie eine Abbildung des Werts der Zeigervariable auf den Zielwert des Zeigers interpretieren, so dass nach hiesiger Logik die Dereferenzierung von `null` `undefined` oder wiederum `null` ergeben müsste.

Gedächtnisses) verschiedene Ausgaben (z. B. weil es einen Zufallsgenerator verwendet oder *nebenläufig* ist; s. Abschnitt 2.1.1), dann ist es nicht funktional im engeren Sinne.

3.3 Der Lambda-Kalkül

Der Lambda-Kalkül wurde vom Mathematiker und Logiker Alonzo Church und seinen Mitarbeitern in den 1930er Jahren entwickelt. Er entstand im Kontext von Anstrengungen der damaligen Zeit, die Mathematik in der Mengentheorie zu begründen, und stellt eine grundlegende Alternative dazu dar, die ohne den Mengenbegriff auskommt. Die auf Mengen basierenden Überlegungen waren nämlich zunächst durch die Entdeckung von *Russells Antinomie* gescheitert; Russel konnte dann aber mit seiner *Typentheorie* (aus der u. a. die Stufen der Prädikatenlogik hervorgegangen sind; Abschnitt 25.1) selbst einen ersten Lösungsansatz liefern, der sich später, in abgewandelter Form, auch im *einfach typisierten Lambda-Kalkül* (Abschnitt 25.2) wiederfindet. Dieser liefert heute die Grundlage der Typsysteme aller funktionalen Programmiersprachen, wenn nicht aller Programmiersprachen überhaupt.

minimal und universal

Der Lambda-Kalkül hat also nicht Mengen und ihre Verhältnisse (die Element- und die Teilmengenbeziehung) zum Gegenstand, sondern ausschließlich Funktionen. Dabei kommt er mit nur drei Sprachelementen aus: (1) Variablen, (2) Abstraktionen oder Funktionen und (3) deren Anwendungen. Die Universalität des Lambda-Kalküls basiert darüber hinaus auf einer ganze Reihe von mathematischen Konzepten, derer sich auch funktionale Programmiersprachen bedienen, wie z. B. *Funktionen höherer Ordnung* und *Currying*: Es ist deswegen überaus sinnvoll, sich im Rahmen dieses Lehrtextes mit dem Lambda-Kalkül zu befassen, selbst wenn er Ihnen einiges an Zeit zum Wundern, Zurücklehnen und Nachdenken abverlangt.

Typen als Erweiterung einer Sprache

Wir beginnen in dieser Lektion mit dem puren Lambda-Kalkül. In Lektion 4 betrachten wir seine Erweiterung zunächst um einfache Typen (der *einfach typisierte Lambda-Kalkül*) und dann um Typparameter (*System F*). Beide Erweiterungen betreffen lediglich die Syntax und die statische Semantik des Lambda-Kalküls; sie ändern nichts an der dynamischen Semantik und sind so gesehen minimale Erweiterungen des Lambda-Kalküls, die gleichwohl ein weites Spektrum dessen abdecken, was heutige Programmiersprachen ausmacht. Idealerweise ermöglicht diese Vorgehensweise auch denjenigen unter Ihnen, die bereits objektorientiert oder objekt-funktional programmieren, ein tieferes Verständnis dafür, was Typen bedeuten und wie sie funktionieren.

3.3.1 Funktionen als Abstraktionen

Wenn wir eine Reihe von ähnlichen Berechnungen wie beispielsweise $1 + 1$, $2 + 1$ und $5 + 1$ vorliegen haben, dann können wir von ihnen abstrahieren, indem wir den variablen Teil, hier den jeweils ersten Summanden der drei Ausdrücke, durch eine Variable x ersetzen, hier also $x + 1$ schreiben. Der Ausdruck $x + 1$ steht dabei nicht nur als Abstraktion der drei angeführten Additionen, sondern jeder Addition von 1 zu einem (durch x repräsentierten) Wert. x ist hier die Variable der Abstraktion; sie wird deswegen auch manchmal **Abstraktionsvariable** genannt.

Man beachte, dass hier Variablen so verwendet werden, wie wir es aus der Mathematik kennen: Sie sind Platzhalter für Werte und keine Speicher wie in der imperativen Programmierung (vgl. Abschnitt 2.1.2)!

Auszeichnung der Abstraktionsvariable; Lambda-Ausdruck; Bindung

Um darzustellen, dass bei einem Ausdruck wie $x + 1$ eine Abstraktion mit Abstraktionsvariable x vorliegt, muss x noch irgendwie ausgezeichnet werden. Die Auszeichnung ist notwendig, weil in einem Ausdruck auch andere Variablen oder Konstanten vorkommen können, die nicht Gegenstand der Abstraktion sind (oder über die in einem anderen Kontext abstrahiert wird). So könnte im Ausdruck $x + c$ sowohl x als auch c Abstraktionsvariable sein (dass üblicherweise „x“ als Name der Abstraktionsvariable verwendet wird ist lediglich Konvention). Übliche Schreibweisen für die Auszeichnung der Abstraktionsvariable x im Ausdruck $x + 1$ sind $x \mapsto x + 1$ (gelesen als „x wird abgebildet auf $x + 1$) und $\lambda x.\, x + 1$. Ersteres ist Ihnen vielleicht aus der einen oder anderen Programmiersprache bekannt, letzteres ist die (namensgebende) Standardschreibweise des Lambda-Kalküls. Beide Schreibweisen sind gleichwertige Abstraktionsausdrücke; man nennt sie auch **Lambda-Ausdrücke**. Die Auszeichnung von x nennt man übrigens **Bindung16** und $\mapsto$ bzw. λ **Bindungsoperatoren**; entsprechend wird x als die **gebundene Variable** bezeichnet.

Geltungsbereich einer Variablenbindung

Ein Bindungsoperator bindet eine Abstraktionsvariable an ihren **Geltungsbereich** (engl. scope), der im Lambda-Kalkül dem sog. **Rumpf** der Abstraktion entspricht. Im Beispiel von $\lambda x.\, x + 1$ bindet also λx die Variable x an den Ausdruck $x + 1$; der Punkt zwischen der Bindung und ihrem Geltungsbereich, dem Rumpf der Abstraktion, fungiert hier als Trennzeichen. Der Geltungsbereich muss gelegentlich explizit ausgezeichnet werden; dafür werden dann Klammern verwendet, die den gesamten Abstraktionsausdruck einschließen, also beispielsweise $(\lambda x.\, x + 1)$. Sind keine Klammern angegeben, erstreckt sich der Geltungsbereich einer Abstraktion bis zum Ende des Ausdrucks, in dem sie enthalten ist (bis zum rechten Ende, wenn sie in keinem Ausdruck enthalten ist).

anonyme Funktionen

Lambda-Ausdrücke wie $\lambda x.\, x + 1$ unterscheiden sich von Funktionsdefinitionen wie $f(x) = x + 1$ dadurch, dass sie keine Namen für die dargestellten Funktionen (wie „f“) oder den Funktionswert (wie „y“ in einer Funktionsgleichung) verwenden (λ ist ein Operator und kein Name!). Man nennt die so dargestellten Funktionen deswegen auch **anonym**. Um (mehrfachen) Bezug auf eine anonyme Funktion nehmen zu können, kann man ihrem Abstraktionsausdruck einen Namen geben; dies geschieht beispielsweise durch die Schreibweise $succ \equiv \lambda x.\, x + 1$. Die Benennung erfolgt hierbei aber nur auf der Metaebene (auf der Ebene der Sprache, in der wir über Lambda-Ausdrücke schreiben); sie ist nicht Teil des Lambda-Kalküls, in dem Funktionen trotz solcher Benennungen anonym bleiben.

[16] Leider bezeichnet man in anderen Kontexten der funktionalen Programmierung mit Bindung auch die *Bindung einer Variable an einen Wert*. Dies kann hier aber nicht gemeint sein, da wir ja mit der Einführung einer Variable gerade von Werten abstrahieren.

Funktionsparameter

In einer anonymen Funktion wie $\lambda x.\, x + 1$ bezeichnet man die Abstraktionsvariable x auch als **Parameter**. Die Bezeichnung Parameter tritt hierbei, sowie in der funktionalen Programmierung allgemein, an die Stelle der in der imperativen Programmierung gebräuchlichen Bezeichnung *formaler Parameter* (s. Abschnitt 2.1.3). Wir werden deswegen ab sofort nur noch von „Parameter“ sprechen. Weiter unten (in Abschnitt 3.3.2) werden wir dann noch den Begriff des tatsächlichen Parameters durch den des Arguments ersetzten; dies ist zwar üblich, aber dennoch insofern etwas unglücklich, als man in der Mathematik von x in $f(x)$ als Argument spricht (wir würden es hier als Parameter bezeichnen).

3.3.2 Funktionsanwendung als Substitution

Die Abstraktion von konkreten Werten in Ausdrücken lässt sich umkehren, indem wir die Abstraktionsvariable x durch einen Wert substituieren. Damit erhalten wir alle Berechnungen (in der Regel unendlich viele) zurück, von denen der Lambda-Ausdruck abstrahiert. Bei dieser **Funktionsanwendung** genannten Umkehrung fällt mit der Abstraktionsvariable dann auch deren Bindung weg. Im Beispiel der Anwendung der Funktion $\lambda x.\, x + 1$ auf 1 bleibt nach Substitution von x mit 1 und Wegfall der Bindung nur noch $1 + 1$ über.

Schreibweise der Funktionsanwendung im Lambda-Kalkül

Anders als bei der in der Mathematik vorherrschenden (Eulerschen) Schreibweise $f(1)$ schreibt man die Funktionsanwendung im Lambda-Kalkül üblicherweise ohne Klammer um das Argument, also beispielsweise als $(\lambda x.\, x + 1)\ 1$ oder $succ\ 1$ und nicht als $(\lambda x.\, x + 1)(1)$ oder $succ(1)$. Die Klammern um den Lambda-Ausdruck $\lambda x.\, x + 1$ selbst sind notwendig, um ihn syntaktisch von dem Ausdruck abzugrenzen, auf den er angewendet wird — ohne die Klammern, also im Ausdruck $\lambda x.\, x + 1\ 1$, wäre $x + 1\ 1$, also die Anwendung des Ergebnisses von $x + 1$ auf 1, Teil des Rumpfes des Lambda-Ausdrucks. Klammern dienen also im Lambda-Kalkül ausschließlich der „Überstimmung“ der Präzedenzen des Bindungs- und des Anwendungsoperators (hier vertreten durch das Leerzeichen zwischen angewendeter Funktion und dem Ausdruck, auf den sie angewendet wird).

Funktionsargument

Man spricht übrigens im Lambda-Kalkül, genau wie in der funktionalen Programmierung allgemein, von dem Ausdruck, auf den eine Funktion angewendet wird, als dem **Argument** der Funktionsanwendung. Die Bezeichnung *Argument* tritt hierbei an die Stelle der in der imperativen Programmierung gebräuchlichen Bezeichnung *tatsächlicher Parameter* (s. Abschnitt 2.1.3). Wir werden deswegen ab sofort nur noch von „Argument“ sprechen.

3.3.3 Funktionen als Werte und Funktionen höherer Ordnung

Obiges Beispiel von einer Funktionsanwendung mag unbefriedigend erscheinen, weil kein Funktionswert berechnet wird — stattdessen bleibt der Ausdruck $1 + 1$ als Residual. Tatsächlich ist aber auch weder $(\lambda x.\, x + 1)\ 1$ noch $1 + 1$ ein gültiger Ausdruck des Lambda-Kalküls in seiner reinen Form: Sowohl Zahlen als auch die auf ihnen definieren Operationen (wie die Addition)

sind nicht Teil der Sprache und müssen darin codiert werden.[17] Was aber, wenn nicht Zahlen, sind die Werte des Lambda-Kalküls?

Funktionen sind Werte

Wenn der Lambda-Kalkül wirklich minimal sein, also so wenige Sprachelemente wie nur möglich enthalten soll, dann bleiben nur die Funktionen selbst als Werte übrig. Bevor wir uns der Frage zuwenden, ob dies ausreichen kann, die Mathematik zu begründen (eine Frage, die im Kontext der Programmierung nicht zentral ist), schauen wir uns an, was dies für die Abstraktion und die Funktionsanwendung bedeutet: Es bedeutet, dass Variablen für Funktionen stehen und bei der Anwendung der Abstraktionen durch Funktionen als Werte substituiert werden können. Dies erlaubt es, im Rumpf einer Lambda-Abstraktion eine Variable wie eine Funktion anzuwenden (wie es beispielsweise im Ausdruck $\lambda x.\, x\ y$ geschieht), denn sie kann ja für eine stehen! Übertragen auf die imperative Programmierung würde das bedeuten, dass bei einem Funktionsaufruf `x(y)` (der einer Anwendung im Lambda-Kalkül entspricht) im Programmtext gar nicht klar ist, welche Funktion aufgerufen wird — dies kann erst zur Laufzeit, also dynamisch, bestimmt werden, nämlich wenn der Wert von `x` bekannt ist. Da Funktionen somit Werten gleichgestellt sind, nennt man sie auch „Bürger erster Klasse“ (wobei sie im reinen Lambda-Kalkül die einzigen Bürger sind).

Funktionen höherer Ordnung

Wenn Werte im Lambda-Kalkül also stets Funktionen sind, dann kann eine Funktionsanwendung im Lambda-Kalkül selbst allenfalls eine Funktion als Wert liefern. Funktionen bilden im Lambda-Kalkül also Funktionen auf Funktionen ab — sie sind somit definitionsgemäß **Funktionen höherer Ordnung**.[18] Die Einbettung von Funktionen höherer Ordnung in den Kontext der Funktionsanwendung ist denn auch ein zentraler Beitrag des Lambda-Kalküls zur Programmierung, wie wir sie heute kennen und schätzen. Sie ist insbesondere auch für die objekt-funktionale Programmierung wesentlich.

Wie aber können nur Funktionen für Werte stehen, die wir allgemein als gegeben annehmen, also beispielsweise für Zahlen? Und wie können wir Operatoren auf solchen Werten als Funktionen definieren, wenn uns dafür nichts als die Substitution zur Verfügung steht? Die Antwort hat Church selbst gegeben mit dem, was wir heute *Church-Codierungen* nennen. Bevor wir uns diese anschauen (in Abschnitt 3.3.10), wenden wir uns der formalen Syntax und Semantik des

[17] Wir erinnern uns, dass der Lambda-Kalkül als Alternative zur Mengentheorie der Grundlegung der Mathematik dienen sollte. Zahlen und die Rechenoperationen auf diesen aus noch primitiveren Elementen herzuleiten ist aber gerade die vornehmste Aufgabe einer solchen Grundlegung. In einer Fundierung der Mathematik in der Mengentheorie werden natürliche Zahlen deswegen als Mengen eingeführt; bei einer Verwendung von Funktionen anstelle von Mengen sollten eine ähnlich minimalistische Fundierung gewählt werden.

[18] Tatsächlich sind im ungetypten Lambda-Kalkül alle Funktionen von höherer Ordnung: Funktionen erster Ordnung gibt es nicht, weil es keine Werte gibt, die keine Funktionen sind. Insofern ist es eigentlich unsinnig, im ungetypten Lambda-Kalkül von der Ordnung von Funktionen zu sprechen, denn diese wird erst durch Typen eingeführt. Das gleiche gilt sinngemäß für die Bezeichnung von Funktionen als „Bürger erster Klasse“ (engl. first-class citizens), denn Funktionen stehen im (ungetypten) Lambda-Kalkül nicht gleichberechtigt neben anderen Werten, da es diese anderen Werte gar nicht gibt.

Lambda-Kalküls zu, zum einen, weil es zeigt, wie minimal diese Sprache wirklich ist, zum anderen, weil es zeigt, wie Syntax und Semantik auch ohne die Angabe eines Compilers oder Interpreters festgelegt werden können (vgl. dazu Kapitel 1.5).

3.3.4 Syntax des Lambda-Kalküls

Die Syntax des Lambda-Kalküls ist denkbar simpel: In Backus-Naur-Form (BNF) notiert wird sie durch die Regel

$$e ::= x \mid \lambda x.e \mid e\,e$$

definiert. Demnach ist ein **Ausdruck** (englisch expression, auch **Term** genannt[19]) e des Lambda-Kalküls entweder

- eine **Variable** x oder
- eine **Abstraktion** $\lambda x.e$, also eine anonyme Funktion bestehend aus dem Bindungsoperator λ, der gebundenen Variable x, dem Punkt als Trennzeichen und dem Rumpf e, der wiederum ein Ausdruck ist, oder
- die **Anwendung** $e\,e$ eines Ausdrucks auf einen Ausdruck (Funktionsanwendung; man beachte hier nochmals die fehlende Klammerung des Arguments der Anwendung).

Die Funktionsanwendung ist übrigens linksassoziativ, d. h., $e\,e\,e \equiv (e\,e)\,e$.

Die einfachste Form eines Ausdrucks im Lambda-Kalkül ist demnach x; alle anderen Ausdrücke sind aus Teilausdrücken zusammengesetzt. In der obigen Grammatikregel ist x übrigens kein *Terminal*, wie Sie es vielleicht in einer klassischen *Produktionsregel* einer *kontextfreien Grammatik* erwarten würden, sondern eine **Metavariable**; die Variablen in Ausdrücken des Lambda-Kalküls können nämlich beliebige Namen (wie eben x, aber auch y, x_1 oder f) tragen, x ist hier nur ein Platzhalter. Zusätzlich können (in der konkreten Syntax, also der linearen Form von Ausdrücken; s. Abschnitt 1.5.1) Klammern gesetzt werden, um die intendierte Interpretation eines Ausdrucks (festgehalten als Syntaxbaum) sicherzustellen.

Man beachte, dass die Grammatik der Sprache des Lambda-Kalküls rekursiv ist — sie umfasst also unendlich viele verschiedene Ausdrücke. Jeder Ausdruck für sich ist jedoch endlich.

induktive Definition

Übrigens: Die Sprache des Lambda-Kalküls lässt sich, als unendliche Menge von Sätzen, auch wie folgt *induktiv* definieren:

- x ist ein Satz der Sprache des Lambda-Kalküls.

[19] Tatsächlich ist der Begriff Term in der Literatur zum Lambda-Kalkül gebräuchlicher. Da wir uns hier aber im Kontext von Programmiersprachen bewegen und Ausdrücke dort allgegenwärtig sind (vgl. Abschnitt 2.1.3), bleiben wir hier bei Ausdruck, schon um die Parallelen herzustellen.

- Wenn e ein Satz der Sprache des Lambda-Kalküls ist, dann ist auch $\lambda x.\, e$ ein Satz der Sprache des Lambda-Kalküls.
- Wenn e_1 und e_2 Sätze der Sprache des Lambda-Kalküls ist, dann ist auch $e_1\ e_2$ ein Satz der Sprache des Lambda-Kalküls.
- Nichts anderes ist ein Satz der Sprache des Lambda-Kalküls.

Dieser Sachverhalt lässt sich auch als eine Menge von **Inferenzregeln** formulieren, in denen Λ für die Menge der Ausdrücke des Lambda-Kalküls stehe:

$$x \in \Lambda \qquad \frac{e \in \Lambda}{\lambda x.\, e \in \Lambda} \qquad \frac{e_1 \in \Lambda \quad e_2 \in \Lambda}{e_1\ e_2 \in \Lambda}$$

Hierbei hat die erste Regel keine Prämisse: Sie ist ein **Axiom**. Wenn man nun einen beliebigen Ausdruck E hat und wissen will, ob er zur Sprache des Lambda-Kalküls gehört, dann kann man versuchen, mittels **Inferenz** aus obigen Regeln abzuleiten (zu inferieren), ob E ein Element von Λ ist. Am Beispiel $E = \lambda y.\, y$:

1. E hat die syntaktische Form $\lambda x.\, e$ mit $x = y$ und $e = y$, also ist $E \in \Lambda$, wenn $y \in \Lambda$.
2. y hat die syntaktische Form x, also ist $y \in \Lambda$.

Daraus folgt $E \in \Lambda$. Derartige **Ableitungen** (Inferenzen) werden uns später (in den Abschnitten 3.4 aus dieser Lektion und Abschnitt 25.2 aus Lektion 3) wieder begegnen. Der Abschluss, also dass alles, was sich nicht mittels obiger beider Regeln und des Axioms ableiten lässt, nicht gilt, bleibt hierbei (bei der Angabe von Inferenzregeln wie den obigen) übrigens häufig implizit.

Einsatz von Pattern matching

Man beachte, dass die Anwendung obiger Inferenzregeln ein sog. **Pattern matching** beinhaltet: Dabei wird der Ausdruck E gegen die Muster (patterns) x, $\lambda x.\, e$ und $e_1\ e_2$ der Konklusionen der Regeln gehalten und geprüft, mit welchem dieser Muster es durch Belegung (Instanziierung) der darin enthaltenen Metavariablen x, e, e_1 und e_2 in Übereinstimmung gebracht werden kann. Bei einem Match werden die Belegungen der Metavariablen (im gegebenen Fall $x = y$ und $e = y$) verwendet, um (rekursiv) die Erfüllung der Prämissen (falls vorhanden — bei einem Axiom ist das nicht der Fall) zu prüfen.

Das Pattern matching, das hier auf der Ebene der *Metasprache* (der Sprache zur Beschreibung der Sprache des Lambda-Kalküls) eingesetzt wird, findet man zunehmend auch als Sprachkonstrukt von Programmiersprachen (wo es eine Bedingung der Verzweigung darstellt und somit der Steuerung des Kontrollflusses dient). Pionier war hier die Sprache ML (die bezeichnenderweise gerade als Metasprache entwickelt wurde, daher der Name). Allerdings gab es damals schon die *logische Programmiersprache Prolog*, die mit der *Unifikation* eine allgemeinere Form des Pattern matching anbietet.

3.3.5 Semantik des Lambda-Kalküls: β-Reduktion

Die Semantik des Lambda-Kalküls legt fest, wie aus Funktionsanwendungen Werte „berechnet“ werden. Das Besondere dabei ist, dass bei dieser „Berechnung“ nicht im herkömmlichen Sinn gerechnet wird (deswegen die Anführungsstriche), sondern sie als primitives *Zeichenspiel* durchgeführt wird: Wie bereits in Abschnitt 3.3.2 angedeutet, erfolgt die Anwendung durch *Substitution*, also die Ersetzung einer Zeichenkette durch eine andere.

Redexe

Wie aber sieht nun dieses Zeichenspiel genau aus? Wenn wir uns konkrete Funktionsanwendungen anschauen, dann sollten diese die Form $(\lambda x.\, e_1)\; e_2$ haben[20]; andere Formen wie beispielsweise $x\; e$ sind zu unbestimmt, um behandelt werden zu können (wir wissen nicht einmal, ob x überhaupt eine Funktion ist, die angewendet werden könnte). Wir nennen Ausdrücke der Form $(\lambda x.\, e_1)\; e_2$ **reduzierbare Ausdrücke** (engl. reducible expressions) oder **Redexe** (der von Church dafür eingeführte Name).

β-Reduktion

Wenn wir also einen Redex der Form $(\lambda x.\, e_1)\; e_2$ vorliegen haben, dann können wir ihn — rein textuell — durch den Ausdruck ersetzen, der entsteht, wenn wir in e_1 alle Vorkommen von x durch e_2 ersetzen (die Bindung von x fällt, wie schon in Abschnitt 3.3.2 gezeigt, zusammen mit x weg). Bei dieser Ersetzung handelt es sich um eine *Variablensubstitution*, wie wir sie auch aus der Mathematik kennen (Abschnitt 3.1): e_2 wird in e_1 für x eingesetzt. Wir schreiben diese Substitution als $[e_2/x]e_1$ und können somit

$$(\lambda x.\, e_1)\; e_2 \longrightarrow_\beta [e_2/x]e_1$$

als Regel unseres Zeichenspiels zur Funktionsanwendung im Lambda-Kalkül festhalten; sie wird als „$(\lambda x.\, e_1)\; e_2$ wird zu dem Ausdruck, der aus der Ersetzung aller Vorkommen von x in e_1 durch e_2 entsteht, reduziert“ gelesen. Dabei steht der Pfeil $\longrightarrow_\beta$ für die auf Substitution basierende **Reduktionsrelation**, von Church **β-Reduktion** genannt.[21] Man beachte, dass die (Syntax der) Substitution $[e_2/x]e_1$ nicht zur Sprache (Syntax) des Lambda-Kalküls gehört (vgl. dazu Abschnitt 3.3.4); sie ist vielmehr Teil der *Metasprache*, mit deren Hilfe wir die Semantik des Lambda-Kalküls spezifizieren.

verschiedene Schreibweisen der Substitution

Bei der mathematischen Notation der Substitution scheint es keinen Konsens zu geben: Guy Steele hat in einer Keynote 22 verschiedene Notationen in der Literatur gefunden, wobei ihm teilweise sogar verschiedene Notationen in einem Papier begegnet sind. Hilfreich ist vielleicht, wenn man sich die Substitution als eine syntaktische Funktion σ vorstellt, die einen Ausdruck e auf einen anderen abbildet. Für die Anwendung dieser Funktion kann man dann (in Anlehnung an den Lambda-Kalkül ohne Klammern) $\sigma\, e$ schreiben. So kann man die Substitution von x in e durch e' als eine Funktion σ auffassen, die jedes Vorkommen von x in e auf e' abbildet und die alle anderen Teilausdrücke

[20] Man beachte, dass die Form $(\lambda x.\, e_1)\; e_2$ ein Muster für unendliche viele Ausdrücke ist: x, e_1 und e_2 sind hier Metavariablen, stehen also für beliebige Variablen bzw. Teilausdrücke.

[21] Es handelt sich tatsächlich um eine Relation und nicht um eine Funktion, da manche Lambda-Ausdrücke kein, andere mehrere sog. **Redukte** (das Ergebnis der Reduktion) haben.

von e unverändert lässt. So kommt man dann zu der Schreibweise $[x \mapsto e']e$, die man auch gelegentlich findet. Ich verwende hier stattdessen $[e'/x]e$, das sich mit der englischen Sprechweise „substitute e' for x in e“ deckt (nicht jedoch mit der deutschen „substituiere x in e durch e'“); vgl. dazu die Sprechweise bei der Substitution in Abschnitt 3.1).

Wir schauen uns die Wirkweise der Reduktionsregel an einem Beispiel an. Wenn wir die Funktion $\lambda x.\, x\ x$ auf die Funktion $\lambda y.\, y$ anwenden, ergibt sich gemäß obiger Regel und der damit verbundenen Substitution

$$(\lambda x.\, x\ x)\ \lambda y.\, y \longrightarrow_\beta (\lambda y.\, y)\ \lambda y.\, y$$

also eine weitere Funktionsanwendung als Ergebnis. Auf dieses kann die Reduktionsregel neuerlich angewendet werden:

$$(\lambda y.\, y)\ \lambda y.\, y \longrightarrow_\beta \lambda y.\, y$$

Normalform

Das Ergebnis stellt nun einen Wert (in Form einer Funktion) dar und kann somit nicht weiter ausgewertet werden. Von Ausdrücken, die nicht weiter reduziert werden können, sagt man, sie seien in **Normalform**. Neben Werten wie $\lambda x.\, x$ (oder $\lambda y.\, y$, der mit $\lambda x.\, x$ gleichbedeutend ist) sind auch Ausdrücke wie $x\ x$ in Normalform (da es hier, trotz der syntaktischen Form einer Anwendung, nichts zu reduzieren gibt — x ist eine Variable und keine Funktion).

Konfluenz oder Church-Rosser

Nun können Ausdrücke des Lambda-Kalküls beliebig tief geschachtelt sein und somit mehrere Redexe enthalten. Die Fragen, die sich somit stellen, sind,

1. in welcher Reihenfolge die Redexe reduziert werden sollen und
2. ob das Ergebnis von der Reihenfolge abhängt.

Es ist eine zentrale Eigenschaft des Lambda-Kalküls, dass das Ergebnis der Reduktion auf eine Normalform *nicht* von der Reihenfolge der Reduktionsschritte abhängt, die Reihenfolge aus theoretischer Sicht also irrelevant ist. Man nennt diese Eigenschaft (die auch auf andere Systeme anwendbar ist) **Konfluenz** oder, nach den beiden, die sie zuerst für den Lambda-Kalkül nachweisen konnten, **Church-Rosser**.

Call-by-value vs. Call-by-name

Während die Reihenfolge der Auswertung im reinen Lambda-Kalkül theoretisch (für das Ergebnis einer Berechnung) ohne Belang ist, hat sie in der Praxis, insbesondere bei Erweiterungen des Lambda-Kalküls zu (objekt-)funktionalen Programmiersprachen, erhebliche Auswirkungen. Vermutlich die gebräuchlichste (und uns am vertrautesten erscheinende) Reihenfolge ist die, nach der ein Funktionsargument stets reduziert wird, *bevor* die dazugehörige Funktionsanwendung reduziert wird. Auf die Mathematik benannter Funktionen erster Ordnung übertragen heißt das, dass bei einem Ausdruck $f\big(g(1)\big)$ erst $g(1)$ berechnet wird und dann das Ergebnis dieser Berechnung als Argument der Anwendung von f eingesetzt wird. In der Programmierung wird dies auch **Call-by-value** genannt, wobei dieser

Name etwas irreführend erscheinen mag, weil er dort, insbesondere in der imperativen Programmierung (in der es für gewöhnlich keine Funktionen höherer Ordnung gibt), von *Call-by-reference* abgegrenzt wird (Abschnitt 2.1.6), was mit dem Lambda-Kalkül wiederum nichts zu tun hat. In der funktionalen Programmierung, hier vertreten durch den Lambda-Kalkül, mit ihren Funktionen höherer Ordnung ist es aber auch möglich, den Ausdruck $g(1)$ selbst (und nicht seinen Wert) in der Funktionsdefinition von f für x einzusetzen und dann zu versuchen, den resultierenden Ausdruck auszuwerten. Wenn also beispielsweise f mit $f(x) = x + 1$ definiert ist, dann ergibt $f\big(g(1)\big)$ auf diese Weise $g(1) + 1$. Man nennt diese Reihenfolge der Reduktion bei Funktionsanwendungen auch **Call-by-name**; sie ist in funktionalen Sprachen wie Haskell der Standard (und dort als *Lazy evaluation* implementiert).

3.3.6 Freie Vorkommen von Variablen und offene Ausdrücke

Wie bereits in Abschnitt 3.3.5 angedeutet gibt es im Lambda-Kalkül Ausdrücke, die weder selbst Werte sind noch sich auf Werte reduzieren lassen. Einfache Beispiele hierfür sind die Ausdrücke x, $x\ y$ und $x\ \lambda y.\ y$, die in *Normalform*, aber keine Werte sind, die sich also, da die Form $(\lambda x.\ e_1)\ e_2$ nicht in ihnen vorkommt, nicht reduzieren lassen.

freie und gebundene Variablen

Vorkommen von Variablen wie das von x in obigen drei Ausdrücken (und auch das von y im zweiten) nennt man **frei**: sie sind nicht (durch einen Binder wie λx oder λy) **gebunden**.[22] Variablen können auch in *Rümpfen* von Abstraktionen frei vorkommen: So ist im Rumpf von $\lambda x.\ y\ x$, dem Ausdruck $y\ x$, das Vorkommen der Variable x zwar durch den Binder λx gebunden, das der Variable y aber frei. Im Rumpf von $\lambda y.\ \lambda x.\ y\ x$, dem Teilausdruck $\lambda x.\ y\ x$, ist hingegen auch das Vorkommen von y nicht frei, da es durch das im Gesamtausdruck vorangestellte λy gebunden ist. Man beachte jedoch, dass in beiden Ausdrücken, wenn sie isoliert betrachtet werden (also ohne den Binder λx und somit nicht als Rümpfe), x frei ist. Ob eine Variable in einem (Teil-)Ausdruck gebunden oder frei ist, hängt also auch vom Kontext des Ausdrucks, dem Geamtausdruck, ab.

geschlossene und offene Ausdrücke

Ausdrücke, in denen keine freien Variablen vorkommen, nennt man **geschlossen**, alle anderen **offen**. Wie wir gesehen haben, können offene Ausdrücke wie $x\ y$ nicht ausgewertet (zu einem Wert reduziert) werden. Deswegen betrachtet man in vielen Kontexten nur geschlossene Ausdrücke als *wohlgeformt*. Geschlossenheit wird häufig auch in der Programmierung verlangt: So dürfen in vielen Programmiersprachen keine Variablen verwendet werden, die nicht zuvor im Programm deklariert (oder initialisiert) worden sind. Der Grund hierfür ist einfach: Bei freien Variablen ist nicht geregelt, woher ihre Werte kommen, so dass das Ergebnis ihrer Verwendung nicht durch ein Programm bestimmt werden kann. Die mangelnde Reduzierbarkeit obiger Ausdrücke spiegelt genau dies wider: Nicht gebundene Variablen können auch nicht durch Werte substituiert werden, so dass sie weder

[22] Tatsächlich nennt man die Vorkommen von Variablen in Ausdrücken frei oder gebunden, nicht die Variablen selbst. Die Redeweisen „freie“ bzw. „gebundene Variablen“ sind aber dennoch gebräuchlich, was in Ordnung ist, solange klar ist, auf welchen Ausdruck sich die Einordnung bezieht.

selbst für Werte stehen noch ihre Anwendungen auf andere Ausdrücke ausgewertet werden können.[23]

3.3.7 Das Phänomen der Divergenz

Die Verwendung des Begriffs *Reduktion* legt nahe, dass ein Ausdruck mit jedem Schritt kleiner (oder zumindest einfacher) wird. Damit wäre aber auch klar, dass alle Reduktionsschrittfolgen endlich sind und somit jede Auswertung eines Ausdrucks über Reduktion irgendwann terminieren muss. Dass aber weder das eine noch das andere der Fall ist, zeigt der Ausdruck $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$, der in einem Schritt auf sich selbst reduziert wird (und den man deswegen unendlich weiter „reduzieren" kann, ohne dass sich etwas änderte):

$$(\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \longrightarrow_\beta (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

Dies mag zunächst enttäuschen, ist aber tatsächlich eine Voraussetzung für die Universalität des Lambda-Kalküls: Er kann sogar Endlosschleifen! Man nennt eine unendliche Reduzierbarkeit auch **Divergenz** (wobei hiermit nicht das Gegenteil von *Konvergenz* im mathematischen Sinn gemeint ist, sondern lediglich die mangelnde Terminierung; Konvergenz bezeichnet in diesem Kontext eher Terminierung in Kombination mit Konfluenz).

Es wird übrigens der Ausdruck $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ oft als Beispiel für eine *Endlosschleife* im Lambda-Kalkül angeführt. Die „Schleife" liegt dabei aber ausschließlich in der Wiederholung der Reduktion, die immer durchgeführt, wenn ein Ausdruck mehrere Reduktionsschritte benötigt, um in eine Normalform überführt zu werden. Insofern handelt es sich bei diesem Ausdruck nicht um eine Schleife — insbesondere codiert sie keine Kontrollstruktur im Lambda-Kalkül. Mehr dazu in Abschnitt 3.3.11.

Ω-Ausdruck

Der Ausdruck $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ wird übrigens auch **Ω-Ausdruck** genannt. Er ist ein leuchtendes Beispiel dafür, welche kleinen Wunder eine minimale Syntax gepaart mit einer nicht minder minimalen Semantik hervorbringen kann.

3.3.8 Das Problem mit den Variablennamen

Wie bereits oben bemerkt repräsentieren die Abstraktionen $\lambda y.\, y$ und $\lambda z.\, z$ dieselbe Funktion. Man darf sich also fragen, welche Rolle die Namen von Variablen im Lambda-Kalkül tragen. Bei genauer Betrachtung (und wie bereits in Abschnitt 1.5.1 bemerkt) sind die Namen der Variablen nur ein technischer Umstand, der wegen der Linearität von Syntax benötig wird, um festzuhalten, dass an mehreren Stellen in einem Ausdruck dasselbe Element vorkommt. Hätte

[23] In diesem Zusammenhang kann man sich auch fragen, ob ein Ausdruck wie $\lambda x.\, y$ wirklich ein Wert sein kann, wenn y frei, also gar nicht bestimmt ist. Der (pure) Lambda-Kalkül macht hier aber keine Einschränkungen. Interessant in diesem Zusammenhang ist auch, dass die Werte $\lambda x.\, y$ und $\lambda x.\, z$ nicht gleich sind (auch nicht α-äquivalent, also gleich bis auf Umbenennung ihrer Variablen), da die freien Variablen y und z prinzipiell für verschiedene Werte stehen können.

die Syntax die Form eines Graphen, bräuchten wir die Namen nicht — an ihrer Stelle stünden dann einfach Kanten zu demjenigen Knoten, der die genannte Variable repräsentiert.[24] Tatsächlich bezeichnet man Abstraktionen, die sich nur im Namen der Abstraktionsvariable unterscheiden (wie $\lambda y.\, y$ und $\lambda z.\, z$) als äquivalent und nennt diese Äquivalenz (nach Church) **α-Äquivalenz**.

α-Konversion

α-Äquivalenz bedeutet, dass wir die gebundenen Variablen eines Lambda-Ausdrucks prinzipiell umbenennen dürfen, ohne seine Bedeutung zu verändern. Manche Autoren definieren, passend zu dieser Äquivalenz, auch eine spezielle Form der Reduktion, α-Reduktion genannt: Sie erlaubt es, eine Abstraktion durch eine andere zu ersetzen, die sich nur im Namen der Abstraktionsvariable unterscheidet. Diese Reduktion wird von manchen Autoren auch α-Konversion genannt.

Name capture

Allerdings sind der α-Konversion enge Grenzen gesetzt. So dürfen wir im Ausdruck $\lambda x.\, \lambda y.\, x\, y$ die Variable y nicht in x umbenennen, denn in $\lambda x.\, \lambda x.\, x\, x$ wird auch das ganz rechts stehende x, im Gegensatz zum früheren y, an das rechte (innere) λx gebunden. Somit werten $(\lambda x.\, \lambda y.\, x\, y)\, \lambda z.\, z$ und $(\lambda x.\, \lambda x.\, x\, x)\, \lambda z.\, z$ zu unterschiedlichen Termen aus (nämlich zu $\lambda y.\, (\lambda z.\, z)\ y$ und $\lambda x.\, x\, x$). Man spricht hier auch von **Name capture**, also dem Umstand, dass die innere Einführung von x durch das rechte λx in seinem Geltungsbereich (Scope) alle Referenzen auf das äußere x abfängt. Dies ist kein spezielles Problem des Lambda-Kalküls, sondern kommt in allen Programmiersprachen, die Namen verwenden und geschachtelte Geltungsbereiche erlauben, vor.

Name capture bei Substitutionen

Name capture ist auch eine Herausforderung für die Substitution, nämlich wenn der Ausdruck, der für eine Abstraktionsvariable eingesetzt wird, freie Variablen enthält, die durch die Substitution in den Geltungsbereich eines Binders mit dem gleichen Variablennamen fallen. Ein einfaches Beispiel hierfür ist $(\lambda x.\, \lambda y.\, x)\ y$: Ohne eine vorherige α-Reduktion wird daraus $\lambda y.\, y$, d. h., die zuvor freie Variable y ist jetzt gebunden. Hätte man das Argument y zuvor in z umbenannt (per α-Reduktion), dann wäre daraus, als Ergebnis der β-Reduktion von $(\lambda x.\, \lambda y.\, x)\ z$, $\lambda y.\, z$ geworden, d. h., die Variable z (vormals y) bliebe weiterhin frei.

Tatsächlich machen (zufällige) Namenskollisionen die Substitution zu einem recht aufwändigen Prozess, der deswegen hier nicht weiter ausgeführt werden soll (interessierte Leser seien auf die Standardliteratur verwiesen). Interessant ist jedoch, dass die gleichen Probleme auch in allen Programmiersprachen mit geschachtelten Namensräumen (Geltungsbereichen) auftreten, nämlich wenn man in Programmen dieser Sprachen Namen von Programmelementen ändert oder Programmelemente von einem Namensraum in einen anderen verschiebt. Man sieht auch hier, dass der eher theoretisch anmutende Lambda-Kalkül ein valides Modell für die praktische Programmierung sein kann.

[24] Dies ignoriert natürlich vollständig die Bedeutung, die Namen für menschliche Betrachter haben: Ihre Wahl sorgt, je nach Betrachter, für die richtige, die falsche oder gar keine Intuition.

3.3.9 Funktionen mit mehreren Argumenten: Currying

Insbesondere in der Informatik haben Funktionen regelmäßig mehr als nur ein Argument. Man spricht dann von **mehrstelligen Funktionen**. Im Lambda-Kalkül müssten mehrstellige Funktionen über mehrere Abstraktionsvariablen in ihrem Binder verfügen; um die einzuführen, müssten wir seine Syntax und Semantik erweitern.

Glücklicherweise kann man, wenn man Funktionen höherer Ordnung zur Verfügung hat, mehrstellige Funktionen leicht als geschachtelte einstellige Funktionen darstellen, wobei die äußere Funktion das jeweils erste Argument entgegennimmt und dies in ihren Rückgabewert, die innere Funktion, die selbst ein Argument, das somit zweite, entgegennimmt, als das erste Argument einsetzt. So lässt sich beispielsweise die Addition als Schachtelung zweier Funktionen darstellen:

$$add \equiv \lambda x.\, \lambda y.\, x + y$$

was eine zweistellige Abstraktion $\lambda x, y.\, x + y$ (die unsere Syntax gar nicht vorsieht) ersetzt. Der Ausdruck $add\ 1\ 2$ beispielsweise reduziert dann in einem ersten Schritt zu $\lambda y.\, 1 + y$ und im zweiten zu $1 + 2$. Man nennt dieses Verfahren der Darstellung mehrstelliger mithilfe einstelliger Funktionen **Currying**, weil der Mathematiker Haskell Curry es populär gemacht hat (es war aber schon vorher bekannt). Es hat mittlerweile auch Einzug in Programmiersprachen gehalten (z. B. in Scala).

Vorteile des Currying; partielle Auswertung

Currying ist ein gutes Beispiel dafür, wie man durch Programmiermuster Sprachen erweitern kann, ohne ihre Syntax oder Semantik anzufassen. Damit bleibt die Sprache klein und somit zugänglicher für formale Betrachtungen (wie Beweise etc.). Zugleich erlaubt Currying die **partielle Auswertung** von Funktionen: Betrachtet man *add* als zweistellige Funktion und hat man einen Fall der Anwendung dieser Funktion, in dem, aus welchem Grund auch immer, das erste Argument feststeht (wie es beispielsweise bei $add\ 1\ x$ der Fall wäre), dann lässt sich die Funktion partiell auswerten (im gegebenen Beispiel zu $\lambda y.\, 1 + y$). Bei einer späteren Anwendung der Funktion muss dann nur noch das zweite Argument eingesetzt werden. Solche partiellen Auswertungen werden beispielsweise von Compilern zur Übersetzungszeit zum Zweck der Optimierung durchgeführt.

3.3.10 Church-Codierungen

Während die Ausdehnung der Funktionsanwendung auf Funktionen als Argumente und Resultate als der wichtigste praktische Beitrag des Lambda-Kalküls zur Programmierung erscheinen mag, hat auch die Tatsache, dass er allein ausreicht, um alle berechenbaren (algorithmischen) Funktionen zu spezifizieren (und damit das zu können, was irgendein Programm kann), ihren Wert: Sie bedeutet nämlich, dass man den Lambda-Kalkül in gewissen Kontexten als Repräsentant für alle Programmiersprachen verwenden kann. Dies ist insbesondere für bestimmte Beweise von immensem Vorteil: Weil der Lambda-Kalkül nur drei syntaktische Formen kennt (s. Abschnitt 3.3.4), fallen Beweise dafür vergleichsweise schlicht aus. Es ist also

durchaus auch von praktischem Interesse, wenn sich mit dem einfachen Lambda-Kalkül alles machen lässt.

Wie lassen sich aber nun primitive Werte wie Zahlen oder die Booleschen Wahrheitswerte, die wir mindestens benötigen, um „alle“ Berechnungen abzudecken, im Lambda-Kalkül ausdrücken? Der Trick ist, nicht die Werte selbst, sondern ihre Verwendungen durch Funktionen zur Grundlage der Codierung zu machen. Am Beispiel der Wahrheitswerte gelingt das folgendermaßen: Definiert man

$$\mathit{true} \equiv \lambda x.\, \lambda y.\, x \qquad \mathit{false} \equiv \lambda x.\, \lambda y.\, y$$

so ergibt die Anwendung von *true* (als einer curryfizierten zweiwertigen Funktion) auf zwei Werte per β-Reduktion den ersten davon:

$$\mathit{true}\ (\lambda a.\, a)\ (\lambda b.\, b) \longrightarrow_\beta (\lambda y.\, \lambda a.\, a)\ (\lambda b.\, b) \longrightarrow_\beta \lambda a.\, a$$

die von *false* stattdessen den zweiten:

$$\mathit{false}\ (\lambda a.\, a)\ (\lambda b.\, b) \longrightarrow_\beta (\lambda y.\, y)\ (\lambda b.\, b) \longrightarrow_\beta \lambda b.\, b$$

Somit lässt sich die Funktion

$$\mathit{ite} \equiv \lambda i.\, \lambda t.\, \lambda e.\, i\ t\ e$$

definieren, deren Anwendung *ite true x y*, analog zu `if true then x else y`, zu x reduziert und *ite false x y*, analog zu `if false then x else y`, zu y. Auch die logischen Operatoren *and*, *or* und *not* lassen sich für die wie oben definierten Argumente *true* und *false* recht einfach codieren:[25]

$$\mathit{and} \equiv \lambda x.\, \lambda y.\, x\ y\ x \qquad \mathit{or} \equiv \lambda x.\, \lambda y.\, x\ x\ y \qquad \mathit{not} \equiv \lambda x.\, x\ \mathit{false}\ \mathit{true}$$

Man beachte, dass diese Operatoren essentiell von der Verwendung der obigen Definitionen von *true* und *false* abhängen. Setzt man hingegen andere Argumente ein, passiert Unsinn. Solchen Unsinn zu verhindern wäre die Rolle eines *Typsystems*, dem wir uns hier aber noch nicht zuwenden wollen.

Die natürlichen Zahlen können nach demselben Prinzip als Funktionen codiert werden. Grundlage der Codierung ist hier der Umstand, dass es zu jeder natürlichen Zahl eine gibt, die um 1 größer ist und die sich aus der Anwendung einer entsprechenden Funktion, üblicherweise „succ“ genannt, auf die Ursprungszahl ergibt. Mithilfe einer solchen Funktion lässt sich beispielsweise die Zahl 1 als *succ* 0, 2 als *succ* (*succ* 0), usw. darstellen, wobei hier der Clou ist, dass schon der Ausdruck selbst und nicht erst das Ergebnis seiner Reduktion für die jeweilige Zahl steht. Für letzteres müssten wir ja auch eine Definition von *succ* angeben, was aber ein Problem

[25] Andere Codierungen der Wahrheitswerte und ihrer Operatoren sind möglich; die hier gezeigten sind die originalen.

ist, wenn wir weder Zahlen noch die Addition von 1 zur Verfügung haben. Tatsächlich ist ja selbst 0 kein Ausdruck des reinen Lambda-Kalküls.

Der Trick ist nun, Lambda-Ausdrücke für die Zahlen einzuführen, die auf obigem Schema der Zahldarstellung basieren, die aber keine Funktionsanwendungen, sondern Werte (also Funktionen wie oben *true* und *false*) sind, die so konstruiert sind, dass sich die Addition per Substitution darstellen lässt. Diese Additionsfunktion muss dazu nur ihre beiden Argumente so ineinander schachteln, dass wieder ein Lambda-Ausdruck entsteht, der eine Zahl darstellt — dass dies die richtige Zahl ist, ergibt sich daraus, dass sich durch die Schachtelung die Zahl der Anwendungen von *succ* in beiden Argumenten addiert. Dies gelingt mit der Codierung

$$0 \equiv \lambda succ.\, \lambda x.\, x \qquad 1 \equiv \lambda succ.\, \lambda x.\, succ\ x \qquad 2 \equiv \lambda succ.\, \lambda x.\, succ\ (succ\ x) \qquad \cdots$$

in der sowohl *succ* als auch x Abstraktionsvariablen (also unbestimmt) sind. Die Addition zweier Zahlen ergibt sich dann als (textuelle) Schachtelung der Anwendungen von *succ* im Rumpf der Funktionen, deren Anzahl die Zahlen jeweils unterscheidet (keinmal für 0, einmal für 1, zweimal für 2 usw.):

$$add \equiv \lambda y.\, \lambda z.\, \lambda succ.\, \lambda x.\, y\ succ\ (z\ succ\ x)$$

Man beachte, dass bei deren Anwendung $add\ 1\ 2$, genau wie bei jeder anderen Anwendung auf ein Paar von wie oben definierten Zahlen, niemals Werte für die Variablen *succ* oder x eingesetzt werden — diese werden schlicht nicht benötigt. Zahlen und deren Addition haben also keine weiteren Voraussetzungen — sie sind vollständig in der Syntax und der Semantik des Lambda-Kalküls codiert.

3.3.11 Rekursion

Wir können also im Lambda-Kalkül verzweigen und natürliche Zahlen addieren. Für eine Turing-mächtige Sprache brauchen wir allerdings noch Schleifen oder, alternativ, rekursive Funktionsanwendungen.

Eine (direkt) rekursive Funktion wendet sich in ihrem Rumpf selbst an. Da im Lambda-Kalkül Funktionen aber anonym sind, kann eine Funktion sich nicht selbst (wie ein Unterprogramm das kann) referenzieren — sie müsste sich vielmehr selbst enthalten. Daran ändert auch der Umstand nichts, dass man Ausdrücken Namen geben kann: Eine Definition wie $fac \equiv \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * fac\ (n - 1)$ benennt keinen gültigen Lambda-Ausdruck, denn die Ersetzung von *fac* im Rumpf durch den definierenden Term findet kein Ende und liefert somit einen unendlich tief geschachtelten Ausdruck. Solche Ausdrücke gibt es jedoch im Lambda-Kalkül nicht (s. Abschnitt 3.3.4).

Nun ist es allerdings so, dass der Lambda-Kalkül mit seiner β-Reduktion eine Substitution „zur Laufzeit“ bietet. Wir müssen also eine rekursive Funktionsdefinition wie die obige gar nicht „zur Übersetzungszeit“, also im Text der rekursiven Funktion, expandieren, sondern können dies auf die Laufzeit verschieben (wobei ja im Lambda-Kalkül streng genommen auch hier der

Programmtext per Substitution expandiert wird). Das einzige, das wir dafür tun müssen, ist, die rekursiv aufzurufende Funktion einer anderen Funktion als Argument zu übergeben: Da deren Parameter (die Abstraktionsvariable) mehrfach in deren Rumpf auftauchen kann und bei Anwendung der Funktion auf die andere der Parameter mehrfach durch die andere Funktion ersetzt wird, kann der rekursive Aufruf bei jedem Reduktionsschritt einmal „ausgerollt“ werden — den Rest besorgt dann die Schleife, die der Reduktion eines Ausdrucks bis hin zu einer Normalform innewohnt (vgl. dazu Abschnitt 3.3.7). Diesen Wirkmechanismus kann man sich am Beispiel der Funktion $\lambda f.\lambda x.\ f\ (f\ x)$ vergegenwärtigen, die, auf eine Funktion g angewendet, eine Funktion $\lambda x.\ g\ (g\ x)$ liefert, die g zweimal hintereinander auf ihr Argument anwendet. Damit wäre aber schon Schluss — insbesondere bekommen wir so keine beliebig tiefe Rekursion bzw. Schleife bei der Reduktion. Wir suchen also eine, die dies herstellt.

Eine solche Funktion liegt nun mit dem sog. Y-Kombinator vor:

$$\mathrm{Y} \equiv \lambda f.\left(\left(\lambda x.\ f(x\,x)\right)\left(\lambda x.\ f(x\,x)\right)\right)$$

Wir kennen im Rumpf dieser Funktion den Ω-*Ausdruck* aus Abschnitt 3.3.7 in leicht abgewandelter Form wieder: der selbstreplizierenden Anwendung von $\lambda x.\ x\ x$ auf $\lambda x.\ x\ x$ wird hier noch die Anwendung einer — der rekursiv anzuwendenden — Funktion f vorangestellt. Wenn g jetzt für eine solche Funktion steht, dann reduziert $Y\ g$ im ersten Schritt zu

$$\left(\lambda x.\ g\,(x\,x)\right)\left(\lambda x.\ g\,(x\,x)\right)$$

im zweiten zu

$$g\left(\left(\lambda x.\ g\,(x\,x)\right)\left(\lambda x.\ g\,(x\,x)\right)\right)$$

im dritten zu

$$g\left(g\left(\left(\lambda x.\ g\,(x\,x)\right)\left(\lambda x.\ g\,(x\,x)\right)\right)\right)$$

usw. Jede Wiederholung der Reduktion führt zu einem neuen, nach dem gleichen Schema „reduzierbaren“ (in Anführungszeichen weil dabei immer länger werdenden) Ausdruck, dessen schrittweise, nicht terminierende Entstehung dem schrittweisen Ausrollen der (endlosen) Rekursion entspricht. Im Grunde findet hier die oben (wegen fehlender Terminierung) als nicht möglich beschriebene unendliche Expansion statt, allerdings nicht in einem Zug, sondern schrittweise, durch wiederholte Anwendung der Substitution, die in der in Abschnitt 3.3.4 definierten Reduktionsrelation steckt. Man sieht sehr schön, wie hier ein operationales Verfahren die Unendlichkeit eines rekursiven Ausdrucks in eine Endlosschleife übersetzt.

Damit mithilfe des Y-Kombinators eine sinnvolle Rekursion hergestellt werden kann, muss die Funktion, auf die er angewendet wird, selbst einen Rekursionsanfang einbauen. Am obigen Beispiel der Fakultät gelingt dies mittels

$$\mathit{fac} \equiv \lambda f.\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f\ (n-1)$$

Die Berechnung der Fakultät von 5 würde dann beispielsweise durch den Ausdruck $Y\ fac\ 5$ angestoßen. Mach beachte jedoch, dass man hierfür eine besondere Reduktionsreihenfolge wählen muss, da sonst die wiederholte Reduktion von $Y\ fac$ zu einer Endlosschleife führt.

3.4 Vom Lambda-Kalkül zur Programmiersprache

Der Lambda-Kalkül istw ein Kalkül und somit in erster Linie etwas für Bleistift und Papier (ein *Zeichenspiel*). Da er aber auch — genau wie die *Turing-Maschine* — ein universelles *Berechnungsmodell* ist und wir mit Computern zudem programmierbare Universalmaschinen haben, die auf der Basis bestimmter, in ihnen fest verdrahteter, mechanisierter Zeichenspiele beliebige andere Zeichenspiele simulieren können, ergibt sich die Frage, ob der Lambda-Kalkül nicht die Grundlage einer Programmiersprache sein kann. Kann er.

Nun könnte der Lambda-Kalkül zunächst nicht weiter von einem Computer, dessen Gegebenheiten sich in der imperativen Programmierung recht deutlich widerspiegeln, entfernt sein. Das können wir aber ignorieren, denn eine 1:1-Umsetzung der auf *Substitution* basierenden Reduktion von Ausdrücken des Lambda-Kalküls in einer imperativen Programmiersprache ist nicht weiter schwer. Auf der anderen Seite ist aber das Formulieren von Algorithmen im (reinen) Lambda-Kalkül recht aufwendig, und zwar nicht zuletzt gerade deswegen, weil er nur so wenige Sprachkonstrukte enthält. Gleichzeitig ist die Abarbeitung (Ausführung) von im Lambda-Kalkül verfassten Algorithmen reichlich ineffizient, da hierzu das Programm (der zu reduzierende Ausdruck) wiederholt umgeschrieben werden muss. Wir müssen den Lambda-Kalkül also etwas aufbohren, um ihn für die Programmierung zugänglicher zu machen. Doch zunächst einmal schauen wir kurz auf den Zusammenhang zwischen Lambda-Ausdrücken und deren Reduktion auf der einen Seite und Programmen und deren Ausführung auf der anderen.

3.4.1 Ausdrücke als Programme

Einen Lambda-Ausdruck $\lambda x.\, e$ kann man als Programm mit einer Eingabevariable x auffassen; dessen Ausgabe ist dann der Wert, zu dem der Rumpf e nach Substitution von x durch die Eingabe reduziert (ggf. in mehreren Schritten). Dabei kann e beliebig komplex sein und insbesondere andere Abstraktionen enthalten, die selbst wieder Eingaben verlangen. Somit lassen sich nicht nur mehrere Eingaben (per *Currying*; s. Abschnitt 3.3.9), sondern auch bedingte Eingaben realisieren, also solche, deren Erwartung von vorausgegangenen Eingaben abhängt (wegen einer darauf basierenden Fallunterscheidung, beispielsweise mittels der Funktion *ite* aus Abschnitt 3.3.10).

Wenn also ein Lambda-Ausdruck $\lambda x.\, e$ ein Programm mit einer Eingabevariable x ist, dann ist eine Anwendung $(\lambda x.\, e)\ e'$ ein Programmaufruf mit Eingabe e'. Eine Besonderheit des Lambda-Kalküls als Programmiersprache höherer Ordnung ist dabei, dass die Eingabe e' selbst ein Programm sein kann. Der Lambda-Kalkül eignet sich also prinzipiell auch für die *Metaprogrammierung*.

3.4.2 Operationale Semantik

Wir hatten in Abschnitt 3.3.5 festgestellt, dass die β-Reduktion an jeder beliebigen Stelle (jedem beliebigen Teilausdruck) eines Ausdrucks ansetzen kann. Einem *Interpreter* einer Programmiersprache würde man so viel Wahlfreiheit jedoch in der Regel nicht einräumen. Auch ist die *Konfluenz* eine recht fragile Eigenschaft und es ist nicht gesagt, dass sie bei Erweiterungen des Lambda-Kalküls zu einer brauchbaren Programmiersprache erhalten bleibt. Die Festlegung der Reihenfolge der Reduktionen ist spätestens dann notwendiger Teil der Semantik der Sprache.

Wir wollen also festlegen, in welcher Reihenfolge die Teilausdrücke eines Programms in der Sprache des Lambda-Kalküls reduziert werden. Dazu können wir die β-Reduktion aus Abschnitt 3.3.5 (die α-Konversion lassen wir hier implizit) mit dem aus Abschnitt 3.3.4 bekannten System der Ableitung, das Pattern matching verwendet, zu einem Regelsystem kombinieren, das für jeden Ausdruck des Lambda-Kalküls die nächste mögliche Reduktion (falls vorhanden) und somit die Reihenfolge der Reduktionen bis hin zu einer *Normalform* eindeutig bestimmt. Zugleich definiert dieses Regelsystem, wiederum in Analogie zu Abschnitt 3.3.4, induktiv eine Menge, nämlich die (unendliche) Menge aller möglichen Reduktionsschritte und somit, da jeder Reduktionsschritt aus zwei Ausdrücken des Lambda-Kalküls besteht, eine Relation (als Teilmenge von $\Lambda \times \Lambda$), nämlich die Reduktionsrelation, die wir hier, da sie nicht nur aus β-Reduktionen besteht, mit $\longrightarrow$ bezeichnen. Wenn wir also ab jetzt $e_1 \longrightarrow e_2$ schreiben, dann heißt das, dass das Paar (e_1, e_2) Element der Reduktionsrelation $\longrightarrow$ ist (analog zu $e \in \Lambda$ in Abschnitt 3.3.4).

Wie sehen nun die Regeln aus, die die schrittweise Reduktion eines Ausdrucks (und damit seinen Wert, falls vorhanden) festlegen? Wir beginnen zunächst mit dem Basisfall, der schon bekannten β-Reduktion, der durch das Axiom

$$(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1$$

dargestellt wird. Dass es sich hierbei um ein Axiom handeln soll und nicht um eine Regel, mag auf den ersten Blick überraschen, scheint doch $(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1$ selbst die Form einer Regel zu haben, nämlich die der β-Reduktion. Außerdem ist an die Anwendung des Axioms selbst eine Bedingung gestellt, nämlich die, dass der Ausdruck, der reduziert werden soll, die Form $(\lambda x.\, e_1)\, e_2$ hat. Stattdessen begreifen wir hier aber Ausdrücke der Form $e_1 \longrightarrow e_2$ selbst als syntaktische Muster, die wir mithilfe des *Pattern matching* (Abschnitt 3.3.4) gegeneinander abgleichen und dabei ihre Metavariablen instanziieren können (wobei hier jeweils zwei Ausdrücke des Lambda-Kalküls gematcht werden: die Quelle und das Ziel der Reduktion). Wenn wir also wissen wollen, ob beispielsweise $(\lambda y.\, y)\, z \longrightarrow z$ ein Element von $\longrightarrow$ ist, dann matchen wir es mit $(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1$ und erhalten, nach geeigneter Instanziierung der Metavariablen x, e_1 und e_2 , $(\lambda y.\, y)\, z \longrightarrow [z/y]y \equiv (\lambda y.\, y)\, z \longrightarrow$ z und somit eine positive Antwort. Wenn wir hingegen wissen wollen, zu welchem Ausdruck $(\lambda y.\, y)\, z$ reduziert, dann setzen wir für den Zielausdruck einfach eine (neue) Metavariable E ein und erhalten so (wiederum durch geeignete Instanziierung der Metavariablen, einschließlich E) $E = z$.

Da die Anwendung der Inferenzregeln auf Pattern matching über vollständige Ausdrücke basiert, reicht das obige Axiom nicht aus, um Teilausdrücke, die sich gemäß Abschnitt 3.3.5 per β-Reduktion reduzieren lassen, zu reduzieren. Dies wäre beispielsweise beim Ausdruck $\big((\lambda x.\, x)\ (\lambda x.\, x)\big)\ (\lambda x.\, x)$ notwendig, der zwar reduzierbar ist, aber nicht selbst die für die Anwendung des obigen Axioms notwendige Form hat. Wir führen daher zwei weitere, sog. Kongruenzregeln ein:

$$\frac{e_1 \longrightarrow E}{e_1\ e_2 \longrightarrow E\ e_2} \qquad \frac{e_2 \longrightarrow E}{(\lambda x.\, e_1)\ e_2 \longrightarrow (\lambda x.\, e_1)\ E}$$

Die linke dieser beiden Regeln besagt, dass wenn eine (beliebiger) Ausdruck e_1, der auf einen (wiederum beliebigen) Ausdruck e_2 angewendet wird, selbst zu einem Ausdruck E (repräsentiert durch die frische, im Quellausdruck nicht enthaltene Metavariable E) reduziert, dass dann dieses Redukt E auf e_2 angewendet das Ergebnis (oder Redukt) der durch die Regel definierten Reduktion ist. Man beachte, dass diese Regel selbst keine β-Reduktion darstellt; die Prämisse der Regel muss aber (ggf. nach mehreren Ableitungsschritten) auf einer β-Reduktion fußen, da diese den (einzigen) Basisfall der Induktion stellt. Diese Regel deckt somit den Fall der β-Reduktion eines Teilausdrucks ab, wobei sie festlegt, dass diese Reduktion *vor* der Reduktion ihres Anwendungsfalls, der Reduktion von $e_1\ e_2$, erfolgen muss.

Die rechte der beiden obigen Regeln gilt für den (allgemeinen) Fall, in dem der Ausdruck, der auf ein Argument e_2 angewendet wird, eine Lambda-Abstraktion ist, e_2 aber selbst noch weiter reduziert werden kann. Analog zur linken Regel wird damit festgelegt, dass zunächst e_2 reduziert werden soll; das Redukt E wird dann zum neuen Argument der Anwendung der Lambda-Abstraktion, die durch diese Regel abgedeckt wird. Dies entspricht einer Reduktionsreihenfolge, in der stets zuerst das Argument einer Funktionsanwendung reduziert wird, bevor die Anwendung selbst reduziert wird, die also genau der Strategie des *Call-by-value* entspricht (s. Abschnitt 3.3.5; man beachte wiederum, dass dieser Begriff im Kontext der funktionalen Programmierung anders zu verstehen ist als im Kontext der imperativen Programmierung; vgl. dazu Abschnitt 2.1.6). Auch hier muss die Reduktion eines Teilausdrucks vor der des ganzen Ausdrucks erfolgen, auch hier wird somit eine Reihenfolge festgelegt.

Die drei obigen Regeln zusammen erlauben übrigens immer noch nicht dieselben Reduktionen wie die β-Reduktion an beliebiger Stelle angesetzt: Es ist nämlich für ein Redukt, das die Form eines Werts hat (also $\lambda x.\, e$), keine weitere Regel mehr anwendbar, selbst wenn e reduzierbar wäre: $\lambda x.\, (\lambda y.\, y)\ z$ ist durch $\longrightarrow$ wie oben definiert nicht weiter reduzierbar, obwohl der Teilausdruck $(\lambda y.\, y)\ z$ es wäre.

Die beiden Kongruenzregeln legen also die Reihenfolge der Reduktionen wie folgt fest: Es wird zuerst der Ausdruck, der an der Stelle der anzuwendenden Funktion steht, reduziert und dann das Argument.[26] Allerdings haben wir im Zusammenspiel mit der ersten Regel, der β-Reduktion, ein Problem: In allen Fällen, in denen ein zu reduzierender Ausdruck die Form $(\lambda x.\, e_1)\ e_2$

[26] Würde man die beiden Regeln zu einer mit zwei Prämissen zusammenlegen (was wir in einem anderen Kontext in Abschnitt 3.4.4 tun werden), dann ginge daraus keine Reihenfolge mehr hervor.

hat, können jetzt zwei Regeln zur Anwendung kommen, nämlich die erste und die letzte der drei. Da die erste aber nicht für Call-by-value, sondern für *Call-by-name* steht (s. wiederum Abschnitt 3.3.5) und Call-by-name eine andere Reduktionsreihenfolge festlegt als Call-by-value, haben wir mit den Regeln nicht erreicht, was wir eigentlich wollten. Dies lässt sich beheben, indem wir die Regel für die β-Reduktion auf den Anwendungsfall beschränken, in dem das Argument selbst ein Wert, also nicht weiter reduzierbar ist (was genau dem Geist von Call-by-value entspricht):

$$(\lambda x.\, e_1)\,(\lambda y.\, e_2) \longrightarrow [(\lambda y.\, e_2)/x]e_1$$

Allerdings verlieren wir damit die Anwendung von Abstraktionen auf freie Variablen (also den Fall $(\lambda y.\, y)\; z$) — diese ist nun nicht mehr reduzierbar. Da wir aber freie Variablen in der Programmierung eher als problematisch erachten, ist dies ein kleiner Preis.

vollständige Reduktion in kleinen Schritten

Mit den obigen Regeln ist also die Reduktionsrelation induktiv definiert.[27] Das Ergebnis dieser Definition, eine unendliche Menge von Paaren (ein Graph; vgl. Abschnitt 3.1) bildet, genau wie die β-Reduktion, einen Ausdruck auf einen anderen ab, in dem an genau einer Stelle ein Redex durch sein Redukt ersetzt ist. Um einen beliebig großen (geschachtelten) Ausdruck auf einen Wert zu reduzieren, sind, genau wie bei der β-Reduktion, mehrere, hintereinander ausgeführte Anwendungen der Reduktionsrelation notwendig. Jede solche Anwendung bezeichnet man auch als **Reduktionsschritt** und die Reduktion, die ja iterativ erfolgt, insgesamt als **schrittweise**. Da die Schritte zudem klein sind (bei jedem Schritt wird genau ein Redex reduziert), bezeichnet man diese Form der operationalen Semantik auch als **Small-Step (Stuctural) Operational Semantics**.

Der Umstand, dass die Reduktionsrelation induktiv definiert ist, gibt Anlass zu zwei Bemerkungen.

1. Da die Reduktionsrelation (wie die Sprache des Lambda-Kalküls selbst) unendlich groß ist, kann sie nicht in einer Tabelle o. ä. hinterlegt werden. Es stellt sich daher die Frage, wie man ihre Anwendung auf einen Ausdruck operationalisiert. Dazu drängt sich ihre induktive Definition auf: Man leitet einfach aus den Regeln ab, ob $e \longrightarrow E$ ein Element der Relation ist, und erhält mit der Instanziierung von E das Redukt von e, falls es eines gibt. Andernfalls misslingt der Ableitungsprozess, womit man weiß, dass e nicht reduzierbar ist. In Programmiersprachen, die über Pattern matching verfügen (wie ML, Haskell, Scala, F# oder Prolog), ist die Implementierung eines solchen Ableitungsprozesses, und damit einer operationalen Semantik (eines Interpreters; vgl. Abschnitt 1.5.2) des Lambda-Kalküls, nicht mehr als eine Fingerübung.
2. Die Sprache von *einem Reduktionsschritt* mag im Kontext der induktiven Definition und der Tatsache, dass die Ableitung eines solchen Reduktionsschritts in der Regel mehrere

[27] Man nennt diese Form der Induktion auch **strukturelle Induktion**, weil sie auf der Struktur (syntaktischen Formen) von Ausdrücken basiert. Deswegen nennt man die dadurch definierte operationale Semantik auch **strukturelle operationale Semantik**.

Induktionsschritte erfordert, verwirren. Der Begriff Reduktionsschritt, genau wie die Bezeichnung Small-Step, bezieht sich jedoch ausschließlich auf das Ergebnis der Ableitung (die Relation), nicht auf den Ableitungsprozess.

⟶ ist eine partielle Funktion

Man beachte, dass durch die Verwendung der letzten drei obigen Regeln sichergestellt ist, dass für jeden zu reduzierenden Ausdruck des Lambda-Kalküls *höchstens* eine Regel zur Anwendung kommen kann. Die Relation ⟶, die durch die Regeln definiert wird, ist somit eine *partielle Funktion*, die jedem Ausdruck höchstens ein Redukt zuordnet. Damit ist unser Ziel, die Reihenfolge der Reduktion eindeutig festzulegen, erreicht.

3.4.3 Laufzeitumgebungen anstelle von Substitution

Wie wir gesehen haben, werden im Lambda-Kalkül, genau wie in der Mathematik, bei der Funktionsanwendung Variablen durch Ausdrücke substituiert und damit Unbekanntes durch Bekanntes ersetzt. Während dieser rein textuelle (syntaktische) Vorgang der Berechnung des Werts einer Funktionsanwendung auf dem Papier entspricht, ist er für die Programmierung reichlich unpraktisch, weil sich der Programmtext dadurch zum Zweck seiner Ausführung (bei Funktionsanwendungen) selbst ändern müsste.

Um dies zu vermeiden, führt man sog. **Laufzeitumgebungen** (engl. runtime, evaluation oder execution environments) ein, in denen bei einer Funktionsanwendung entstehenden Belegungen von Variablen mit Werten hinterlegt werden und in denen die Variablenwerte bei Verwendung der Variablen im Rumpf der angewendeten Funktion nachgeschlagen werden können (das sog. *Lookup*). Dies entspricht in etwa der Speicherung der Werte von Variablen, wie wir sie aus der imperativen Programmierung kennen, mit dem wesentlichen Unterschied, dass die Variablenwerte in der funktionalen Programmierung unveränderlich sind — die Wertzuweisung an eine Variable erfolgt einmalig bei der Funktionsanwendung.[28]

Umgebungen als Abbildungen

Mathematisch wird eine Laufzeitumgebung Ξ zumeist als eine (endliche) Abbildung von Variablennamen x auf Variablenwerte v aufgefasst, die durch eine endliche Menge von Paaren $x \mapsto v$ dargestellt wird. Ein Lookup wird demnach als $\Xi(x)$ notiert. Um einer Laufzeitumgebung Ξ ein neues Name/Wert-Paar hinzuzufügen, wird eine Notation wie $\Xi, x \mapsto v$ verwendet, die Ξ erweitert (man sagt auch „aktualisiert“, obwohl Abbildungen natürlich keinen Zustand haben), wobei die so entstehende Abbildung durch die (rekursive) Funktionsgleichung

[28] Wäre dies nicht der Fall, wären Substitution und Speicherung von Variablenwerten in einer Laufzeitumgebung nicht zueinander funktional äquivalent. Genau diese Äquivalenz ist aber Grundlage der sog. *referentiellen Transparenz* der funktionalen Programmierung: Jedes Vorkommen einer Variable innerhalb ihres Geltungsbereichs ist ein Platzhalter für denselben Wert. Dies gilt als eine der größten Stärken der funktionalen Programmierung; vgl. dazu die Bemerkungen zur Problematik der imperativen Programmierung in Abschnitt 2.1.2.

$$(\Xi, x \mapsto v)(x') = \begin{cases} v & \text{falls } x' = x \\ \Xi(x') & \text{sonst} \end{cases}$$

definiert ist. Damit lassen sich alle Umgebungen aus der leeren Umgebung per Erweiterung konstruieren. Allerdings dürfen darin keine Variablennamen mehrfach vorkommen — wir müssen dafür Sorge tragen, dass gleich benannte Variablen vorher umbenannt werden (per *α-Konversion* oder *-Reduktion*; s. Abschnitt 3.3.8).

Umgebungen als Listen

Sofern Variablennamen darin mehrfach vorkommen dürfen, kann man sich eine Umgebung Ξ aber auch als eine Liste von Name/Wert-Paaren vorstellen, an deren einem Ende (dem rechten bei obiger Notation für die Erweiterung) neue Elemente angefügt werden können. Die Eindeutigkeit der Abbildung von einem Variablennamen auf einen Variablenwert bleibt dann dadurch erhalten, dass in dieser Liste die Werte für Namen von dem Ende her nachgeschlagen werden, an dem neue Namen/Wert-Paare angehängt werden. So wird garantiert, dass immer die zuletzt zu einer Laufzeitumgebung hinzugefügte Variable mit dem gesuchten Namen gefunden wird, was den Regeln geschachtelter Geltungsbereich*e* (Scopes) entspricht und zugleich das Name capture oder Hiding (Abschnitt 3.3.8) nachbildet.

Durch die Einführung einer Laufzeitumgebung Ξ können wir uns nun die Substitution ersparen: Anstatt bei der Anwendung einer Lambda-Abstraktion $\lambda x.\, e$ alle Vorkommen der Variable x im Rumpf e auf einmal durch das Argument der Anwendung zu ersetzen, machen wir Variablen selbst reduzierbar und lassen sie bei der Anwendung zunächst stehen. Wenn ein Vorkommen von x dann im Verlauf der Reduktion von e selbst reduziert werden soll, schlagen wir den Inhalt der Variable in der Laufzeitumgebung nach. Dazu muss aber die Laufzeitumgebung bei der Reduktion immer mitgeführt werden, so dass die Reduktionsrelation $\longrightarrow$ zu einer Abbildung von Paaren mit der allgemeinen Form $\langle e \mid \Xi \rangle \longrightarrow \langle e' \mid \Xi' \rangle$ erweitert werden muss. Für die Funktionsanwendung erhalten wir so das Axiom

$$\langle (\lambda x.\, e_1)\ (\lambda x.\, e_2) \mid \Xi \rangle \longrightarrow \langle e_1 \mid \Xi, x \mapsto (\lambda x.\, e_2) \rangle$$

d. h., die Substitution von x in e_1 durch $\lambda x.\, e_2$ fällt wie angekündigt zugunsten einer Erweiterung der Laufzeitumgebung Ξ um $x \mapsto \lambda x.\, e_2$ weg, so dass die nun in e_1 *freie Variable* x durch die Erweiterung der Laufzeitumgebung gebunden[29] wird. Um diese Bindung bei der Auswertung von Ausdrücken auch wirksam werden zu lassen, brauchen wir noch eine Regel für das Nachschlagen der Werte von Variablen in der Laufzeitumgebung:

$$\frac{\Xi(x) = e}{\langle x \mid \Xi \rangle \longrightarrow \langle e \mid \Xi \rangle}$$

Ist eine Variable nicht in der Umgebung enthalten (war die Variable also frei), ist die Regel nicht anwendbar und die Auswertung bleibt stecken. Dies war allerdings auch schon bei der

[29] Hier haben wir nun den Fall, dass unter Bindung einer Variable auch die Zuordnung eines Werts verstanden wird.

Anwendung auf freie Variablen unter operationale Semantik mit Substitution der Fall (s. Abschnitt 3.4.2). Man beachte, dass freie Variablen nicht fälschlich durch Namensvetter in der Laufzeitumgebung gebunden werden können, da wir vorausgesetzt haben, dass sich alle Variablen namentlich unterscheiden.

Die beiden Kongruenzregeln müssen ebenfalls erweitert werden:

$$\frac{\langle e_1 \mid \Xi\rangle \longrightarrow \langle E \mid \Xi'\rangle}{\langle e_1\ e_2 \mid \Xi\rangle \longrightarrow \langle E\ e_2 \mid \Xi'\rangle} \qquad \frac{\langle e_2 \mid \Xi\rangle \longrightarrow \langle E \mid \Xi'\rangle}{\langle (\lambda x.\, e_1)\ e_2 \mid \Xi\rangle \longrightarrow \langle (\lambda x.\, e_1)\ E \mid \Xi'\rangle}$$

Dabei wird berücksichtigt, dass auch die Reduktion, die die Regelprämisse darstellt, die Laufzeitumgebung erweitern kann. So gilt z. B. für zwei (nicht weiter interessierende) Abstraktionen a und b als Argumente

$$\begin{aligned}&\langle (\lambda x.\, \lambda y.\, x)\ a\ b \mid \Xi\rangle \longrightarrow \langle (\lambda y.\, x)\ b \mid \Xi, x \mapsto a\rangle \longrightarrow\\ &\langle x \mid \Xi, x \mapsto a, y \mapsto b\rangle \longrightarrow \langle a \mid \Xi, x \mapsto a, y \mapsto b\rangle\end{aligned}$$

Das Ziel einer (wiederholten) Reduktion ist auch bei Verwendung einer Laufzeitumgebung anstelle der Substitution stets eine Abstraktion (die einzigen Werte des puren Lambda-Kalküls), mit dem Unterschied, dass diese Abstraktion nunmehr freie Variablen enthalten kann (wie $\lambda y.\, x$ nach dem ersten Reduktionsschritt aus dem obigen Beispiel, abgeleitet über die linke Kongruenzregel), deren Werte jedoch in der Umgebung hinterlegt (und dadurch gebunden) sind. So kann denn auch beispielsweise für den Ausdruck $(\lambda x.\, \lambda y.\, x)\ a$, einer Vereinfachung des obigen Beispiels, nur noch ein Paar $\langle \lambda y.\, x \mid \Xi\rangle$ mit $\Xi(x) = a$ das Ergebnis einer Reduktion sein, wo vorher (bei Verwendung der Substitution) $\lambda y.\, a$ das Ergebnis gewesen wäre. Die Umgebung ist also jetzt stets Teil des Ergebnisses einer Berechnung. Dies wird später noch eine wichtige Rolle spielen.

3.4.4 Natürliche Semantik

Genau wie die Substitution als Teil der Reduktion verlangt auch die Reduktion selbst ein fortwährendes Umschreiben des Programms — die Einführung einer Laufzeitumgebung allein kann dies nicht abstellen. Für die Umsetzung auf einem Computer ist das aber immer noch absolut unpraktisch: Hier möchte man, genau wie in der imperativen Programmierung, zwischen dem Programm und seiner Ausführung unterscheiden und dabei insbesondere das eine als statisch und das andere als dynamisch betrachten. Das bedeutet jedoch, dass wir einen Ausdruck nicht mehr schrittweise umschreiben können, bis das Ergebnis einen Wert darstellt, sondern dass die Überführung von einem Ausdruck in einen Wert direkt, quasi in einem großen Schritt, erfolgen muss. Wir nennen diesen Vorgang **Auswertung** (engl. evaluation) und die dazugehörige Abbildung **Auswertungsrelation** und definieren sie, wie die Reduktionsrelation in den Abschnitten 3.4.2 und 3.4.3, induktiv.

Funktionsabschluss Da wir die Substitution nicht wieder einführen wollen, setzen wir bei der Definition der Auswertungsrelation von Anfang an auf eine Laufzeitumgebung Ξ für das Fest-

halten von Variablenwerten, die sich aus Funktionsanwendungen (dem Einsetzen der Funktionsarguments in die Funktionsparameter) ergeben. Daraus folgt, dass das Ergebnis der Auswertung, ein Wert, nicht aus einer Abstraktion allein bestehen kann; vielmehr ist ein Wert ein Paar $\langle \lambda x.\, e \mid \Xi \rangle$, wobei Ξ die Belegung der in e auftretenden freien Variablen mit Werten enthält. Ein solches Paar nennt man auch **Funktionsabschluss** oder nur **Abschluss** (engl. closure).

Auswertung im Kontext einer Laufzeitumgebung

Zwar wollen wir mit der Auswertungsrelation jedem Ausdruck seinen Wert direkt zuordnen, aber die Ableitung der Abbildung eines Ausdrucks auf seinen Wert verlangt trotzdem mehrere Ableitungsschritte, nämlich wenn der Ausdruck Teilausdrücke umfasst, deren Wert zuerst bestimmt werden muss, bevor der Wert des gesamten Ausdrucks bestimmt werden kann. Operational ist die Auswertung also (genau wie die Reduktion) rekursiv angelegt. Dies nutzen wir aus, um den Mangel der fehlenden Berücksichtigung von Geltungsbereichen der Abstraktionsvariablen (Parameter) gleich mit abzustellen, indem wir die Auswertung selbst (und nicht nur die Werte) in den Kontext einer Laufzeitumgebung stellen und diese Laufzeitumgebung für die Unterauswertung eines Funktionsrumpfes — und nur für diese — um die Abstraktionsvariable erweitern. Dazu machen wir die Auswertungsrelation dreistellig: Sie bildet eine Laufzeitumgebung Ξ und einen Ausdruck e auf einen Wert v (für engl. value) ab, der wie oben beschrieben ein Abschluss ist, (v ist eine neue Metavariable, die für solche Werte steht). Wir schreiben im folgenden $\Xi \vdash e \Downarrow v$ für ein Element dieser Relation, wobei $\vdash$, das sog. Turnstile-Symbol (weil es manche wohl an ein Drehkreuz erinnert) hier wie „unter der Annahme von“ oder „gegeben“ gelesen wird, also $\Xi \vdash e \Downarrow v$ als „unter der Annahme von Ξ wertet e zu v aus“.[30] Man beachte, dass im Gegensatz zur Reduktionsrelation das Ergebnis der Anwendung der Auswertungsrelation auf einen Ausdruck e stets ein Wert v und nicht ein beliebiger Ausdruck e' ist.

Wir sind also nun bereit, die Auswertungsrelation durch die Angabe von Ableitungsregeln zu definieren. Wir beginnen mit der Auswertung von Abstraktionen zu Funktionsabschlüssen, die axiomatisch ist:

$$\Xi \vdash \lambda x.\, e \Downarrow \langle \lambda x.\, e \mid \Xi \rangle$$

In einer gegebenen Umgebung Ξ wertet also eine Funktion $\lambda x.\, e$ zu ihrem Abschluss unter Ξ aus.

Während der Funktionsabschluss die Laufzeitumgebung nur kopiert, wird sie durch die Funktionsanwendung erweitert. Dies zeigt sich in der Regel

$$\frac{\Xi \vdash e_1 \Downarrow \langle \lambda x.\, e_3 \mid \Xi' \rangle \qquad \Xi \vdash e_2 \Downarrow v_2 \qquad \Xi', x \mapsto v_2 \vdash e_3 \Downarrow v_3}{\Xi \vdash\ e_1\ e_2 \Downarrow v_3}$$

[30] Die Verwendung von $\vdash$ als erster Teil eines dreistelligen Infix-Operators $\_ \vdash \_ \Downarrow \_$ (zur Darstellung von Elementen einer dreistelligen Relation) mag Lesern mit Vorbildung in mathematischer Logik irritieren: Dort ist nämlich $\vdash$ ein häufig genutzter zweistelliger Operator, der für Ableitung steht (also etwa $A \vdash B$ für „B lässt sich aus A ableiten“). Wir verwenden hier aber den Bruchstrich für die Ableitung.

Die Anwendung eines Ausdrucks e_1 auf einen Ausdruck e_2 wertet gemäß dieser Regel zu dem Wert v_3 aus, der sich aus der Auswertung des Rumpfs e_3 der Funktion, zu der e_1 auswertet, ergibt, und zwar unter der Annahme, dass x den Wert v_2 hat, zu dem e_2 auswertet.[31] Die Regel legt das Auswertungsverfahren also auf *Call-by-value* fest. Man beachte, dass der Wert v_2, der für x in Ξ eingetragen wird, selbst ein Abschluss ist, also selbst eine Umgebung enthält.

Ein weiterer wesentlicher Unterschied zur Reduktionsrelation mit Umgebungen aus Abschnitt 3.4.3 ist, dass hier die Bindung von x (an den Wert v_2) ausschließlich für den Ausdruck e_3 (den Rumpf der angewendeten Funktion e_1) gültig ist — die Umgebung wird damit *lokal* für diesen Ausdruck und nicht, wie zuvor, von an die Geschwister von Teilausdrücken weitergereicht. Wenn wir nun auch noch Ξ als Liste (oder Stapel, und nicht wie zuvor als Menge) von Paaren erachten, mit der Notation $\Xi, x \mapsto v$ die Liste um ein Element verlängern und beim Lookup $\Xi(x)$ die Liste von hinten her durchsuchen (last in, first out), dann haben wir damit genau die geschachtelten Geltungsbereiche von Variablennamen im Lambda-Kalkül nachgebildet (und erlauben somit auch, dass mehrere Variablenbindungen sich einen Namen teilen).

Die Auswertung des Vorkommens einer Variable x in einem Ausdruck erfolgt schließlich, analog zur Reduktionssemantik, durch Nachschlagen des Variablenwerts in der Laufzeitumgebung:

$$\frac{\Xi(x) = v}{\Xi \vdash x \Downarrow v}$$

Eine Variable wertet also zu dem Wert aus, der in der Umgebung für sie gespeichert ist. Ist eine Variable nicht in der Umgebung enthalten, ist die Regel auch hier nicht anwendbar und die Auswertung bleibt stecken. Man beachte, dass auch hier der Wert ein Abschluss ist. Das ist insofern bemerkenswert, als Laufzeitumgebungen andere Laufzeitumgebungen enthalten können, nämlich die, die Bestandteil eines Abschlusses sind.

Wenn wir diese Auswerteregeln auf das Beispiel $(\lambda x.\, \lambda y.\, x)\ a\ b$ des vorigen Abschnitts anwenden, dann ergibt sich für den darin enthaltenen Teilausdruck $(\lambda x.\, \lambda y.\, x)\ a$ die Ableitung

$$\frac{\Xi \vdash \lambda x.\, \lambda y.\, x \Downarrow \langle \lambda x.\, \lambda y.\, x \mid \Xi \rangle \quad \Xi \vdash a \Downarrow \langle a \mid \Xi \rangle \quad \Xi \vdash \lambda y.\, x \Downarrow \langle \lambda y.\, x \mid \Xi, x \mapsto a \rangle}{\Xi \vdash (\lambda x.\, \lambda y.\, x)\ a \Downarrow \langle \lambda y.\, x \mid \Xi, x \mapsto a \rangle}$$

und somit nach Einsetzung des Ergebnisses in die Prämisse der Ableitung des Werts des Gesamtausdrucks

$$\frac{\Xi \vdash (\lambda x.\, \lambda y.\, x)\ a \Downarrow \langle \lambda y.\, x \mid \Xi, x \mapsto a \rangle \quad \Xi \vdash b \Downarrow \langle b \mid \Xi \rangle \quad \Xi, x \mapsto a, y \mapsto b \vdash x \Downarrow \langle a \mid \Xi, \dots \rangle}{\Xi \vdash (\lambda x.\, \lambda y.\, x)\ a\ b \Downarrow \langle a \mid \Xi, x \mapsto a, y \mapsto b \rangle}$$

[31] Man beachte, dass die Auswertung einer Funktionsanwendung hier, und anders als in den Reduktionssemantiken der Abschnitte 3.4.2 und 3.4.3, die Auswertung mehrerer Teilausdrücke zur Voraussetzung hat (die Prämissen über dem Strich sind implizit konjugiert, also durch ein logisches Und verbunden).

also das gleiche Ergebnis.

Big-step (strukturelle) operationale Semantik

Diese Form der Semantik, die häufig „natürlich“ genannt wird, ist ebenfalls syntaxgetrieben und wird deswegen, wie die Small-step-Semantik, als *strukturell* bezeichnet. In Abgrenzung von der Reduktionssemantik bezeichnet man sie als **Big-step (strukturelle) operationale Semantik**, weil die Auswertungsrelation jeden Ausdruck mit einer Anwendung auf einen Wert abbildet (und nicht wie die Reduktionsrelation in mehreren Iterationen der Anwendung). Beide sind jedoch auf ähnliche Weise induktiv (und damit rekursiv) definiert.

Basis für Interpreter

Die Big-step-Semantik des Lambda-Kalküls (sowie einer Sprache allgemein) eignet sich im Prinzip sehr gut als Vorlage für die Implementierung eines Interpreters für diese Sprache. Im Gegensatz zur Small-step-Semantik kann in der Big-step-Semantik in der Prämisse einer Regel allerdings die Auswertung mehrerer Teilausdrücke vorausgesetzt werden (wie an der Regel zur Funktionsanwendung zu sehen). Damit ist aber die Reihenfolge der Auswertung dieser Teilausdrücke nicht mehr festgelegt, was insbesondere in Sprachen mit Seiteneffekten (Wertzuweisung an globale Variablen) ein Problem sein kann. Daher ist mit einem Interpreter häufig auch eine Interpretation der Regeln vorgesehen, die eine bestimmte Reihenfolge der Auswertung der Prämissen vorsieht (in der Regel von links nach rechts, so wie es beispielsweise auch Prolog als Regelinterpreter macht). Damit verlässt man aber das Gebiet der klassischen Logik, in der die Prämissen gleichwertig nebeneinanderstehen.

3.4.5 Variablen, Let-Ausdrücke und benannte Funktionen

Wie wir gesehen haben, wird bei Verwendung einer Laufzeitumgebung anstelle der Substitution im Zuge einer Funktionsanwendung der Name der Abstraktionsvariable (des Parameters der Funktion) und deren Wert (das Argument bei der Funktionsanwendung) in der Laufzeitumgebung eingetragen und danach der Rumpf der Funktion ausgewertet. Hätten wir, ähnlich wie in der imperativen Programmierung, im Lambda-Kalkül Variablen und Wertzuweisung (Abschnitt 2.1.2), könnten wir eine Funktionsanwendung $(\lambda x.\, e')\ e$ auch wie folgt interpretieren: Wir weisen erst der Variable x den Wert von e zu und berechnen dann den Wert von e' (wobei x in e' vorkommen kann). Diese Interpretation nahelegend hat sich auch die syntaktische Form

$$\text{let } x = e \text{ in } e'$$

als Alternative zu $(\lambda x.\, e')\ e$ eingebürgert, die durch Anwendung der Regel

$$\frac{\Xi \vdash e \Downarrow v \qquad \Xi, x \mapsto v \vdash e' \Downarrow v'}{\Xi \vdash \text{let } x = e \text{ in } e' \Downarrow v'}$$

ausgewertet wird und die zu obiger Funktionsanwendung äquivalent (und somit lediglich *syntaktischer Zucker* dafür) ist. Man beachte, dass hier, anders als bei einer imperativen Wertzuweisung, die Gültigkeit von x auf den durch den In-Teil gekennzeichneten Ausdruck e begrenzt ist. Gegenüber der Funktionsanwendung hat der Let-Ausdruck den Vorteil, dass die Variable x

und ihr Wert, der Wert von e, unmittelbar nebeneinander stehen. Dies zahlt sich nicht nur aus, wenn e' lang ist, sondern entspricht auch dem imperativen Pendent `{let x = e; e'}` (hier in JavaScript-Syntax). Tatsächlich ist die Sperrung[32] von x und dem für x einzusetzenden Wert, dem von e, durch den Funktionsrumpf e' ja etwas, das durch die fehlende Benennung von Funktionen im Lambda-Kalkül überhaupt erst nötig wird: Hätte die anonyme Funktion in $(\lambda x.\, e')\ e$ beispielsweise den Namen „f“ und die Definition $f(x) = e'$, dann könnten wir einfach $f(e)$ schreiben. Interessanterweise lösen Let-Ausdrücke nicht nur die Sperrung von Variable und Argument auf, sondern erlauben zugleich die Benennung von Funktionen. So wird in dem Ausdruck

$$\text{let } f = \lambda x.\, x \text{ in } f\ v$$

zunächst die Variable f an den Abschluss der Funktion $\lambda x.\, x$ (die Identitätsfunktion) gebunden und die so benannte Funktion durch Verwendung ihres Namens dann auf den Wert v angewendet. Es entspricht der Let-Teil der Definition einer benannten Funktion f, deren Gültigkeitsbereich auf den In-Teil, hier den Ausdruck $f\ v$ (stellvertretend für irgendein Teilprogramm), beschränkt ist. Gleichwohl ist obiger Ausdruck doch nur eine alternative Schreibweise für $(\lambda f.\, f\ v)\ \lambda x.\, x$. (und der Name „f“ der Name einer Variable und nicht der der Funktion).

Let-Ausdrücke sind in der funktionalen Programmierung sehr gebräuchlich. Sie vereinen die Deklaration einer Variable mit ihrer Initialisierung. Dabei wird aus der Analogie zur Funktionsanwendung unmittelbar deutlich, dass, anders als in imperativen Programmiersprachen, der Wert einer Variable nach ihrer Initialisierung nicht mehr änderbar ist — es gibt in der (puren) funktionalen Programmierung keine Wertzuweisung, sondern nur Substitution und eine damit gleichgestellte Variablennutzung. Aber auch in nicht-puren funktionalen Programmiersprachen und, seit einiger Zeit, auch in vielen imperativen Programmiersprachen, gibt es die Möglichkeit, Variablen so auszuzeichnen (etwa durch das Schlüsselwort `const` oder `final`), dass ihnen nach ihrer Initialisierung keine Werte mehr zugewiesen werden können. Dies ermöglicht u. a. Compilern eine *Variablensubstitution zur Übersetzungszeit*.

3.4.6 Rekursion

Wie wir in 3.3.11 gesehen hatten, ist Rekursion im reinen Lambda-Kalkül zwar möglich, aber umständlich. Die Einführung eines Namens für eine Funktion mittels eines Let-Ausdrucks hilft hier nicht weiter, da die Gültigkeit des Namens auf den Ausdruck im In-Teil beschränkt ist, die Definition des Namens also auf den Namen selbst keinen Bezug nehmen kann.

letrec

Diese Beschränkung wird durch eine Definition mit letrec aufgehoben:

$$\textit{letrec}\ f = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1) \text{ in } f(5)$$

[32] Sperrung bezeichnet hier die Unterbrechung eigentlich zusammengehörender Wörter durch andere.

beispielsweise definiert die Fakultät ohne das ganze Brimborium mit dem Y-Kombinator. Auch wenn man solche Letrec-Ausdrücke als syntaktischen Zucker für Ausdrücke des reinen Lambda-Kalküls ansehen kann, werden Programmiersprachen diese direkt übersetzen.

3.4.7 Weitere Sprachelemente als abgeleitete Formen

Auf den ersten Blick unterscheidet sich die funktionale Programmierung von der imperativen u. a. dadurch, dass sie nur Ausdrücke und keine Anweisungen (Befehle) kennt. Insbesondere scheint es, wenn ein Programm nur aus einem (tief geschachtelten) Ausdruck besteht, keine *Anweisungsfolgen* zu geben. Das stimmt aber nur bedingt, wenn man bedenkt, dass wir mit der operationalen Semantik auch die *Auswertungsreihenfolge* von Teilausdrücken festgelegt haben. Und wenn man bedenkt, dass imperative Sprachen ja schon gezeigt haben, dass man Ausdrücke zu Anweisungen befördern kann, indem man einfach ihr Ergebnis verwirft (Abschnitt 2.1.3).

Sequenz

Tatsächlich lässt sich eine Folge von zwei Anweisungen s_1 und s_2 als ein Ausdruck darstellen, wenn wir eine Funktion

$$seq \equiv \lambda x.\, \lambda y.\, y$$

definieren: Dann würde der Ausdruck $seq\ s_1\ s_2$ (unter der Annahme von Call-by-value) dazu führen, dass erst s_1 (als Argument der Anwendung von $\lambda x.\, \lambda y.\, y$) ausgewertet wird und dann s_2 (als Argument der Anwendung des resultierenden $\lambda y.\, y$, das das Ergebnis der Auswertung von s_1 nicht verwendet, also verwirft). Längere „Befehlsfolgen“ würden dann durch entsprechend geschachtelte Anwendungen von *seq* entstehen. Wir können also die neue syntaktische Form $s_1; s_2$ einführen, ohne die Semantik des Lambda-Kalküls erweitern zu müssen — dafür betrachten wir sie einfach als *syntaktischen Zucker* für obige Funktionsanwendung.

Wie bereits in Abschnitt 3.4.5 angedeutet ist auch die Variablendeklaration relativ leicht als syntaktischer Zucker für den Lambda-Kalkül darzustellen: Ein Anweisungsblock `{var x=c; s}` lässt sich im Lambda-Kalkül als $(\lambda x.\, s)\ c$ ausdrücken, wobei s auch eine Anweisungsfolge sein kann. Zudem bieten Church-Codierungen wie die für *true* und *false* sowie für die natürlichen Zahlen die Grundlage für Verzweigungen und einfache Arithmetik (Abschnitt 3.3.10); andere Church-Codierungen (z. B. für Paare) sind die Grundlage weiterer, komplexerer Codierungen, so dass sich ganz ansehnliche funktionale Programmiersprachen direkt im Lambda-Kalkül codieren lassen.

3.4.8 Speicher und Referenzen

Eine Sache, die nicht ohne weiteres gelingt, ist die Codierung von veränderlichen Variablen, wie man sie aus der imperativen Programmierung sowie aus nicht-reinen funktionalen Programmiersprachen kennt. Dafür kann der Lambda-Kalkül jedoch um eine weitere Umgebungsvariable erweitern werden, die den Speicher repräsentiert und die die Reduktions- bzw. Auswertungsrelation erweitert. Der schreibende und lesende Zugriff auf diesen Speicher erfolgt dann über eine neue Form von Werten, Adressen genannt, die man sich wie Indizes in ein Array

(oder Hardwareadressen des Hauptspeichers) vorstellen kann. Der lesende und schreibende Zugriff auf den Speicher erfolgt dann über Dereferenzierung der Adresswerte, wofür der Lambda-Kalkül allerdings auch syntaktisch um entsprechende Elemente (Formen) erweitert werden muss. Man beachte, dass diese Umsetzung veränderlicher Variablen im Wesentlichen der Abbildung von Variablen in den Hauptspeicher durch den Compiler bei der imperativen Programmierung entspricht, wobei diese Programmiersprachen die Dereferenzierung der Adressen, die mit den Variablennamen verbunden sind, vorm Programmierer verborgen halten.

3.5 Das Wesen funktionaler Programmierung

zustandslos

In der reinen funktionalen Programmierung haben Variablen keine Speicherfunktion, sondern abstrahieren lediglich von konkreten Werten, durch die sie jederzeit vollständig ersetzt werden können (per *Substitution*). Insofern haben funktionale Programme keinen Zustand, der ja für die imperative Programmierung kennzeichnend ist (s. Abschnitt 2.3). So zumindest die Legende.

versteckter Zustand

Tatsächlich speichern Variablen aber auch in der reinen funktionalen Programmierung Werte, und zwar auf dem Stack, wenn man eine Laufzeitumgebung anstelle von Substitution verwendet, sowie andernfalls im sich durch β-Reduktion ständig ändernden Programmtext selbst. Insofern haben dann doch auch funktionale Programme einen Zustand. Dies merkt man spätestens dann, wenn man ein funktionales Programm debugt und sich zu diesem Zweck den Fortschritt des Programms (die Zustandswechsel) genauer anschaut.

eigene Terminologie

Wie es sich für ein eigenständiges Paradigma gehört, wird in der funktionalen Programmierung anders gesprochen als in der imperativen. So werden Funktionen nicht aufgerufen, sondern angewendet, sie geben nichts zurück, sondern werten zu etwas aus usw. Diese verschiedenen Sprechweisen stellen insbesondere für Lerner eine nicht zu unterschätzende Hürde dar, weil oft nicht klar ist, ob mit unterschiedlichen Begriffen vielleicht dasselbe gemeint ist oder mit gleichen verschiedenes. Die objekt-funktionale Programmierung insbesondere steht daher vor der Herausforderung, die Begriffswelten der imperativen (objektorientierten) und der funktionalen Programmierung zu harmonisieren.

Metaprogrammierung

In der funktionalen Programmierung sind Funktionen Werte und somit (Unter-)Programme und Daten zugleich. Dies ist eine wesentliche Voraussetzung für die sog. **Metaprogrammierung**, also der Form der Programmierung, bei der Programme Ein- und Ausgaben sind. Wenn man zusätzlich die Möglichkeit hat, Daten auszuführen (wie etwa im Lambda-Kalkül durch Anwendung eines Werts auf einen Ausdruck), dann lassen sich damit Interpreter schreiben oder aber der Vorrat an Kontrollsturen einer Programmiersprache, und somit die Programmiersprache selbst, erweitern. Zugleich lassen sich aber auch Teile eines Programms durch das Programm selbst konstruieren und ausführen. Beispiele hierfür werden uns in den nachfolgenden Lektionen begegnen.

Literaturhinweise

Für die imperative Programmierung ist Niklaus Wirths „Algorithmen und Datenstrukturen“ nach wie vor sehr lesenswert, insbesondere auch deswegen, weil es nie die Abbildung in den Speicher aus den Augen lässt. Aus derselben „goldenen“ Ära stammt auch „The C Programming Language“ von Brian Kernighan und Dennis Ritchie, was eine Art Gegenentwurf darstellt. Während C oft als „combining the expressiveness of assembly language with the flexibility of assembly language” beschrieben wird, so gewährt diese Sprache doch tiefere Einblicke in die Organisation und die Adressierung von Speicher (insbesondere dynamische Speicherverwaltung sowie Call-by-reference), die andere Programmiersprachen hinter „höheren“ Programmierkonzepten verbergen. Letztlich muss man die Mechanismen der Speicherorganisation und -adressierung aber sowieso kennen, nämlich spätestens dann, wenn man verstehen will, warum die höheren Programmierkonzepte im Einzelfall nicht so funktionieren, wie man eigentlich dachte.

Einführungen in die funktionale Programmierung über den Weg des Lambda-Kalküls gibt es viele. Ich selbst habe den Lambda-Kalkül mit Benjamin Pierces exzellentem Buch „Types and Programming Languages“ kennen und schätzen gelernt.