Programmierung – eine Einführung in die Informatik mit Standard ML

von
Prof. Dr. Gert Smolka

Oldenbourg Verlag München Wien

Gert Smolka ist seit 1990 Professor für Informatik an der Universität des Saarlandes, wo er in Saarbrücken den Lehrstuhl für Programmiersysteme leitet. Zusammen mit seinen Schülern hat er die Programmiersprachen Oz und Alice entwickelt. Informatik studiert hat Gert Smolka an der Universität Karlsruhe und an der Cornell University, promoviert hat er an der Universität Kaiserslautern. Im Zentrum seiner Forschungsarbeiten stehen Logik und Berechnung, mit Anwendungen bei Programmiersprachen, in der Künstlichen Intelligenz und in der Computerlinguistik.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Lektorat: Dr. Margit Roth
Herstellung: Anna Grosser
Coverentwurf: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Druckhaus „Thomas Müntzer“ GmbH, Bad Langensalza

ISBN 978-3-486-58601-5

Vorwort

Dieses Lehrbuch bietet eine Einführung in die Informatik, die Algorithmen, Datenstrukturen und den Aufbau von Programmiersprachen behandelt. Es ist zusammen mit einer neu konzipierten Anfängervorlesung entstanden, die der Autor an der Universität des Saarlandes entwickelt hat. Die Leitidee ist einfach: Programmierung soll so vermittelt werden, dass sie als gelungene Synthese von Theorie und Praxis erscheint. Programmierkenntnisse sollen nicht vorausgesetzt werden, aber praktische Programmierfertigkeiten sollen ab der ersten Vorlesungswoche trainiert werden.

Das Buch behandelt rekursive Datenstrukturen, Korrektheitsbeweise und Laufzeitbestimmungen. Die Syntax und Semantik von Programmiersprachen wird mit Grammatiken und Inferenzregeln beschrieben. Auf dieser Grundlage werden Parser, Interpreter und Übersetzer programmiert. Speicheroperationen und veränderliche Datenstrukturen werden ausführlich behandelt. Schließlich werden maschinennahe Strukturen mithilfe einer virtuellen Maschine vermittelt.

Dieses Buch ist keine Einführung in die Feinheiten einer Programmiersprache. Stattdessen vermittelt es die eher mathematischen Aspekte der Programmierung und schafft eine Grundlage, auf der die üblichen Programmiersprachen schnell erlernt werden können. Es verwendet die theorienahe Sprache Standard ML, um auch anspruchsvolle Programme einfach realisieren zu können. Dank der für diese Sprache verfügbaren Interpreter ist der Einstieg in das praktische Programmieren auch für Anfänger einfach.

Das Buch ist in 16 Kapitel gegliedert. Jedes Kapitel kann in etwa einer Vorlesungswoche behandelt werden. Kapitel 2 und 8 können zügiger behandelt werden, da Teile des Materials bei Bedarf später nachgelesen werden können.

Kapitel 1-3 führen die Grundbausteine der funktionalen Programmierung ein. Zunächst beschreiben wir Funktionen für Zahlen mit Rekursionsgleichungen und realisieren sie dann durch Prozeduren. Dabei lernen wir endrekursive Prozeduren mit Akkus kennen. Danach erläutern wir den syntaktischen und semantischen Aufbau von Programmiersprachen. Schließlich formulieren wir die Rekursionsschemen für bestimmte und unbestimmte Iteration mithilfe von höherstufigen Prozeduren.

In Kapitel 4 und 5 geht es um das Programmieren mit linear-rekursiven Listen. Neben Sortieralgorithmen (Einfügen, Mischen, Quicksort) entwickeln wir eine Prozedur für die Primzahlzerlegung. Dabei lernen wir den Begriff der Invariante kennen. Außerdem beschäftigen wir uns mit der Darstellung ganzzahliger Mengen.

In Kapitel 6 und 7 geht es um das Programmieren mit baumrekursiven Strukturen. Dabei

lernen wir Konstruktortypen und das Werfen und Fangen von Ausnahmen kennen. Wir zeigen, wie man arithmetische Ausdrücke darstellt und auswertet. Danach arbeiten wir mit einem allgemeinen Baummodell, das die primäre Baumrekursion mit einer Rekursion für Unterbaumlisten kombiniert. Schließlich beschäftigen wir uns mit der Darstellung finitärer Mengen durch gerichtete Bäume.

Die Kapitel 8-11 haben einen mathematischen Charakter. Zunächst behandeln wir Mengen, Tupel, Graphen, Relationen, Funktionen und Terminierung. Dann betrachten wir Prozeduren als gedankliche Objekte, die unabhängig von einer konkreten Programmiersprache existieren. Dabei geht es uns um den Beweis von Korrektheitseigenschaften und die Bestimmung von Laufzeiten. Induktive Beweise führen wir als rekursive Beweise ein. Für die Bestimmung von Laufzeiten verwenden wir Rekurrenzsätze.

In Kapitel 12 und 13 geht es um die Syntax und Semantik von Programmiersprachen. Wir beginnen mit der abstrakten Syntax. Die statische und dynamische Semantik beschreiben wir mit Inferenzregeln, die abstrakte und konkrete Syntax mit Grammatiken. Mithilfe dieser Beschreibungen realisieren wir Lexer, Parser (rekursiver Abstieg), Elaborierer und Interpreter.

In Kapitel 14 geht es um die Realisierung von Datenstrukturen mit Strukturen, Signaturen und Funktoren. Dabei unterscheiden wir zwischen der abstrakten Benutzersicht und der konkreten Implementierung einer Datenstruktur. Wir lernen Vektoren und binäre Suche kennen. Außerdem behandeln wir eine effiziente Implementierung funktionaler Schlangen. Dabei begegnen wir dem Begriff der Darstellungsinvariante.

In Kapitel 15 erweitern wir unser Programmiermodell um Speicheroperationen, mit denen wir veränderliche Objekte realisieren können. Wir behandeln Reihungen, Stapel, Schlangen und Schleifen. Außerdem gehen wir auf die Darstellung von Listen und Bäumen in linearen Speichern ein. Damit sind wir in der Lage, Aussagen über den Platzbedarf bei der Programmausführung zu machen.

In Kapitel 16 behandeln wir zwei maschinennahe Sprachen W und M. W verfügt über imperative Variablen und Schleifen. M wird durch eine Stapelmaschine mit Halde realisiert. Die Programme von M sind Befehlsfolgen, die Konditionale und Schleifen durch Sprünge realisieren. Mit M erklären wir das maschinennahe Ausführungsmodell für rekursive Prozeduren. Schließlich entwickeln wir einen Übersetzer, der W nach M übersetzt.

Der Erwerb methodischen Wissens steht und fällt mit der Bearbeitung von Übungsaufgaben. Jedes Kapitel enthält eine Vielzahl von Aufgaben. Oft soll der Leser kleine Prozeduren schreiben und sie mit Beispielen erproben.

Webseite

Die Webseite dieses Buchs steht unter www.ps.uni-saarland.de/prog-buch/. Dort finden Sie Hinweise zu Interpretern und Online-Dokumenten für Standard ML, die Texte der größeren Programme dieses Buchs, Musterlösungen für ausgewählte Aufgaben und eine Fehlerliste. Folien finden Sie keine, da ich die Vorlesung überwiegend an der Tafel halte.

Danksagung

Ein gutes Lehrbuch zu schreiben ist mehr Arbeit als man denkt. Ohne die sehr positive Resonanz der Studierenden, die ihr Studium im Laufe der Jahre mit verschiedenen Versionen dieses Texts begonnen haben, hätte ich dieses Projekt kaum zum Abschluss bringen können.

Mein herzlicher Dank gilt allen Assistenten und Tutoren, die die Vorlesung im Lauf der Jahre tatkräftig unterstützt und Ideen beigetragen haben. Namentlich erwähnen will ich Thorsten Brunklaus, Moritz Hardt, Marco Kuhlmann, Niko Paltzer, Raphael Reischuk, Andreas Rossberg, Jan Schwinghammer und Guido Tack. Bedanken möchte ich mich auch bei meinen Kollegen Holger Hermanns, Andreas Podelski und Reinhard Wilhelm, die die Vorlesung nach Vorgängerversionen dieses Texts gehalten haben. Korrekturgelesen haben zu verschiedenen Zeitpunkten Marco Kuhlmann, Simon Pinkel, Julia Luxenburger, Matthias Höschele, Nikolai Knopp, Simon Moll, Kim Pecina, Raphael Reischuk und Daniel Wand. Schließlich gilt mein Dank Raphael Reischuk, mit dem ich die vorliegende Version des Buchs erstellt habe.

Widmung

Für Felix, Steffen und die liebe Frau Schlange.

Saarbrücken, September 2007 Gert Smolka

Vorwort zur zweiten Auflage

Für die zweite Auflage haben Raphael Reischuk und ich alle bekannten Fehler korrigiert. Außerdem wurden einige Formulierungen verbessert und eine Aufgabe hinzugefügt.

Saarbrücken, Mai 2011 Gert Smolka

Inhaltsverzeichnis

1 Schnellkurs

Wir beginnen mit Beispielen, die Sie mit grundlegenden Programmierkonstrukten und Programmiertechniken bekannt machen. Dabei lernen Sie, wie man kleine Programme schreibt und sie mit einem Interpreter ausführt.

1.1 Programme

Hier ist unser erstes Programm:

```
val x = 4*7+3
val y = x*(x-29)
```

Wie fast alle Programme, die wir in diesem Buch behandeln werden, ist es in der Programmiersprache Standard ML geschrieben. Es besteht aus zwei Deklarationen. Die erste deklariert den Bezeichner *x*, die zweite den Bezeichner *y*. Die Ausführung des Programms berechnet die Werte der Bezeichner *x* und *y*:

$x = 31$
$y = 62$

Bevor wir uns weitere Programme ansehen, wollen wir ein paar programmiersprachliche Begriffe einführen. Programme werden ähnlich wie Texte durch aufeinander folgende **Wörter** dargestellt. Das obige Programm ist mit vier Arten von Wörtern dargestellt:

Bezeichner	*x*, *y*
Konstanten	3, 4, 7, 29
Operatoren	+, −, ∗
Schlüsselwörter	*val*, =, (,)

Bezeichner dienen als Namen, die bei der Ausführung eines Programms an Werte gebunden werden. **Konstanten** sind Wörter, die bestimmte Werte bezeichnen. Beispielsweise bezeichnet die Konstante *7* die Zahl 7. **Operatoren** sind Wörter, die Operationen darstellen. Beispielsweise stellt der Operator ∗ die Multiplikationsoperation für Zahlen dar. **Schlüsselwörter** dienen dazu, den Aufbau eines Programms darzustellen.

Eine **Deklaration** hat die Form

val ⟨*Bezeichner*⟩ = ⟨*Ausdruck*⟩

und deklariert einen Bezeichner, der in den nachfolgenden Deklarationen eines Programms benutzt werden kann. Bei der Ausführung einer Deklaration wird der **deklarierte Bezeichner** an den Wert **gebunden**, den die Ausführung des Ausdrucks der Deklaration liefert.

Ausdrücke werden mit Konstanten, Operatoren, Bezeichnern und Klammern gebildet, wie wir am Beispiel des Ausdrucks $x * (x - 29)$ sehen können.

Ein **Programm** ist eine Folge von Deklarationen. Bei der Ausführung eines Programms werden seine Deklarationen der Reihe nach ausgeführt. Wenn wir die letzte Deklaration eines Programms streichen, bekommen wir ein kürzeres Programm. Umgekehrt können wir ein Programm durch Hinzufügen einer Deklaration verlängern.

Im Zusammenhang mit Programmen verstehen wir unter einem **Wert** ein Objekt, mit dem bei der Ausführung eines Programms gerechnet werden kann. Später werden wir verschiedene **Typen** von Werten kennenlernen. Vorerst benötigen wir nur Werte des Typs *int*. Dabei handelt es sich um ganze Zahlen.

Programmiersprachen orientieren sich sprachlich am Englischen. Beispielsweise sind die Wörter *val* und *int* aus englischen Wörtern abgeleitet:

value	Wert
integer	ganze Zahl

Schließlich erwähnen wir noch eine Besonderheit von Standard ML, über die Sie gelegentlich stolpern werden: Als Negationsoperator für Zahlen wird das Zeichen "~" verwendet. Sie müssen also "~7" schreiben, wenn Sie -7 meinen. Das normale Negationszeichen "$-$" dient als Subtraktionsoperator (z. B. $x - y$).

Aufgabe 1.1 Betrachten Sie das folgende Programm:

```
val x = 7+4
val y = x*(x-1)
val z = ~x*(y-2)
```

Welche Bezeichner, Konstanten, Operatoren und Schlüsselwörter kommen in dem Programm vor? An welche Werte bindet das Programm die vorkommenden Bezeichner?

1.2 Interpreter

Experimentieren ist ein wichtiger Teil des Programmierens. Durch gezielte Experimente können neue Ideen entwickelt und Fragen zu Programmen und zur Programmiersprache geklärt werden.

Ein **Interpreter** ist ein virtuelles Labor für das Experimentieren mit Programmen. Ab jetzt werden wir ständig mit einem Interpreter arbeiten. Auf der Webseite des Buches finden Sie Hinweise zu frei erhältlichen Interpretern für Standard ML.

Nachdem Sie einen Interpreter für Standard ML gestartet haben, können Sie in einem Interaktionsfenster Programme eingeben. Wir beginnen mit der Eingabe des Programms

```
val x = 4*7+3
```

Damit der Interpreter mit der Bearbeitung des Programms beginnt, muss nach dem Programm zunächst ein Semikolon ";" eingegeben und direkt danach die Eingabetaste (enter) gedrückt werden. Das Hilfszeichen ";" ist erforderlich, damit mehrere Zeilen am Stück übergeben werden können. Nach der Übergabe einer Eingabe prüft der Interpreter zunächst, ob es sich bei der Eingabe um ein gemäß den Regeln der Sprache zulässiges Programm handelt. Wenn dies wie in unserem Beispiel der Fall ist, führt er das Programm aus. Danach informiert er den Benutzer über das Ergebnis:

```
val x = 31 : int
```

Bei der Ausgabe der berechneten Werte für die deklarierten Bezeichner gibt der Interpreter auch die für die Bezeichner ermittelten Typen an. Vorerst arbeiten wir nur mit Bezeichnern des Typs *int*. Die Ausgaben des Interpreters stellen wir immer in *diesem Schriftsatz* dar. In Zukunft werden wir die Eingabe meistens zusammen mit der Ausgabe darstellen:

```
val x = 4*7+3
val x = 31 : int
```

Ein bereits eingegebenes Programm kann durch die Eingabe weiterer Deklarationen verlängert werden. Wir erweitern unser Programm um die Deklaration

```
val y = (x-29)*x
val y = 62 : int
```

Erwartungsgemäß berechnet der Interpreter den Wert von *y* durch Rückgriff auf den bereits berechneten Wert von *x*. Als Nächstes erweitern wir unser Programm um zwei Deklarationen, die wir zusammen eingeben:

```
val a = x-y
val b = x+y
val a = ~31 : int
val b = 93 : int
```

Wenn Sie wollen, können Sie die beiden Deklarationen auch in einer Zeile eingeben:

```
val a = x-y val b = x+y
val a = ~31 : int
val b = 93 : int
```

1.2.1 Mehrfachdeklaration

Bezeichner können mehrfach deklariert werden:

```
val x = 2
```
val x = 2 : int

```
val x = 3
```
val x = 3 : int

```
val y = x*x
```
val y = 9 : int

Bei der Benutzung eines mehrfach deklarierten Bezeichners wird immer der durch die zuletzt ausgeführte Deklaration ermittelte Wert verwendet. Wenn wir als Nächstes die Deklaration

```
val x = x*x
```
val x = 9 : int

eingeben, passiert Folgendes: Zuerst wird der Ausdruck $x * x$ mit der bereits existierenden Bindung $x = 3$ ausgewertet. Das liefert den Wert 9. Dann wird der Bezeichner x an den Wert 9 gebunden.

1.2.2 Ergebnisbezeichner

Um Ihnen Schreibarbeit zu ersparen, verwendet der Interpreter einen Trick: Deklarationen des sogenannten **Ergebnisbezeichners** *it* können ohne den Vorspann "*val it =*" eingegeben werden. Statt

```
val it = 4*7+3
```
val it = 31 : int

können Sie also kürzer

```
4*7+3
```
31 : int

eingeben. Das ist praktisch, wenn Sie zunächst nur den Wert eines Ausdrucks sehen wollen. Wenn Sie den Wert danach weiter benutzen wollen, können Sie auf ihn mithilfe des Ergebnisbezeichners zugreifen:

```
val x = it+it
```
val x = 62 : int

```
it+it
```
62 : int

```
it-60
```
2 : int

1.2.3 Fehlermeldungen

Ein Interpreter prüft für jede Eingabe, ob sie gemäß den Regeln der Sprache zulässig ist. Unzulässige Eingaben werden mit einer Fehlermeldung beantwortet. Hier ist ein Beispiel:

```
vall x = 4
! Toplevel input:
! vall x = 4;<EOF>
! ^^^^^^^^^
! Unbound value identifier: vall
```

Die Fehlermeldung besagt, dass das Wort *vall* vom Interpreter als ein ungebundener Bezeichner eingestuft wurde. Da die Fehlermeldungen des Interpreters auf Programmierer ausgerichtet sind, die sich mit Standard ML gut auskennen, werden die meisten Fehlermeldungen für Sie vorerst größtenteils unverständlich sein.

1.3 Prozeduren

Unter einer **Prozedur** verstehen wir eine Berechnungsvorschrift, die bei der **Anwendung** auf ein **Argument** ein **Ergebnis** liefert. Prozeduren werden durch Gleichungen beschrieben. Beispielsweise beschreibt die Gleichung

$$quadrat(x) = x \cdot x$$

eine Prozedur, die zu einer Zahl das Quadrat liefert. Diese Prozedur können wir in Standard ML mit der **Prozedurdeklaration**

```
fun quadrat (x:int) = x*x
val quadrat : int → int
```

an den Bezeichner *quadrat* binden. Der Bezeichner *quadrat* erhält dabei den Typ $int \to int$, was besagt, dass er an eine Prozedur gebunden ist, die auf Argumente des Typs *int* angewendet werden kann und Ergebnisse des Typs *int* liefert.

Ausdrücke der Bauart $quadrat(2+3)$ werden als **Prozeduranwendungen** bezeichnet. Sie bestehen aus zwei Teilausdrücken. Der erste Teilausdruck beschreibt die anzuwendende Prozedur, und der zweite den als Argument zu verwendenden Wert.

```
quadrat(2+3)
25 : int

quadrat 4
16 : int
```

Statt *quadrat 4* kann man auch $quadrat(4)$ schreiben. Eine Klammerung des Argumentausdrucks ist nur dann erforderlich, wenn dieser wie bei $quadrat(2+3)$ aus mehreren

Teilausdrücken besteht. Wenn man hier die Klammern weglässt, wird die Prozeduranwendung der Addition untergeordnet:

```
quadrat 2 + 3
```
7 : int

```
(quadrat 2) + 3
```
7 : int

Hier sind Beispiele für Ausdrücke mit zwei Prozeduranwendungen:

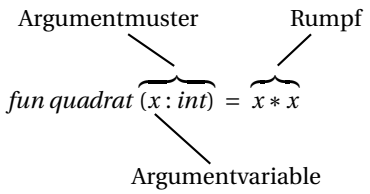
```
quadrat 2 + quadrat 3
```
13 : int

```
quadrat (2 + quadrat 3)
```
121 : int

```
quadrat (quadrat 3)
```
81 : int

Wie wir bereits festgestellt haben, hat die Prozedur *quadrat* den Typ $int \to int$. Allgemein besteht ein **Prozedurtyp** $t_1 \to t_2$ aus einem **Argumenttyp** t_1 und einem **Ergebnistyp** t_2. Eine Prozedur des Typs $t_1 \to t_2$ kann auf Argumente des Typs t_1 angewendet werden und liefert Ergebnisse des Typs t_2.

Am Beispiel der obigen Prozedurdeklaration wollen wir noch einige Sprechweisen für Prozeduren einführen:



Zu einer Prozedur gehört ein Argumentmuster und ein Rumpf. Das **Argumentmuster** bestimmt den Argumenttyp der Prozedur und führt einen Bezeichner ein, der das Argument darstellt und als **Argumentvariable** bezeichnet wird. Der **Rumpf** einer Prozedur ist ein Ausdruck, der beschreibt, wie das Ergebnis der Prozedur zu berechnen ist.

Bei einer Prozedur handelt es sich um eine Berechnungsvorschrift für eine Funktion. Das erklärt das einleitende Schlüsselwort *fun* bei Prozedurdeklarationen. Eine Funktion beschreibt eine Abbildung von Argumenten auf Ergebnisse, ohne sich dabei auf eine bestimmte Berechnungsvorschrift festzulegen. Daher kann ein und dieselbe Funktion durch verschiedene Prozeduren berechnet werden. Beispielsweise berechnet die Prozedur

```
fun quadrat' (y:int) = y*(y-1)+y
```
val quadrat' : int → int

dieselbe Funktion wie die Prozedur *quadrat*.

Berechnungsvorschriften bezeichnet man in der Informatik als **Algorithmen**. Bei Prozeduren handelt es sich dementsprechend um Algorithmen, die durch Gleichungen beschrieben sind und Funktionen berechnen.

Aufgabe 1.2 Deklarieren Sie eine Prozedur $p : int \to int$, die für x das Ergebnis $2x^2 - x$ liefert. Identifizieren Sie das Argumentmuster, die Argumentvariable und den Rumpf Ihrer Prozedurdeklaration.

1.4 Vergleiche und Konditionale

Ein Vergleich ist eine Operation, die testet, ob eine Bedingung erfüllt ist, und dementsprechend den Wert *false* oder *true* liefert:

```
3<3
```
false : bool

```
3<=3
```
true : bool

```
3>=3
```
true : bool

```
3=3
```
true : bool

```
3<>3
```
false : bool

Die Zeichenkombinationen <=, >= und <> bezeichnen die Operatoren für die Vergleiche $\leq$, $\geq$ und $\neq$. Die Werte *false* und *true* werden als **Boolesche Werte** bezeichnet und haben den Typ *bool,* der keine sonstigen Werte hat.

Ein **Konditional** ist ein Ausdruck, der eine Fallunterscheidung gemäß eines Booleschen Werts realisiert:

```
if false then 5 else 7
```
7 : int

```
if true then 5 else 7
```
5 : int

Da Vergleiche Boolesche Werte liefern, können sie mit Konditionalen kombiniert werden:

```
if 4<2 then 3*5 else 7*1
```
7 : int

```
if 4=2*2 then 3*5 else 7*1
```
15 : int

Konditionale werden vor allem im Rumpf von Prozeduren verwendet. Hier ist eine Prozedur, die den sogenannten Absolutbetrag einer ganzen Zahl liefert:

```
fun betrag (x:int) = if x<0 then ~x else x
```
val betrag : int → int

```
betrag ~3
```
3 : int

Allgemein hat ein Konditional die Form *if* e_1 *then* e_2 *else* e_3. Die Teilausdrücke e_1, e_2 und e_3 werden als **Bedingung**, **Konsequenz** und **Alternative** des Konditionals bezeichnet. Bei *if*, *then* und *else* handelt es sich um Schlüsselwörter. In die Bedingung, Konsequenz oder Alternative eines Konditionals können weitere Konditionale geschachtelt werden:

```
if 4<2 then 3 else if 2<3 then ~1 else 1
```
~1 : int

Aufgabe 1.3 (Signum) Schreiben Sie eine Prozedur *signum* : *int* → *int*, die für negative Argumente −1, für positive Argumente 1, und für 0 das Ergebnis 0 liefert.

1.5 Lokale Deklarationen und Hilfsprozeduren

Das Programm

```
val a = 2*2
val b = a*a
val c = b*b
```

berechnet die Potenz 2^8 mithilfe von 3 Multiplikationen. Wir stellen uns jetzt die Frage, wie wir eine Prozedur schreiben können, die zu einer beliebigen Zahl x die Potenz x^8 mit nur 3 Multiplikationen bestimmt. Das gelingt mithilfe sogenannter **lokaler Deklarationen**:

```
fun hoch8 (x:int) =
    let
        val a = x*x
        val b = a*a
    in
        b*b
    end
```
val hoch8 : int → int

```
hoch8 2
```
256 : int

Statt mit lokalen Deklarationen können wir auch mit einer **Hilfsprozedur** arbeiten:

```
fun q (y:int) = y*y
fun hoch8 (x:int) = q (q (q x))
```

Aufgabe 1.4 Schreiben Sie eine Prozedur $hoch17 : int \to int$, die zu einer Zahl x die Potenz x^{17} berechnet. Dabei sollen möglichst wenig Multiplikationen verwendet werden. Schreiben Sie die Prozedur auf zwei Arten: Mit einer Hilfsprozedur und mit lokalen Deklarationen.

1.6 Tupel

Ein **Tupel** ist eine Folge $(v_1, \ldots, v_n)$ von Werten. Wir unterscheiden zwischen den **Positionen** (die Zahlen $1, \ldots, n$) und den **Komponenten** (die Werte $v_1, \ldots, v_n$) eines Tupels. Die Komponente v_i an der i-ten Position eines Tupels bezeichnen wir als die **i-te Komponente** des Tupels. Die Anzahl der Positionen eines Tupels bezeichnen wir als die **Länge** des Tupels. Als Beispiel betrachten wir das Tupel $(7, 2, true, 2)$. Dieses Tupel hat die Positionen 1, 2, 3, 4; die Komponenten 2, 7, *true*; und die Länge 4. Hier sind Beispiele für Ausdrücke, die Tupel beschreiben:

```
(7, 2, true, 2)
```
*(7, 2, true, 2) : int * int * bool * int*

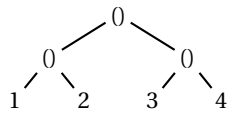
```
(7+2, 2*7)
```
*(9, 14) : int * int*

Der Typ eines Tupelausdrucks ergibt sich aus den Typen der Ausdrücke, die die Komponenten des Tupels beschreiben. Beispielsweise hat der Tupelausdruck $(3, true, 3)$ den Typ $int * bool * int$. Allgemein besteht ein **Tupeltyp** $t_1 * \cdots * t_n$ aus $n \geq 2$ Typen $t_1, \ldots, t_n$.

Tupel sind sogenannte **zusammengesetzte Werte**. Das bedeutet insbesondere, dass Tupel Werte sind. Also ist es möglich, Tupel zu größeren Tupeln zusammenzufassen:

$((1,2), (3,4)) : (int * int) * (int * int)$

Den Aufbau solcher **geschachtelten Tupel** können wir grafisch mithilfe von **Baumdarstellungen** verdeutlichen:



Wir führen noch einige zusätzliche Sprechweisen für Tupel ein. Unter einem n-**stelligen Tupel** verstehen wir ein Tupel der Länge n. Zweistellige Tupel bezeichnen wir als **Paare** und dreistellige Tupel als **Tripel**. Das nullstellige Tupel () heißt auch **leeres Tupel**. Es hat den speziellen Typ *unit*, der keine weiteren Werte hat:

```
()
() : unit
```

Gedanklich unterscheiden wir zwischen dem einstelligen Tupel (5) und der Zahl 5. Einstellige Tupel lassen sich in Standard ML allerdings nicht mit Klammern darstellen, da der Ausdruck (5) als eine geklammerte Darstellung der Zahl 5 interpretiert wird.

Auf die Komponenten eines Tupels kann mithilfe sogenannter **Projektionen** zugegriffen werden:

```
val x = (5-2, 1<2, 2*2)
val x = (3, true, 4) : int * bool * int

#3 x
4 : int

#2 x
true : bool
```

Aufgabe 1.5

a) Geben Sie ein Tupel mit 3 Positionen und nur einer Komponente an.

b) Geben Sie einen Tupelausdruck an, der den Typ $int * (bool * (int * unit))$ hat.

c) Geben Sie ein Paar an, dessen erste Komponente ein Paar und dessen zweite Komponente ein Tripel ist.

1.7 Kartesische Muster

Auf die Komponenten eines Tupels kann man auch mit Deklarationen zugreifen, die mit **kartesischen Mustern** formuliert sind:

```
val (x,y) = (3,4)
val x = 3 : int
val y = 4 : int
```

Diese Deklaration bindet die Bezeichner x und y gemäß dem Muster (x, y) an die erste und zweite Komponente des Paares $(3,4)$.

Ein interessantes Beispiel ist die Deklaration

```
val (x,y) = (y,x)
val x = 4 : int
val y = 3 : int
```

Diese vertauscht die Werte der Bezeichner x und y.

Wir wollen jetzt eine Prozedur

$$swap : int * int \to int * int$$

deklarieren, die die Komponenten eines Paares vertauscht. Beispielsweise soll $swap(3,4)$ das Paar $(4,3)$ liefern. Bei *swap* handelt es sich also um eine Prozedur, die ein Paar als Argument bekommt und ein Paar als Ergebnis liefert. Hier sind drei mögliche Deklarationen für *swap*:

```
fun swap (p:int*int) = (#2p, #1p)

fun swap (p:int*int) = let val (x,y) = p in (y,x) end

fun swap (x:int, y:int) = (y,x)
```

Die Deklarationen liefern Prozeduren, die sich nach außen hin völlig gleich verhalten. Die dritte Deklaration formuliert *swap* mit einem **kartesischen Argumentmuster**, das zwei Argumentvariablen x und y einführt.

Hier ist eine Prozedur, die zu zwei Zahlen die größere liefert:

```
fun max (x:int, y:int) = if x<y then y else x
val max : int * int → int

max (5,3)
5 : int

max (~5,3)
3 : int
```

Obwohl das streng genommen nicht der Fall ist, werden wir sagen, dass Prozeduren wie *swap* und *max* zwei Argumente haben. Diese Sprechweise ist zwar ungenau, aber bequem und allgemein üblich.

Aufgabe 1.6 Schreiben Sie eine Prozedur $min : int * int \to int$, die zu zwei Zahlen die kleinere liefert. Deklarieren Sie *min* analog zu *swap* auf 3 verschiedene Arten: mit Projektionen, mit einer lokalen Deklaration und mit einem kartesischen Argumentmuster.

Aufgabe 1.7 Schreiben Sie eine Prozedur $max : int * int * int \to int$, die zu drei Zahlen die größte liefert, auf zwei Arten:

a) Benutzen Sie keine Hilfsprozedur und drei Konditionale.

b) Benutzen Sie eine Hilfsprozedur und insgesamt nur ein Konditional.

1.8 Ganzzahlige Division: Div und Mod

Ganzzahlige Division und Restbestimmung sind über die Operatoren *div* und *mod* verfügbar:

```
12 div 3
```
4 : int

```
12 mod 3
```
0 : int

```
12 div 5
```
2 : int

```
12 mod 5
```
2 : int

Beim Programmieren mit Zahlen spielen die Operationen Div und Mod eine wichtige Rolle. Es lohnt sich, ihre Definitionen zu kennen:

$$x \operatorname{div} y = \left\lfloor \frac{x}{y} \right\rfloor$$

$$x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor y$$

Dabei bezeichnet $\left\lfloor \frac{x}{y} \right\rfloor$ die größte ganze Zahl, die kleiner gleich $\frac{x}{y}$ ist.

Division durch null ist bekanntlich undefiniert. Wenn ein Interpreter eine Operation ausführt, die für die gegebenen Argumente undefiniert ist, bricht er die Ausführung des Programms mit einem **Laufzeitfehler** ab:

```
1 div 0
```
! Uncaught exception: Div

Aufgabe 1.8 Bestimmen Sie gemäß der obigen Definitionen den Wert von $(3 \bmod -2)$ und $(-3 \bmod 2)$. Überzeugen Sie sich davon, dass Ihr Interpreter die richtigen Werte liefert.

Aufgabe 1.9 Machen Sie sich klar, dass für zwei natürliche Zahlen x, y mit $x \geq 0$ und $y > 0$ gilt: $0 \leq (x \bmod y) < y$.

Aufgabe 1.10 Schreiben Sie eine Prozedur *teilbar* : $int * int \to bool$, die für (x, y) testet, ob x durch y ohne Rest teilbar ist.

Aufgabe 1.11 (Zeitangaben) Oft gibt man eine Zeitdauer im *HMS-Format* mit Stunden, Minuten und Sekunden an. Beispielsweise ist 2h5m26s eine hervorragende Zeit für einen Marathonlauf.

a) Schreiben Sie eine Prozedur $sec : int * int * int \to int$, die vom HMS-Format in Sekunden umrechnet. Beispielsweise soll $sec(1,1,6)$ die Zahl 3666 liefern.

b) Schreiben Sie eine Prozedur $hms : int \to int * int * int$, die eine in Sekunden angegebene Zeit in das HMS-Format umrechnet. Beispielsweise soll *hms* 3666 das Tupel $(1,1,6)$ liefern. Berechnen Sie die Komponenten des Tupels mithilfe lokaler Deklarationen.

1.9 Rekursion

Für die Berechnung vieler Funktionen benötigt man Prozeduren, die Gleichungen wiederholt anwenden. Als Beispiel betrachten wir eine Prozedur, die Potenzen x^n durch wiederholte Multiplikation mit x berechnet. Dabei soll n eine natürliche Zahl sein $(0,1,2,\dots)$. Wir formulieren die Prozedur zunächst mithilfe von zwei Gleichungen, die wir als **Rekursionsgleichungen** bezeichnen:

$$x^0 = 1$$
$$x^n = x \cdot x^{n-1} \qquad \text{für } n > 0$$

Wenn wir die Rekursionsgleichungen von links nach rechts anwenden, können wir jede Potenz berechnen. Hier ist ein Beispiel:

$$\begin{aligned}
2^3 &= 2 \cdot 2^2 && \text{zweite Gleichung}\\
&= 2 \cdot 2 \cdot 2^1 && \text{zweite Gleichung}\\
&= 2 \cdot 2 \cdot 2 \cdot 2^0 && \text{zweite Gleichung}\\
&= 2 \cdot 2 \cdot 2 \cdot 1 && \text{erste Gleichung}\\
&= 8
\end{aligned}$$

Der Clou bei dieser Berechnung ist, dass die zweite Gleichung größere Potenzen auf kleinere zurückführt und dass durch wiederholtes Anwenden der zweiten Gleichung der Exponent auf 0 reduziert wird, sodass die Berechnung mit der ersten Gleichung beendet werden kann. Die wiederholte Anwendung der zweiten Gleichung wird als **Rekursion** bezeichnet.

Um den durch die Rekursionsgleichungen gegebenen Algorithmus in Standard ML durch eine Prozedur zu realisieren, verbinden wir die beiden Gleichungen mithilfe eines Konditionals zu einer Gleichung:

$$x^n = \text{if } n > 0 \text{ then } x \cdot x^{n-1} \text{ else } 1 \qquad \text{für } n \geq 0$$

Aus dieser Gleichung ergibt sich die folgende Prozedurdeklaration:

```
fun potenz (x:int, n:int) : int =
    if n>0 then x*potenz(x,n-1) else 1
val potenz : int * int → int
```

$$
\begin{aligned}
\underline{potenz(4,2)} &= \underline{if\ 2>0}\ then\ 4*potenz(4,2-1)\ else\ 1\\
&= \underline{if\ true\ then\ 4*potenz(4,2-1)\ else\ 1}\\
&= 4*potenz(4,\underline{2-1})\\
&= 4*\underline{potenz(4,1)}\\
&= 4*(\underline{if\ 1>0}\ then\ 4*potenz(4,1-1)\ else\ 1)\\
&= 4*(\underline{if\ true\ then\ 4*potenz(4,1-1)\ else\ 1})\\
&= 4*(4*potenz(4,\underline{1-1}))\\
&= 4*(4*\underline{potenz(4,0)})\\
&= 4*(4*(\underline{if\ 0>0}\ then\ 4*potenz(4,0-1)\ else\ 1))\\
&= 4*(4*(\underline{if\ false\ then\ 4*potenz(4,0-1)\ else\ 1}))\\
&= 4*\underline{(4*1)}\\
&= \underline{4*4}\\
&= 16
\end{aligned}
$$

Abbildung 1.1: *Ein Ausführungsprotokoll*

```
potenz (2,10)
```
1024 : int

Prozeduren, deren Rumpf so wie *potenz* eine **Selbstanwendung** enthält, werden als **rekursiv** bezeichnet. Selbstanwendungen werden auch als **rekursive Anwendungen** bezeichnet. Bei der Deklaration rekursiver Prozeduren geben wir, so wie bei *potenz* gezeigt, den Ergebnistyp der Prozedur nach dem Argumentmuster an, damit der Typ der Prozedur für die Überprüfung der Selbstanwendung zur Verfügung steht.

Das in Abbildung 1.1 gezeigte **Ausführungsprotokoll** beschreibt die Ausführung der Prozeduranwendung $potenz(4,2)$ im Detail. Jeder **Ausführungsschritt** betrifft einen Teilausdruck, der jeweils durch Unterstreichung hervorgehoben ist. Das **verkürzte Ausführungsprotokoll** für die Anwendung $potenz(4,2)$ fasst jeweils mehrere Ausführungsschritte zusammen:

$$
\begin{aligned}
potenz(4,2) &= 4*potenz(4,1)\\
&= 4*(4*potenz(4,0))\\
&= 4*(4*1)\\
&= 16
\end{aligned}
$$

Bei der Ausführung einer Prozeduranwendung unterscheiden wir zwei Phasen. Die erste Phase führt die Teilausdrücke der Anwendung aus und bestimmt eine Prozedur und einen als Argument dienenden Wert. Das aus der Prozedur und dem Wert bestehende

Paar bezeichnen wir als **Prozeduraufruf**. Beispielsweise liefert die Prozeduranwendung $potenz(2*2,\ 1+1)$ den Prozeduraufruf, der aus der Prozedur *potenz* und dem Wert $(4,2)$ besteht. Die zweite Phase führt den Prozeduraufruf aus.

Für die Ausführung eines Aufrufs einer rekursiven Prozedur müssen in der Regel weitere Aufrufe der Prozedur ausgeführt werden. Dadurch ergibt sich eine Folge von Aufrufen, die als **Rekursionsfolge** bezeichnet wird. Für den Aufruf $potenz(4,3)$ ergibt sich beispielsweise die folgende Rekursionsfolge:

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

Die Prozedur *potenz* wählt die richtige Rekursionsgleichung mithilfe eines Konditionals aus. Damit das wie gezeigt funktioniert, ist wesentlich, dass bei der Ausführung eines Konditionals zunächst nur die Bedingung ausgeführt wird. Abhängig davon, welchen Wert die Bedingung liefert, wird dann entweder nur die Konsequenz oder nur die Alternative des Konditionals ausgeführt. Das sorgt dafür, dass es bei einem Aufruf $potenz(x,0)$ zu keinem Aufruf $potenz(x,-1)$ kommt.

Aufgabe 1.12 Sei die folgende rekursive Prozedurdeklaration gegeben:

```
fun f(n:int, a:int) : int = if n=0 then a else f(n-1, a*n)
```

a) Geben Sie die Rekursionsfolge für den Aufruf $f(3,1)$ an.

b) Geben Sie ein verkürztes Ausführungsprotokoll für den Ausdruck $f(3,1)$ an.

c) Geben Sie ein detailliertes Ausführungsprotokoll für den Ausdruck $f(3,1)$ an. Halten Sie sich dabei an das Beispiel in Abbildung 1.1. Wenn es mehrere direkt ausführbare Teilausdrücke gibt, soll immer der am weitesten links stehende zuerst ausgeführt werden. Sie sollten insgesamt 18 Ausführungsschritte bekommen.

Aufgabe 1.13 Schreiben Sie eine rekursive Prozedur $mul : int * int \rightarrow int$, die das Produkt einer natürlichen und einer ganzen Zahl durch wiederholte Addition berechnet. Beschreiben Sie den zugrunde liegenden Algorithmus zunächst mit Rekursionsgleichungen.

Aufgabe 1.14 Der ganzzahlige Quotient $x\ div\ y$ lässt sich aus x durch wiederholtes Subtrahieren von y bestimmen. Schreiben Sie eine rekursive Prozedur $mydiv : int * int \rightarrow int$, die für $x \geq 0$ und $y \geq 1$ das Ergebnis $x\ div\ y$ liefert. Geben Sie zunächst Rekursionsgleichungen für $x\ div\ y$ an.

Aufgabe 1.15 Auch der ganzzahlige Rest $x\ mod\ y$ lässt sich aus x durch wiederholtes Subtrahieren von y bestimmen. Schreiben Sie eine rekursive Prozedur $mymod : int * int \rightarrow int$, die für $x \geq 0$ und $y \geq 1$ das Ergebnis $x\ mod\ y$ liefert. Geben Sie dazu zunächst Rekursionsgleichungen für $x\ mod\ y$ an.

Eine häufig gestellte Frage

Wenn in Aufgaben wie 1.13 oder 1.14 nach einer Prozedur gefragt wird, deren Ergebnisse nur für einen Teil der Argumente spezifiziert sind, was soll die Prozedur dann für die anderen Argumente machen?

Wie sich eine Prozedur für Argumente verhält, für die die Aufgabenstellung keine Vorgaben macht, spielt für die Lösung der Aufgabe keine Rolle und Sie sollten sich darüber auch keine Gedanken machen. Die partielle Spezifikation von Prozeduren soll Ihnen die Beschäftigung mit unwesentlichen Details ersparen.

Aufgabe 1.16 (Stelligkeit) Schreiben Sie eine rekursive Prozedur $stell : int \to int$, die zu einer Zahl die Stelligkeit ihrer Dezimaldarstellung liefert. Beispielsweise soll $stell\ 117 = 3$ gelten. Geben Sie zunächst die Rekursionsgleichungen für *stell* an. Verwenden Sie ganzzahlige Division durch 10, um die Zahl zu erhalten, die durch Streichen der letzten Ziffer entsteht.

Aufgabe 1.17 (Quersumme) Schreiben Sie eine rekursive Prozedur $quer : int \to int$, die die Quersumme einer ganzen Zahl berechnet. Die Quersumme einer Zahl ist die Summe ihrer Dezimalziffern. Beispielsweise hat die Zahl -3754 die Quersumme 19. Geben Sie zunächst die Rekursionsgleichungen für *quer* an. Verwenden Sie Restbestimmung modulo 10, um die letzte Ziffer einer Zahl zu bestimmen.

Aufgabe 1.18 (Binomialkoeffizienten) Schreiben Sie eine rekursive Prozedur $binom : int * int \to int$, die für $n, k \geq 0$ den Binomialkoeffizienten $\binom{n}{k}$ berechnet. Verwenden Sie den durch die folgenden Gleichungen gegebenen Algorithmus:

$$\binom{n}{0} = 1 \qquad \binom{0}{k} = 0 \quad \text{für } k > 0 \qquad \binom{n}{k} = \frac{n \cdot \binom{n-1}{k-1}}{k} \quad \text{für } n, k > 0$$

Aufgabe 1.19 (Sortieren von Tripeln)

a) Schreiben Sie eine Prozedur $sort : int * int * int \to int * int * int$, die ganzzahlige Tripel sortiert. Beispielsweise soll $sort(7,2,5) = (2,5,7)$ gelten. Verwenden Sie nur 2 Konditionale und Rekursion.

b) Schreiben Sie mithilfe von *sort* eine Prozedur $max : int * int * int \to int$, die zu drei Zahlen die größte liefert. Verwenden Sie dabei keine kartesischen Muster.

1.10 Natürliche Quadratwurzeln und Akkus

Nicht jede Funktion lässt sich unmittelbar durch Rekursionsgleichungen berechnen. Oft ist es erforderlich, die Funktion durch Rückführung auf eine Hilfsfunktion zu bestimmen, die mit zusätzlichen Argumenten versehen ist.

Ein typisches Beispiel ist die Funktion $\lfloor\sqrt{n}\rfloor$, die zu einer natürlichen Zahl n die **natürliche Quadratwurzel** liefert. Auf der Suche nach einem Algorithmus erweisen sich die folgenden Gleichungen als hilfreich:

$$\begin{aligned}\lfloor\sqrt{n}\rfloor &= \max\{k \in \mathbb{N} \mid k^2 \leq n\} \\ &= \min\{k \in \mathbb{N} \mid k^2 > n\} - 1\end{aligned}$$

Die zweite Gleichung führt die Bestimmung von $\lfloor\sqrt{n}\rfloor$ auf die Berechnung der kleinsten natürlichen Zahl k mit der Eigenschaft $k^2 > n$ zurück. Diese Zahl können wir bestimmen, indem wir einen *Zähler k*, der zunächst den Wert 1 hat, solange um 1 erhöhen, bis erstmals die Eigenschaft $k^2 > n$ gilt. Um dieses Vorgehen mit Gleichungen formulieren zu können, führen wir eine Hilfsfunktion

$$\begin{aligned}&w\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ &w(k,n) = \min\{i \in \mathbb{N} \mid i \geq k \text{ und } i^2 > n\}\end{aligned}$$

ein, deren erstes Argument dem Zähler entspricht. Da

$$\lfloor\sqrt{n}\rfloor = w(1,n) - 1$$

gilt, können wir $\lfloor\sqrt{n}\rfloor$ durch Rückführung auf die Hilfsfunktion w bestimmen. Die Berechnung von w gelingt mithilfe der folgenden Rekursionsgleichungen:

$$\begin{aligned}w(k,n) &= k && \text{für } k^2 > n \\ w(k,n) &= w(k+1,n) && \text{für } k^2 \leq n\end{aligned}$$

Überzeugen Sie sich davon, dass die Funktion w die Rekursionsgleichungen tatsächlich erfüllt. Hier ist das verkürzte Ausführungsprotokoll für die Bestimmung der natürlichen Quadratwurzel von 15:

$$\begin{aligned}\left\lfloor\sqrt{15}\right\rfloor &= w(1,15) - 1 \\ &= w(2,15) - 1 \\ &= w(3,15) - 1 \\ &= w(4,15) - 1 \\ &= 4 - 1 = 3\end{aligned}$$

Die Umsetzung der Gleichungen in zwei Prozeduren ist einfach:

```
fun w (k:int,n:int) : int = if k*k>n then k else w(k+1,n)
val w : int * int → int

fun wurzel (n:int) = w(1,n)-1
val wurzel : int → int

wurzel 15
3 : int
```

Zusammenfassend stellen wir fest, dass wir einen Algorithmus für die Berechnung von natürlichen Quadratwurzeln entwickelt haben, der mithilfe von Gleichungen formuliert ist. Dazu war die Einführung einer Hilfsfunktion mit einem zusätzlichen Argument erforderlich. Solche zusätzlichen Argumente werden prägnant als **Akkus** bezeichnet (Kurzform für **Akkumulatorargument**).

Aufgabe 1.20 Schreiben Sie eine Prozedur, die zu $n \in \mathbb{N}$ das kleinste $k \in \mathbb{N}$ mit $k^3 \geq n$ berechnet. Definieren Sie zunächst die zugrunde liegenden Funktionen und geben Sie die zu ihrer Berechnung erforderlichen Gleichungen an.

1.11 Endrekursion

Unter einer **endrekursiven Prozedur** versteht man eine rekursive Prozedur, bei der das Ergebnis im Rekursionsfall unmittelbar durch eine rekursive Anwendung geliefert wird. Ein typisches Beispiel ist die Prozedur w aus dem letzten Abschnitt:

```
fun w (k:int, n:int) : int = if k*k>n then k else w(k+1,n)
```

Wenn die Selbstanwendung $w(k+1,n)$ zur Ausführung kommt, dann liefert sie das Ergebnis der Prozedur. Diese Eigenschaft kann man sehr schön in den verkürzten Ausführungsprotokollen sehen:

$$w(1,24) = w(2,24) = w(3,24) = w(4,24) = w(5,24) = 5$$

Bei Endrekursion handelt es sich um eine einfache Form der Rekursion, die besonders effizient ausgeführt werden kann. Statt von Endrekursion spricht man auch von **iterativer Rekursion**.[1]

Unser Paradebeispiel für Rekursion, die Prozedur *potenz* aus § 1.9, ist nicht endrekursiv. Es stellt sich jetzt die Frage, ob wir Potenzen auch mit einer endrekursiven Prozedur berechnen können. Die Antwort ist ja, allerdings müssen wir dazu wie bei der natürlichen Quadratwurzel eine Hilfsfunktion mit einem Akku berechnen:

$$p : \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$$
$$p(a,x,n) = a \cdot x^n$$

Da $p(1,x,n) = x^n$ gilt, können wir die Potenzen mit einer Prozedur bestimmen, die die Funktion p berechnet. Und für p ist es nicht weiter schwer, endrekursive Rekursionsgleichungen anzugeben:

$$p(a,x,0) = a$$
$$p(a,x,n) = p(a \cdot x,\, x,\, n-1) \qquad \text{für } n > 0$$

Damit gelingt die endrekursive Berechnung von Potenzen in Standard ML wie folgt:

[1] Für Experten: Endrekursive Prozeduren können mit Schleifen realisiert werden. Siehe § 15.8.

```
fun p (a:int, x:int, n:int) : int = if n<1 then a else p(a*x,x,n-1)
```
*val p : int * int * int → int*

```
fun potenz (x:int, n:int) = p(1,x,n)
```
*val potenz : int * int → int*

Aufgabe 1.21 Schreiben Sie eine Prozedur $mul : int * int \to int$, die das Produkt aus einer natürlichen und einer ganzen Zahl mit einer endrekursiven Hilfsprozedur durch wiederholte Addition berechnet.

Aufgabe 1.22 Schreiben Sie eine Prozedur $quer : int \to int$, die die Quersumme einer ganzen Zahl mithilfe einer endrekursiven Hilfsprozedur berechnet.

Aufgabe 1.23 (Reversion) Unter der Reversion $rev\,n$ einer natürlichen Zahl n wollen wir die natürliche Zahl verstehen, die man durch Spiegeln der Dezimaldarstellung von n erhält. Beispielsweise soll $rev\,1234 = 4321$, $rev\,76 = 67$ und $rev\,1200 = 21$ gelten.

a) Schreiben Sie zunächst eine endrekursive Prozedur $rev' : int * int \to int$, die zu zwei natürlichen Zahlen m und n die Zahl liefert, die sich ergibt, wenn man die reversierte Dezimaldarstellung von n rechts an die Dezimaldarstellung von m anfügt. Beispielsweise soll $rev'(65,73) = 6537$ und $rev'(0,12300) = 321$ gelten. Die Arbeitsweise von rev' ergibt sich aus dem verkürzten Ausführungsprotokoll $rev'(65,73) = rev'(653,7) = rev'(6537,0) = 6537$.

b) Schreiben Sie mithilfe der Prozedur rev' eine Prozedur rev, die natürliche Zahlen reversiert.

c) Machen Sie sich klar, dass die entscheidende Idee bei der Konstruktion des Reversionsalgorithmus die Einführung einer Hilfsfunktion mit einem Akku ist. Überzeugen Sie sich davon, dass Sie rev nicht ohne Weiteres durch Rekursionsgleichungen bestimmen können.

1.12 Divergenz

Betrachten Sie die endrekursive Prozedur

```
fun p (x:int) : int = p x
```

Die Ausführung eines Aufrufs $p\,x$ der Prozedur schreitet unendlich voran, da die Ausführung des Aufrufs $p\,x$ erneut zur Ausführung des Aufrufs $p\,x$ führt:

$$p\,x = p\,x = p\,x = p\,x = p\,x = \cdots$$

Eine Ausführung, die nach endlich vielen Schritten endet, bezeichnet man als **terminierend**. Eine nicht terminierende Ausführung bezeichnet man dagegen als **divergierend**. Man sagt auch, dass eine Prozedur für bestimmte Argumente **terminiert** oder **divergiert**. Die obige Prozedur divergiert für alle Argumente.

Hier ist eine zweite, nicht endrekursive Prozedur, die wie p für alle Argumente divergiert:

Eine gewinnbringende Strategie

Programmieren können Sie nur durch eigene geistige Aktivität lernen. Bloßes Lesen eines Kapitels genügt nicht. Die beschriebene gedankliche Welt muss in Ihrem Kopf so konkrete Formen annehmen, dass Sie sich darin wie in einer vertrauten Umgebung bewegen können. Der dafür erforderliche Mindestaufwand ist die Bearbeitung der Übungsaufgaben. Bearbeitung heißt, die Lösungen der Aufgaben selbstständig durch Nachdenken und Probieren zu entwickeln.

Beim Einstieg in die Programmierung erweist sich gezieltes Experimentieren als eine gewinnbringende Lernstrategie. Erproben Sie alle Beispielprogramme dieses Buchs mit einem Interpreter. Dabei werden Ihnen oft interessante Varianten einfallen. Wenn Sie diese erkunden, können Sie in kurzer Zeit viel lernen.

Bei den meisten Übungsaufgaben sollen Sie kleine Programme schreiben. Erproben Sie Ihre Programme stets mit einem Interpreter und überzeugen Sie sich davon, dass sie tun, was sie tun sollen. Typischerweise wird das zunächst nicht der Fall sein. Dann müssen Sie durch Experimentieren, Nachdenken und erneutes Studium des Lehrmaterials schrittweise herausfinden, was Sie wo falsch gemacht haben. Dieser als **Debugging** bezeichnete Prozess ist ein wesentlicher Teil der Lernarbeit, der Ihre Problemlösefähigkeit schult und Ihr Verständnis vertieft.

```
fun q (x:int) : int = 0 + q x
```

Im Unterschied zu p werden die bei der Ausführung von q durchlaufenen Berechnungszustände jedoch immer größer, sodass mehr und mehr Speicherplatz benötigt wird.

$$q\ x\ =\ 0+(q\ x)\ =\ 0+(0+(q\ x))\ =\ 0+(0+(0+(q\ x)))\ =\ \ldots$$

Das hat zur Folge, dass ein Interpreter die Ausführung von q nach kurzer Zeit wegen **Speichererschöpfung** abbrechen muss:

```
val it = q 0
! Uncaught exception: Out_of_memory
```

Bei der Ausführung mit einem Interpreter kann sich die Divergenz einer rekursiven Prozedur also entweder dadurch zeigen, dass der Interpreter mit der Ausführung nicht zum Ende kommt oder dass der Interpreter die Ausführung wegen Speichererschöpfung abbricht. Generell gibt es bei der Ausführung eines Programms durch einen Interpreter die folgenden Möglichkeiten:

- **Reguläre Terminierung**. Die Ausführung des Programms endet nach endlich vielen Schritten erfolgreich. Das ist der Normalfall.
- **Abbruch wegen Laufzeitfehler**. Der Interpreter bricht die Ausführung des Programms ab, da eine Operation einen Fehler signalisiert. Ein typisches Beispiel ist Division durch null.

- **Abbruch wegen Speichererschöpfung**. Der Interpreter bricht die Ausführung des Programms ab, da der dem Interpreter zur Verfügung stehende Speicherplatz erschöpft ist.
- **Abbruch durch den Benutzer**. Die noch laufende Ausführung des Programms wird durch den Benutzer abgebrochen.

Machen Sie sich klar, dass es sich bei Divergenz um ein gedankliches Konzept handelt, das sich nur bedingt mit einem Interpreter beobachten lässt. Wenn ein Interpreter bereits n Tage an der Ausführung eines Programms rechnet, ist offen, ob er nach weiteren n Tagen regulär terminiert, abbricht oder immer noch rechnet.

Aufgabe 1.24 Schreiben Sie eine Prozedur $int \to int$, die für negative Argumente divergiert und für nicht-negative Argumente x das Ergebnis x liefert.

Aufgabe 1.25 Dieter Schlau liebt Prozeduren. Er deklariert die Prozedur

```
fun ifi (b:bool, x:int, y:int) = if b then x else y
```

und behauptet, dass er sie anstelle des Konditionals verwenden kann, wenn Konsequenz und Alternative den Typ *int* haben. Anna ist skeptisch und zeigt ihm schließlich die folgenden Deklarationen:

```
fun p (n:int) : int = if n=0 then p(n-1) else n
fun q (n:int) : int = ifi(n=0, q(n-1), n)
```

Dieter kann erst keinen Unterschied im Verhalten der Prozeduren p und q erkennen, aber ein Experiment mit dem Interpreter belehrt ihn eines Besseren.

Für welche Argumente verhalten sich die Prozeduren p und q unterschiedlich? Warum? Welche Eigenschaft des Konditionals geht bei der Verwendung der Prozedur *ifi* verloren?

1.13 Festkomma- und Gleitkommazahlen

Computer arbeiten mit zwei unterschiedlichen Zahlensystemen, deren Elemente als **Festkomma-** beziehungsweise **Gleitkommazahlen** bezeichnet werden. Beide Systeme arbeiten mit Darstellungen fester Größe (oft 32 oder 64 Bit) und umfassen daher jeweils nur endlich viele Zahlen. Festkommazahlen entsprechen ganzen Zahlen, und Gleitkommazahlen reellen Zahlen mit endlicher Dezimalbruchdarstellung.

1.13.1 Festkommazahlen

Die Festkommazahlen heutiger PCs stellen die ganzen Zahlen im Intervall $\{-2^{31}, \ldots, 2^{31}-1\}$ dar. Wenn eine Operation wie Addition oder Multiplikation zu einer Zahl außerhalb des darstellbaren Intervalls führt, spricht man von einem **Überlauf**.

Standard ML Interpreter realisieren die Werte des Typs *int* mit Festkommazahlen. Auf PCs schränken sie den darstellbaren Bereich meistens auf das Intervall $\{-2^{30}, \ldots, 2^{30}-1\}$

ein. Wenn es bei der Ausführung eines Programms zu einem Überlauf kommt, wird der Laufzeitfehler *Overflow* signalisiert:

```
4*1073741823
! Uncaught exception: Overflow
```

Aufgabe 1.26 (Fakultäten) Für $n \geq 0$ können wir die sogenannte n-te Fakultät $n!$ wie folgt definieren:

$$0! = 1$$
$$n! = 1 \cdot \cdots \cdot n \qquad \text{für } n \geq 1$$

Beispielsweise gilt $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$.

a) Geben Sie zwei Gleichungen an, mit denen $n!$ berechnet werden kann.

b) Realisieren Sie die Gleichungen mit einer Prozedur $fac : int \to int$. Schreiben Sie die Prozedur so, dass ihre Ausführung für negative Argumente wegen Speichererschöpfung abgebrochen wird.

c) Die Fakultäten werden schnell groß. Beispielsweise gilt $10! = 3628800$. Ermitteln Sie mit einem Interpreter das erste n, für das die Ausführung Ihrer Prozedur zu einem Überlauf führt.

1.13.2 Gleitkommazahlen

Eine Gleitkommazahl stellt eine Zahl x gemäß der Gleichung

$$x = m \cdot 10^n$$

durch zwei ganze Zahlen m und n dar, die als **Mantisse** und **Exponent** bezeichnet werden. Beispielweise wird die Zahl $12,65 = 1265 \cdot 10^{-2}$ durch das Paar $(1265, -2)$ dargestellt. Mantisse und Exponent müssen jeweils innerhalb eines vorgegebenen endlichen Intervalls liegen. Die entsprechenden Gleitkommazahlen beschreiben endlich viele Stützpunkte innerhalb eines Intervalls, das aus unendlich vielen reellen Zahlen besteht.

Gleitkommazahlen sind in Standard ML als Werte des Typs *real* verfügbar. Hier ist ein Beispiel:

```
4.5 + 2.0 * 5.5
15.5 : real
```

Die Konstanten für *real* halten sich an die englische Schreibweise und verwenden statt des bei uns üblichen Kommas den Punkt.

Für die Bezeichnung der Gleitkommaoperationen (z. B. Addition, Multiplikation) verwendet Standard ML dieselben Operatoren (z. B. $+$, $*$) wie für die Festkommaoperationen. Man sagt, dass diese Operatoren **überladen** sind, da sie gemäß zweier Typen an-

wendbar sind:

$int * int \to int$
$real * real \to real$

Bevor ein Programm ausgeführt wird, wird für jedes Auftreten eines überladenen Operators ermittelt, welche Operation es bezeichnet. Die Typen der überladenen Operatoren lassen keine gemischten Argumente zu:

```
2 * 5.5
```
! Type clash: expression of type int cannot have type real

Eine solche Mischung wäre nicht sinnvoll, da Festkomma- und Gleitkommazahlen zu unterschiedlichen Zahlensystemen gehören.

Die Beschreibung sehr großer und sehr kleiner Gleitkommazahlen wird durch spezielle Konstanten unterstützt:

`1.7878E45` $\rightsquigarrow$ $1{,}7878 \cdot 10^{45}$

`1.7878E~45` $\rightsquigarrow$ $1{,}7878 \cdot 10^{-45}$

Aufgabe 1.27 Schreiben Sie eine Prozedur p: $real * real \to real$, die die Funktion

$f(x,y) = (x-3)(y+5)^2$

mit Gleitkommazahlen berechnet. Verwenden Sie eine lokale Deklaration, damit die Addition $y+5$ nur einmal berechnet werden muss.

1.13.3 Rundungsfehler

Wenn eine Gleitkommaoperation zu einer Zahl außerhalb des darstellbaren Bereichs führt, liefert sie eine Gleitkommazahl, die vom wirklichen Ergebnis so wenig wie möglich abweicht:

```
1.0 / 3.0
```
0.333333333333 : real

```
2.0 * 5.00000000001
```
10.0 : real

Man sagt, dass die Gleitkommaoperationen **Rundungsfehler** machen. Wenn mehrere Gleitkommaoperationen hintereinander ausgeführt werden, können sich die Rundungsfehler so akkumulieren, dass das Gleitkommaergebnis keine Ähnlichkeit mehr mit dem exakten Ergebnis hat.

Das Rechnen mit Gleitkommaoperationen bezeichnet man als **Gleitkommaarithmetik**. Gleitkommaarithmetik spielt für viele Anwendungen eine wichtige Rolle (z. B. Wettervorhersagen). Die *Numerik* ist eine eigenständige Forschungsrichtung, die sich mit Algorithmen beschäftigt, für deren Ergebnisse man trotz Rundungsfehlern eine Mindestgenauigkeit garantieren kann.

Aufgabe 1.28 Schreiben Sie eine rekursive Prozedur *power* : *real* ∗ *int* → *real*, die zu einer reellen Zahl x und einer natürlichen Zahl n die Potenz x^n mittels Gleitkommaoperationen berechnet. Welche Zahl liefert *power*(3.0, 100)? Handelt es sich dabei wirklich um die Zahl 3^{100}?

1.13.4 Beispiel: Newtonsches Verfahren

Als Beispiel betrachten wir das Newtonsche Verfahren zur Bestimmung reeller Quadratwurzeln. Sei x eine positive reelle Zahl. Dann definiert

$$a_0 = \frac{x}{2}$$
$$a_n = \frac{1}{2}(a_{n-1} + \frac{x}{a_{n-1}}) \quad \text{für n>0}$$

eine Folge $a_0, a_1, a_2, \ldots$ von Approximationen, die gegen $\sqrt{x}$ konvergieren. Die Berechnung der Approximationen erledigen wir mit der rekursiven Prozedur

```
fun newton (a:real, x:real, n:int) : real =
    if n<1 then a else newton (0.5*(a+x/a), x, n-1)
```
*val newton : real * real * int → real*

die zu a_m, x und n die Approximation a_{m+n} liefert. Damit deklarieren wir die Prozedur

```
fun sqrt (x:real) = newton (x/2.0, x , 5)
```
val sqrt : real → real

die zu x die Approximation a_5 liefert:

```
sqrt 4.0
```
2.0 : real

```
sqrt 2.0
```
1.41421356237 : real

```
sqrt 81.0
```
9.00000941552 : real

Aufgabe 1.29 Schreiben Sie eine Prozedur *sqrt* : *real* → *real*, die zu x die erste Approximation a_n liefert, die $\sqrt{x}$ mit der Genauigkeit $|x - a_n^2| < 10^{-4}$ approximiert. Schreiben Sie zusätzlich eine Prozedur *sqrt*′ : *real* → *real* ∗ *int*, die zu x das Paar (a_n, n) liefert, damit Sie sehen können, wie viele Approximationsschritte erforderlich waren.

Math.sqrt	: *real* → *real*	$\sqrt{x}$	
Math.sin	: *real* → *real*		
Math.asin	: *real* → *real*		
Math.exp	: *real* → *real*	e^x	
Math.pow	: *real* ∗ *real* → *real*	x^y	
Math.ln	: *real* → *real*		
Math.log10	: *real* → *real*		
Real.fromInt	: *int* → *real*		
Real.round	: *real* → *int*		
Real.floor	: *real* → *int*	$\lfloor x \rfloor$	*Rundung nach unten*
Real.ceil	: *real* → *int*	$\lceil x \rceil$	*Rundung nach oben*
Math.pi	: *real*	π	
Math.e	: *real*	e	

Abbildung 1.2: *Einige Standardprozeduren*

1.14 Standardstrukturen

Im Zusammenhang mit Zahlen benötigen wir eine Vielzahl von Prozeduren, die Standardaufgaben wie die Berechnung von Quadratwurzeln erledigen. Standard ML stellt solche **Standardprozeduren** im Rahmen sogenannter **Standardstrukturen** zur Verfügung. Abbildung 1.2 zeigt einige Prozeduren aus den Standardstrukturen *Math* und *Real*. Hier sind Anwendungsbeispiele:

```
Math.sqrt 4.0
2.0 : real

Real.fromInt 45
45.0 : real

Real.round 1.5
2 : int
```

Neben Prozeduren können Standardstrukturen auch andere Werte zur Verfügung stellen. Beispielsweise stellt die Standardstruktur *Math* unter dem Bezeichner *pi* eine Gleitkommazahl zur Verfügung, die die reelle Zahl π so gut wie möglich approximiert:

```
Math.pi
3.14159265359 : real
```

Die Objekte einer Standardstruktur werden durch **zusammengesetzte Bezeichner** bezeichnet (z.B. *Math.pi*), die aus dem Bezeichner der Struktur (z.B. *Math*) und aus dem Bezeichner des Objektes (z. B. *pi*) bestehen.

Eine Beschreibung der Standardstrukturen für Standard ML finden Sie im Web unter www.standardml.org/Basis.

Aufgabe 1.30 Deklarieren Sie Prozeduren des Typs *real* → *real*, die die folgenden Funktionen mit Gleitkommaoperationen berechnen:

a) $g(x) = 2x + 1,4e$

b) $h(x) = \sin x + \cos(2\pi x)$

Bemerkungen

Anhand einiger Beispiele haben Sie in diesem Kapitel einen ersten Einblick in die Programmierung bekommen. Sie wissen jetzt, was unter Deklarationen, Konditionalen, Tupeln und rekursiven Prozeduren zu verstehen ist. Außerdem können Sie kleine Programme schreiben und sie mithilfe eines Interpreters ausführen.

Mit rekursiven Prozeduren kann eine Vielzahl von Algorithmen formuliert werden. Unser erstes Beispiel war eine Prozedur für die Berechnung von Potenzen. Der Ausgangspunkt für die Konstruktion einer Prozedur ist die zu berechnende Funktion. Zunächst müssen Rekursionsgleichungen gefunden werden, mit denen die Funktion berechnet werden kann. Manchmal ist die Einführung von Hilfsfunktionen mit zusätzlichen Argumenten (sogenannte Akkus) erforderlich. Nachdem die richtigen Gleichungen gefunden sind, ist die Formulierung der Prozedur eine Routineangelegenheit.

Unter einer **Datenstruktur** versteht man ein Darstellungsformat für eine bestimmte Klasse von Objekten. Bisher haben wir einige einfache Datenstrukturen kennengelernt, die in Standard ML direkt verfügbar sind: Festkommazahlen, Gleitkommazahlen, Boolesche Werte und Tupel.

Eine Programmiersprache vermittelt zwischen Mensch und Maschine. Programme werden von Menschen geschrieben, damit sie von Maschinen ausgeführt werden können. Also müssen Programmiersprachen so entworfen werden, dass sie einerseits Menschen bei der Formulierung komplexer Datenstrukturen und Algorithmen unterstützen und andererseits maschinell ausführbar sind. Zu jeder Programmiersprache gehört eine gedankliche Welt, in der eine Vielzahl von gedanklichen Objekten existieren. Dazu gehören Programme, Deklarationen, Ausdrücke, Prozeduren, Tupel und Zahlen. Während der Ausführung eines Programms erlangen diese gedanklichen Objekte eine gewisse physikalische Präsenz.

Allgemein betrachtet geht es bei der Programmierung um die systematische Konstruktion komplexerer Objekte aus einfacheren Objekten. Programmiersprachen stellen den praktischen Rahmen für diesen Konstruktionsprozess dar. Da wir bisher nur sehr einfache Beispiele betrachtet haben, ist dieser wichtige Aspekt allerdings noch wenig sichtbar.

Dieses Kapitel endet so wie die nachfolgenden mit einem Verzeichnis, das die eingeführten Fachbegriffe auflistet. Zu jedem der Begriffe sollten Sie ein paar erklärende Worte sagen können und wissen, wo er im Kapitel erläutert wird. Wenn ein Begriff erstmals erläutert wird, erscheint er **fett gedruckt**.

Verzeichnis

Programme und Deklarationen; Bindung eines Bezeichners an einen Wert.

Wörter: Bezeichner, Konstanten, Operatoren, Schlüsselwörter.

Ausdrücke: Prozeduranwendungen, Konditionale (Bedingung, Konsequenz, Alternative).

Werte: Zahlen, Boolesche Werte, Tupel, Prozeduren.

Typen: *int*, *real*, *bool*, *unit*; Tupel- und Prozedurtypen.

Tupel: Paare, Tripel, leeres Tupel, Positionen, Komponenten, Stelligkeit, Projektionen, geschachtelte Tupel und Baumdarstellung, kartesische Muster.

Prozeduren: Argument und Ergebnis, Funktionen, Prozedurtypen, (kartesische) Argumentmuster, Argumentvariablen, Rumpf, Prozeduranwendungen und Prozeduraufrufe, lokale Deklarationen, Hilfsprozeduren.

Rekursive Prozeduren: Selbstanwendung, Ausführungsprotokoll, Rekursionsfolge, Rekursionsgleichungen, Endrekursion, Divergenz.

Natürliche Quadratwurzeln: Akkus.

Ganzzahlige Division: Div und Mod.

Interpreter: Ergebnisbezeichner, Fehlermeldungen, Debugging.

Ausführung: Reguläre Terminierung, Abbruch wegen Laufzeitfehler oder Speichererschöpfung, keine Terminierung.

Festkomma- und Gleitkommazahlen: Überlauf, Fakultäten, Gleitkommaarithmetik, Mantisse und Exponent, überladene Operatoren, Rundungsfehler, Newtonsches Verfahren.

Standardstrukturen: Standardprozeduren, zusammengesetzte Bezeichner.

Debugging.

Algorithmen und Datenstrukturen.

2 Programmiersprachliches

Wir gehen jetzt näher auf die sprachlichen Aspekte von Programmen ein. Dabei unterscheiden wir zwischen Syntax und Semantik. Bei der Syntax geht es um die Form und die Darstellung sprachlicher Objekte, bei der Semantik um deren Bedeutung und Ausführung.

2.1 Darstellung und Aufbau von Phrasen

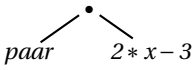
Ausdrücke, Typen, Deklarationen und Programme werden zusammenfassend als **Phrasen** bezeichnet.

Nach welchen Regeln werden Phrasen gebildet? Wie werden Phrasen mit Wörtern beschrieben? Welche Wörter gibt es? Wie werden Wörter mit Zeichen beschrieben? Diese Fragen gehören zu dem Themenkreis, der als Syntax bezeichnet wird.

Zunächst gehen wir genauer auf den Aufbau von Ausdrücken ein. Als Beispiel betrachten wir den Ausdruck

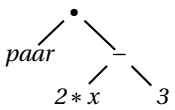
```
paar (2*x-3)
```

Dabei handelt es sich um eine Prozeduranwendung, die aus zwei Teilausdrücken gebildet ist. Grafisch können wir die Prozeduranwendung wie folgt darstellen:



Die zwei Teilausdrücke, aus denen die Prozeduranwendung gebildet ist, werden als die **Komponenten** der Prozeduranwendung bezeichnet.

Die linke Komponente der Prozeduranwendung ist ein Ausdruck, der nur aus einem Bezeichner besteht. Man spricht von einem **atomaren Ausdruck**. Dagegen ist die rechte Komponente der Prozeduranwendung wieder ein **zusammengesetzter Ausdruck**. Es handelt sich um eine Differenz, die aus dem Ausdruck $2 * x$, dem Operator $-$ und der Konstante 3 gebildet ist:



```
if x*y<15then(x,2+y)else paar(2*x-3)

if x * y < 15 then ( x , 2 + y ) else paar ( 2 * x - 3 )
```

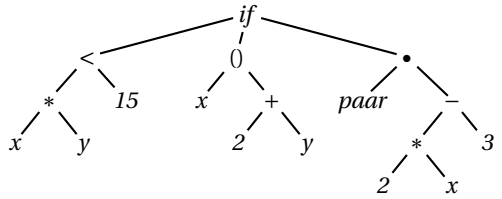
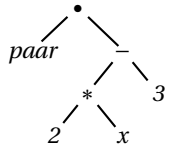


Abbildung 2.1: *Zeichen-, Wort- und Baumdarstellung eines Ausdrucks*

Die linke Komponente der Differenz ist ein Produkt, das aus der Konstante *2*, dem Operator $*$ und dem Bezeichner *x* gebildet ist:



Wir haben jetzt die **Baumdarstellung** des Ausdrucks $paar(2 * x - 3)$ vorliegen. Diese zeigt, wie der Ausdruck Schritt für Schritt aus einfacheren Ausdrücken gebildet ist. Am Anfang stehen dabei Bezeichner (*paar*, *x*) und Konstanten (*2*, *3*). Aus diesen atomaren Ausdrücken werden mithilfe der **syntaktischen Formen** für Operator- und Prozeduranwendungen zusammengesetzte Ausdrücke gebildet.

Ausdrücke sind abstrakte Objekte, die auf verschiedene Weise dargestellt werden können. Die übliche Darstellung durch Zeichen wird als **Zeichendarstellung** bezeichnet. Sie wird für die Interaktion mit dem Interpreter verwendet. Der Interpreter übersetzt eingegebene Zeichendarstellungen in Baumdarstellungen, die als Grundlage für die nachfolgenden Verarbeitungsschritte dienen.

Als Zwischenstufe für den Übergang von der Zeichen- zur Baumdarstellung gibt es die sogenannte **Wortdarstellung**. Diese beschreibt einen Ausdruck durch eine Folge von Wörtern. Die Wortdarstellung des Beispielausdrucks $paar(2 * x - 3)$ besteht aus 8 Wörtern:

```
paar  (  2  *  x  -  3  )
```

Die Zeichendarstellung kann die dargestellten Wörter durch **Leerzeichen** trennen. Zu den Leerzeichen gehören Zwischenraum, Zeilenwechsel und Tabulator. Das obige Beispiel zeigt, dass in vielen Fällen keine Trennung durch Leerzeichen erforderlich ist. Abbildung 2.1 zeigt die Zeichen-, Wort- und Baumdarstellung eines größeren Ausdrucks. Die Zeichendarstellung verwendet nur da Leerzeichen, wo sie erforderlich sind.

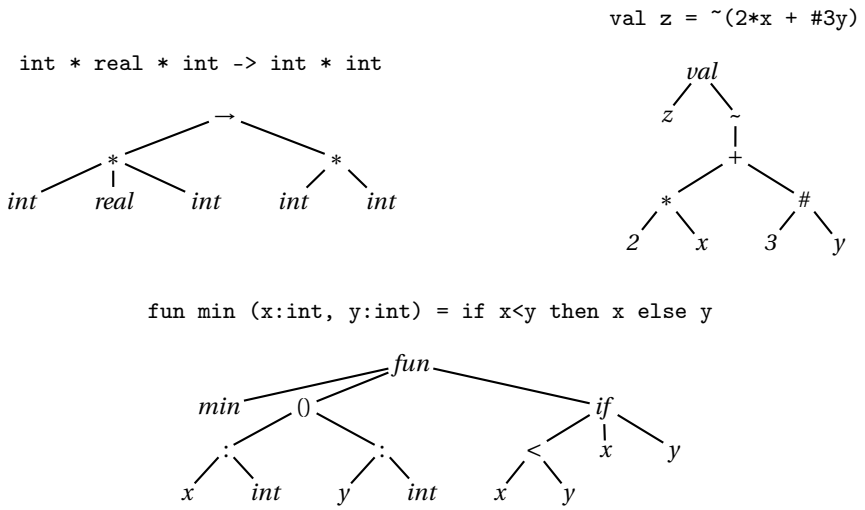


Abbildung 2.2: *Baumdarstellung von Typen und Deklarationen*

Leerzeichen darf man nur dann weglassen, wenn dadurch keine Worte verschmelzen. Wenn man beispielsweise bei "*paar 2*" das Leerzeichen weglässt, bekommt man statt einer Prozeduranwendung den Bezeichner "*paar2*". Ein weiteres Beispiel ist "*4+ ~5*". Mit dem Leerzeichen handelt es sich um die Zeichendarstellung einer Summe, ohne das Leerzeichen bekommen wir die Wortfolge

```
4 +~ 5
```

bei der es sich nicht um die Darstellung eines Ausdrucks handelt.

Unsere Ausführungen zu Aufbau und Darstellung von Ausdrücken gelten auch für andere Phrasen. Abbildung 2.2 zeigt Beispiele für die Baumdarstellung von Typen und Deklarationen.

Die Baumdarstellung einer Phrase stellt den hierarchischen Aufbau der Phrase direkt dar. Aus der Zeichendarstellung muss der Aufbau der Phrase dagegen erst gemäß den Regeln der Sprache erschlossen werden. Diese nicht immer einfache Aufgabe wird als **Syntaxanalyse** bezeichnet.

2.2 Syntaxübersicht

Wir geben jetzt eine Übersicht über die syntaktischen Objekte der bisher vorgestellten Teilsprache von Standard ML. Die Übersicht soll Ihnen im Folgenden als Referenz für syntaktische Fragen dienen.

2.2.1 Wörter

Wir haben die folgenden Wortarten kennengelernt:

- **Konstanten**. Beispiele für Konstanten sind *3*, *~56*, *1.7*, *1.7878E45*, *false*, *true* und ().
- **Operatoren**. Beispiele für zweistellige Operatoren sind +, −, ∗, /, *div*, *mod*, <, ≤, >, ≥, = und ≠. Außerdem haben wir den einstelligen Negationsoperator ~ kennengelernt. Die Vergleichsoperatoren ≤, ≥ und ≠ werden durch die Zeichenkombinationen <=, >= und <> dargestellt.
- **Schlüsselwörter**. Beispiele sind *val*, *fun*, *if*, *then*, *else*, *let*, *in*, *end*, *int*, *bool*, *unit*, =, (,), :, →, ∗ und #. Auch das Komma ist ein Schlüsselwort. Das Schlüsselwort → wird durch die Zeichenkombination -> dargestellt.
- **Bezeichner**. Bezeichner sind Wörter, die aus Buchstaben, Ziffern sowie den Zeichen "_" (Unterstrich) und "′" (Hochkomma) gebildet sind. Sie müssen mit einem Buchstaben beginnen. Außerdem gelten nur solche Wörter als Bezeichner, die nicht zu einer der bereits erwähnten Wortarten gehören. **Zusammengesetzte Bezeichner** bestehen aus zwei einfachen Bezeichnern, die durch einen Punkt verbunden sind, zum Beispiel "*Math.pi*".

Die Wörter = und ∗ spielen Doppelrollen, da sie als Operatoren und als Schlüsselwörter auftreten können. Betrachten Sie dazu die Deklaration

```
fun f (t: int*int*int) = if #1t=0 then #2t else 2*(#3t)
```

Vor dem Schlüsselwort *if* treten = und ∗ als Schlüsselwörter auf, danach als Operatoren.

2.2.2 Phrasen

Die Phrasen einer Sprache werden in **syntaktische Kategorien** unterteilt. Unter anderem gibt es bei unserer Teilsprache die Kategorien der Ausdrücke, Deklarationen und Typen. Außerdem unterscheidet man zwischen **atomaren** und **zusammengesetzten Phrasen**. Zusammengesetzte Phrasen werden mithilfe von einfacheren Phrasen gebildet. Die bei der Bildung einer zusammengesetzten Phrasen verwendeten Teilphrasen heißen **Komponenten** der zusammengesetzten Phrase.

Abbildungen 2.3 und 2.4 definieren die syntaktischen Kategorien der bisher betrachteten Teilsprache von Standard ML mithilfe von **syntaktischen Gleichungen**. Beispielsweise besagt die Gleichung für die syntaktische Kategorie ⟨*atomarer Ausdruck*⟩, dass ein atomarer Ausdruck entweder eine Konstante oder ein Bezeichner ist.

Die syntaktischen Gleichungen in Abbildung 2.3 und 2.4 legen die Baumdarstellungen der Phrasen bis auf unwesentliche Details fest. Beispielsweise besagt die Gleichung

⟨*Projektion*⟩ ::= # ⟨*positive ganze Zahl*⟩ ⟨*Ausdruck*⟩

dass die Projektion *#5 x* durch den Baum

⟨*Anwendung*⟩ ::=
⟨*Operatoranwendung*⟩
| ⟨*Prozeduranwendung*⟩
| ⟨*Projektion*⟩

⟨*atomarer Ausdruck*⟩ ::=
⟨*Konstante*⟩
| ⟨*Bezeichner*⟩

⟨*Ausdruck*⟩ ::=
⟨*atomarer Ausdruck*⟩
| ⟨*Anwendung*⟩
| ⟨*Konditional*⟩
| ⟨*Tupelausdruck*⟩
| ⟨*Let-Ausdruck*⟩

⟨*Konditional*⟩[1] ::= *if* ⟨*Ausdruck*⟩ *then* ⟨*Ausdruck*⟩ *else* ⟨*Ausdruck*⟩

⟨*Let-Ausdruck*⟩ ::= *let* ⟨*Programm*⟩ *in* ⟨*Ausdruck*⟩ *end*

⟨*Operatoranwendung*⟩ ::=
⟨*einstelliger Operator*⟩ ⟨*Ausdruck*⟩
| ⟨*Ausdruck*⟩ ⟨*zweistelliger Operator*⟩ ⟨*Ausdruck*⟩

⟨*Projektion*⟩ ::= # ⟨*positive ganze Zahl*⟩ ⟨*Ausdruck*⟩

⟨*Prozeduranwendung*⟩ ::= ⟨*Ausdruck*⟩ ⟨*Ausdruck*⟩

⟨*Tupelausdruck*⟩[2] ::= (⟨*Ausdruck*⟩ , … , ⟨*Ausdruck*⟩)

1. Die drei Komponenten eines Konditionals heißen **Bedingung**, **Konsequenz** und **Alternative**.
2. Tupelausdrücke müssen mit mindestens zwei Komponenten gebildet werden.

Abbildung 2.3: *Syntaktische Gleichungen für Ausdrücke*

⟨*Argumentmuster*⟩[1] ::= (⟨*Argumentspezifikation*⟩ , … , ⟨*Argumentspezifikation*⟩)

⟨*Argumentspezifikation*⟩ ::= ⟨*Bezeichner*⟩ : ⟨*Typ*⟩

⟨*atomarer Typ*⟩ ::=
int
| *real*
| *bool*
| *unit*

⟨*Deklaration*⟩ ::=
⟨*Val-Deklaration*⟩
| ⟨*Prozedurdeklaration*⟩

⟨*Programm*⟩ ::= ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩

⟨*Prozedurdeklaration*⟩[2] ::=
fun ⟨*Bezeichner*⟩ ⟨*Argumentmuster*⟩ = ⟨*Ausdruck*⟩
| *fun* ⟨*Bezeichner*⟩ ⟨*Argumentmuster*⟩ : ⟨*Typ*⟩ = ⟨*Ausdruck*⟩

⟨*Prozedurtyp*⟩ ::= ⟨*Typ*⟩ → ⟨*Typ*⟩

⟨*Tupeltyp*⟩ ::= ⟨*Typ*⟩ * … * ⟨*Typ*⟩

⟨*Typ*⟩ ::=
⟨*atomarer Typ*⟩
| ⟨*Prozedurtyp*⟩
| ⟨*Tupeltyp*⟩

⟨*Val-Deklaration*⟩ ::= *val* ⟨*Val-Muster*⟩ = ⟨*Ausdruck*⟩

⟨*Val-Muster*⟩[1,3] ::=
⟨*Bezeichner*⟩
| (⟨*Bezeichner*⟩ , … , ⟨*Bezeichner*⟩)

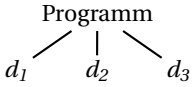
1. Die durch ein Muster eingeführten Bezeichner werden als **Variablen** bezeichnet. Eine Variable darf in einem Muster nicht mehrfach vorkommen.
2. Wir unterscheiden zwischen dem **Kopf** (dem Teil vor =) und dem **Rumpf** (dem Teil nach =) einer Prozedurdeklaration. Der optionale Typ nach dem Argumentmuster heißt **Ergebnistyp**.
3. Val-Muster der Form (⟨*Bezeichner*⟩,…,⟨*Bezeichner*⟩) heißen **kartesische Muster**.

Abbildung 2.4: *Syntaktische Gleichungen für Deklarationen*

dargestellt wird. Ein zweites Beispiel ist die Gleichung

⟨*Programm*⟩ ::= ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩

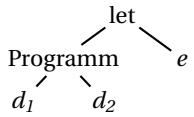
gemäß der ein aus drei Deklarationen bestehendes Programm $d_1\ d_2\ d_3$ durch den Baum



dargestellt wird. Schließlich betrachten wir noch Let-Ausdrücke, deren syntaktische Form durch die Gleichung

⟨*Let-Ausdruck*⟩ ::= *let* ⟨*Programm*⟩ *in* ⟨*Ausdruck*⟩ *end*

gegeben ist. Gemäß dieser Gleichung und der Gleichung für Programme hat ein Let-Ausdruck *let* $d_1\ d_2$ *in e end* die folgende Baumdarstellung:



Beachten Sie, dass von den für die Wortdarstellung von Let-Ausdrücken erforderlichen Schlüsselwörtern *let*, *in* und *end* nur das einleitende Schlüsselwort *let* in der Baumdarstellung vorkommt.

Aufgabe 2.1 Geben Sie die Baumdarstellungen der folgenden durch Zeichendarstellungen beschriebenen Phrasen an.

a) `int * int -> bool`

b) `if x<3 then 3 else p 3`

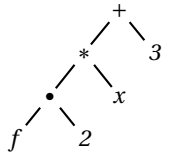
c) `let val x = 2+y in x-y end`

d) `fun p(x:int,n:int):int=if n>0then x*p(x,n-1)else 1`

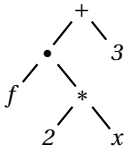
2.3 Klammern

Eine Folge von Wörtern, die einen Ausdruck oder einen Typ beschreibt, kann stets in Klammern eingeschlossen werden. Die dadurch gegebene Gruppierungsmöglichkeit versetzt uns in die Lage, alle Phrasen eindeutig durch Wortdarstellungen zu beschreiben.

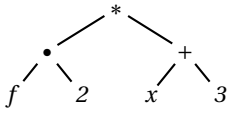
Hier sind Beispiele:



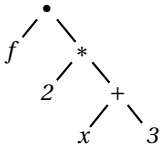
$f\ 2 * x + 3$



$f(2 * x) + 3$



$(f\ 2) * (x + 3)$



$f(2 * (x + 3))$

Wenn Sie wollen, können Sie **überflüssige Klammern** setzen. Beispielsweise beschreiben die Zeichendarstellungen

```
2*f~2+3    2*(f~2)+3    (2*(f~2))+3    ((2*(f~2))+3)
```

alle den gleichen Ausdruck.

Standard ML ist mit diversen **Klammersparregeln** ausgestattet. Hier sind Beispiele für die wichtigsten:

$3 * 4 + 5$	$\rightsquigarrow$	$(3 * 4) + 5$	Punkt vor Strich
$2 + 3 + 4$	$\rightsquigarrow$	$(2 + 3) + 4$	Operatoranwendung klammert links
$2 - 3 + 4$	$\rightsquigarrow$	$(2 - 3) + 4$	
$f\ 3 + 4$	$\rightsquigarrow$	$(f\ 3) + 4$	Prozeduranwendung vor Operatoranwendung
$f\ g\ 3$	$\rightsquigarrow$	$(f\ g)\ 3$	Prozeduranwendung klammert links

Die letzte Regel wird Ihnen erst einleuchten, wenn Sie mehr von Standard ML gesehen haben (siehe kaskadierte Prozeduren in § 3.1). Wir erwähnen sie aber bereits hier, damit Sie nicht zu viele Klammern weglassen. Wenn Sie es versehentlich doch tun, wird der Interpreter das mit vorerst für Sie undurchschaubaren Fehlermeldungen quittieren.

Auch für Typen gibt es eine Klammersparregel:

$int * int \rightarrow int * int \quad \rightsquigarrow \quad (int * int) \rightarrow (int * int)$ Stern vor Pfeil

Eine Zusammenstellung aller Klammersparregeln finden Sie in Anhang A (S. 353).

Merken Sie sich die folgenden Tatsachen zur Darstellung von Phrasen:

1. Jede Phrase hat genau eine Baumdarstellung.
2. Eine Phrase kann mehrere Wortdarstellungen haben, die sich durch das Weglassen oder Hinzufügen von überflüssigen Klammerpaaren unterscheiden.
3. Verschiedene Phrasen derselben syntaktischen Kategorie haben stets verschiedene Wortdarstellungen.

Aufgabe 2.2 Geben Sie die Baumdarstellungen der folgenden durch Zeichendarstellungen beschriebenen Phrasen an.

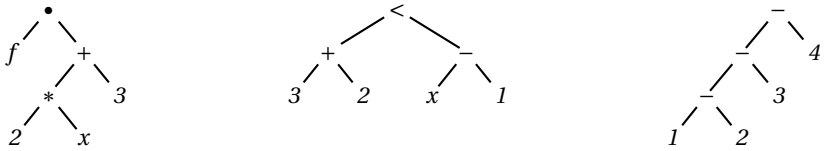
a) `x+3*f x-4`

b) `1+2+3+4`

c) `1+2*x-y*3+4`

d) `int*(int*int)*int->int*int`

Aufgabe 2.3 Geben Sie für die folgenden Ausdrücke minimal geklammerte Zeichendarstellungen an:



Sie können die möglichen Zeichendarstellungen durch gezieltes Experimentieren mit einem Interpreter ermitteln.

2.4 Freie Bezeichner und Umgebungen

Der Ausdruck $x+y$ beschreibt erst dann einen Wert, wenn wir Werte für die Bezeichner x und y vorgeben. Wir sagen, dass x und y in dem Ausdruck **frei auftreten**, oder dass x und y **freie Bezeichner** des Ausdrucks sind.

Eine **Umgebung** ist eine Sammlung von Bezeichnerbindungen. Beispielsweise **bindet** die Umgebung

$$[x := 5,\ y := 7]$$

den Bezeichner x an den Wert 5 und den Bezeichner y an den Wert 7. Bezüglich dieser Umgebung beschreibt der Ausdruck $x+y$ den Wert 12. Wir sprechen vom **Wert eines Ausdrucks in einer Umgebung**.

In der Deklaration

```
fun q (x:int) = 3+(p x)
```

kommt der Bezeichner p frei vor. Die Deklaration kann daher nur dann ausgeführt werden, wenn eine Bindung für p bereitgestellt wird.

Generell versteht man unter den **freien Bezeichnern einer Phrase** diejenigen Bezeichner, die in der Phrase ein Auftreten haben, das nicht im Rahmen der Phrase gebunden ist. Phrasen ohne freie Bezeichner bezeichnet man als **geschlossen** und Phrasen mit freien Bezeichnern als **offen**.

Betrachten Sie das Programm

```
fun p (x:int) = x
fun q (x:int) = 3+(p x)
```

Dieses Programm ist geschlossen. Die zweite Deklaration des Programms ist für sich alleine genommen offen, da der Bezeichner *p* in der Deklaration frei auftritt. Dagegen ist die erste Deklaration des Programms geschlossen.

Hier ist ein interessantes Beispiel für eine offene Deklaration:

```
val x = x
```

Diese Deklaration hat den freien Bezeichner *x*, da die durch die Deklaration eingeführte Bindung nicht innerhalb der Deklaration gilt. Dagegen ist die rekursive Prozedurdeklaration

```
fun f (x:int) : int = if x<1 then 1 else x*f(x-1)
```

geschlossen, da die durch die Deklaration eingeführte Bindung für *f* auch innerhalb der Deklaration gilt.

2.5 Tripeldarstellung von Prozeduren

Für sich alleine genommen beschreibt die Deklaration

```
fun p (x:int) = x+a
```

keine Prozedur, da der Wert für den freien Bezeichner *a* fehlt. Für die Darstellung einer Prozedur benötigen wir also neben dem sogenannten **Code** (eine Prozedurdeklaration) zusätzlich eine Umgebung, die die freien Bezeichner des Codes bindet. Darüber hinaus ist es hilfreich, den Typ der Prozedur zur Verfügung zu haben. Im Folgenden werden wir Prozeduren also gemäß der Gleichung

Prozedur = (Code, Typ, Umgebung)

darstellen. Wir sprechen von der **Tripeldarstellung einer Prozedur**.

Als Beispiel betrachten wir das Programm

```
val a = 2*7
fun p (x:int) = x+a
```

Seine Ausführung bindet den Bezeichner *p* an die Prozedur

$$(\underbrace{fun\ p\ x = x + a}_{\text{Code}},\ \underbrace{int \to int}_{\text{Typ}},\ \underbrace{[a := 14]}_{\text{Umgebung}})$$

Im Code der Prozedur verzichten wir auf die Typangabe für die Argumentvariable, da diese Information als Teil des Prozedurtyps dargestellt wird.

Unser zweites Beispiel ist das Programm

```
val a = 2*7
fun p (x:int) = x+a
fun q (x:int) = x + p x
```

Seine Ausführung liefert die folgenden Bezeichnerbindungen:

$$\begin{aligned} a &:= 14 \\ p &:= (\mathit{fun}\ p\ x = x + a,\ \mathit{int} \to \mathit{int},\ [a := 14]) \\ q &:= (\mathit{fun}\ q\ x = x + p\ x,\ \mathit{int} \to \mathit{int}, \\ &\qquad [p := (\mathit{fun}\ p\ x = x + a,\ \mathit{int} \to \mathit{int},\ [a := 14])]) \end{aligned}$$

Beachten Sie, dass die Darstellung der Prozedur q in sich abgeschlossen ist, da sie die Darstellung der Prozedur p enthält.

Betrachten Sie jetzt das Programm

```
fun p (x:int) = x
fun q (x:int) = p x
fun p (x:int) = 2*x
val a = (p 5, q 5)
```

Da die Prozedur q mit der ersten Bindung von p gebildet wird, kommt diese auch bei der Anwendung von q in der vierten Zeile zum Einsatz, obwohl an dieser Stelle bereits eine andere Bindung für p vorliegt. Das bedeutet, dass a an $(10,5)$ gebunden wird. Insgesamt liefert das obige Programm die folgenden Bindungen:

$$\begin{aligned} p &:= (\mathit{fun}\ p\ x = 2 * x,\ \mathit{int} \to \mathit{int},\ []) \\ q &:= (\mathit{fun}\ q\ x = p\ x,\ \mathit{int} \to \mathit{int},\ [p := (\mathit{fun}\ p\ x = x,\ \mathit{int} \to \mathit{int},\ [])]) \\ a &:= (10,5) \end{aligned}$$

Die durch die erste Deklaration für p eingeführte Bindung überlebt also in der Umgebung der an q gebundenen Prozedur.

Wir halten fest, dass der Rumpf einer Prozedur immer mit den Bindungen arbeitet, die bei der Ausführung der Deklaration der Prozedur vorlagen. Diese Konvention wird als **statisches** oder **lexikalisches Bindungsprinzip** bezeichnet.

Aufgabe 2.4 Welche Bindungen berechnet das folgende Programm?

```
fun f (x:bool) = if x then 1 else 0
val x = 5*7
fun g (z:int) = f(z<x)<x
val x = g 5
```

2.6 Semantische Zulässigkeit

Bevor ein Interpreter ein Programm ausführt, prüft er zunächst, ob das Programm **semantisch zulässig** ist. Die damit verbundenen Bedingungen betreffen die Bindung von Bezeichnern und den typgerechten Aufbau von Ausdrücken.

Zunächst muss ein semantisch zulässiges Programm geschlossen sein (siehe § 2.4). Das heißt, alle in einem Programm vorkommenden Bezeichnerauftreten müssen innerhalb des Programms gebunden sein. Beispielsweise ist das Programm

```
fun f (x:int) : int = if x<y then 1 else x*f(x-1)
```

unzulässig, da das Auftreten des Bezeichners *y* in der Bedingung des Konditionals ungebunden ist. Wenn wir eine Deklaration für *y* hinzufügen

```
val y = 1
fun f (x:int) : int = if x<y then 1 else x*f(x-1)
```

bekommen wir ein geschlossenes Programm, in dem alle Bezeichnerauftreten ordnungsgemäß gebunden sind.

Die Regeln für den typgerechten Aufbau von Ausdrücken stellen sicher, dass Operationen bei der Ausführung nur auf typgerechte Argumente angewendet werden. Damit wird beispielsweise ausgeschlossen, dass eine Addition auf ein Tupel oder eine Projektion auf eine Zahl angewendet wird.

Typgerecht aufgebaute Phrasen bezeichnen wir als **wohlgetypt**. Bei der Typprüfung einer Phrase müssen für die freien Bezeichner Typen vorgegeben werden. Zu einem wohlgetypten Ausdruck kann ein Typ bestimmt werden. Beispielsweise hat der Ausdruck $x + y$ den Typ *real*, wenn wir für x und y jeweils den Typ *real* vorgeben.

Hier ist die **Typregel** für die Anwendung zweistelliger Operatoren:

$$\frac{e_1 : t_1 \qquad o : t_1 * t_2 \to t \qquad e_2 : t_2}{e_1\ o\ e_2 \ :\ t}$$

Die Regel besagt, dass eine Anwendung $e_1\ o\ e_2$ wohlgetypt ist und den Typ t hat, wenn die folgenden Bedingungen erfüllt sind:

1. Der linke Teilausdruck e_1 hat den Typ t_1.
2. Der Operator o hat den Typ $t_1 * t_2 \to t$.
3. Der rechte Teilausdruck e_2 hat den Typ t_2.

Als Beispiel betrachten wir den Ausdruck $x + 5$ und nehmen an, dass der Bezeichner x den Typ *int* hat. Da der Operator + den Typ $int * int \to int$ und die Konstante 5 den Typ *int* hat, folgt mit der obigen Typregel, dass der Ausdruck wohlgetypt ist und den Typ *int* hat.

Hier ist die Typregel für Konditionale:

$$\frac{e_1 : \mathit{bool} \quad e_2 : t \quad e_3 : t}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 :\ t}$$

Die Regel verlangt, dass die Bedingung des Konditionals den Typ *bool* hat. Weiter verlangt die Regel, dass die Konsequenz des Konditionals denselben Typ wie die Alternative des Konditionals hat. Wenn diese Bedingungen erfüllt sind, ordnet die Regel dem Konditional den gemeinsamen Typ der Konsequenz und Alternative zu.

Sie wissen jetzt genug, um die Typregeln für Tupelausdrücke und Prozeduranwendungen zu verstehen:

$$\frac{e_1 : t_1 \quad \cdots \quad e_n : t_n}{(e_1, \ldots, e_n) :\ t_1 * \cdots * t_n} \qquad \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{e_1\ e_2 :\ t_2}$$

2.7 Ausführung

Wir beschreiben jetzt so genau wie möglich, wie die folgenden Ausführungsprobleme gelöst werden können:

- *Ausführung von Ausdrücken.* Gegeben einen Ausdruck e und eine Umgebung V, bestimme den Wert, den e für V liefert.
- *Ausführung von Prozeduraufrufen.* Gegeben eine Prozedur p und einen Wert v, bestimme den Wert, den p für v liefert.
- *Ausführung von Deklarationen.* Gegeben eine Deklaration D und eine Umgebung V, bestimme die Umgebung, die D für V liefert.
- *Ausführung von Programmen.* Gegeben ein Programm P und eine Umgebung V, bestimme die Umgebung, die P für V liefert.

Wie wir bald sehen werden, bestehen zwischen diesen Problemen zirkuläre Abhängigkeiten.

Wir halten fest, dass wir die folgenden Arten von Werten betrachten: Zahlen, Boolesche Werte, Tupel, und Prozeduren.

Für Umgebungen benötigen wir eine als **Adjunktion** bezeichnete Operation $V_1 + V_2$, die zwei Umgebungen V_1, V_2 zu einer Umgebung kombiniert:

$$[x := 5,\ y := 7] \ +\ [x := 1,\ z := 3] \ =\ [x := 1,\ y := 7,\ z := 3]$$

$$[x := 1,\ z := 3] \ +\ [x := 5,\ y := 7] \ =\ [x := 5,\ y := 7,\ z := 3]$$

Falls ein Bezeichner von beiden Umgebungen gebunden wird, wird er von der adjungierten Umgebung $V_1 + V_2$ gemäß V_2 gebunden. Wir sagen auch, dass die Bindungen der rechten Umgebung die Bindungen der linken Umgebung überschreiben. Im Gegensatz zur Addition von Zahlen ist die Adjunktion von Umgebungen also keine kommutative Operation.

2.7.1 Ausführung von Ausdrücken

Die Ausführung eines Ausdrucks in einer Umgebung V erfolgt je nach Form des Ausdrucks gemäß einer der folgenden Regeln:

1. Die Ausführung eines Ausdrucks, der nur aus einer Konstante besteht, liefert den durch die Konstante bezeichneten Wert.
2. Die Ausführung eines Ausdrucks, der nur aus einem Bezeichner besteht, liefert den von V für den Bezeichner vorgegebenen Wert.
3. Die Ausführung einer einstelligen Operatoranwendung $o\ e$ beginnt mit der Ausführung des Teilausdrucks e in V. Wenn diese den Wert v liefert, liefert die Anwendung den Wert, den die durch den Operator o bezeichnete Operation für v liefert.
4. Die Ausführung einer zweistelligen Operatoranwendung $e_1\ o\ e_2$ beginnt mit der Ausführung der Teilausdrücke e_1 und e_2 in V. Wenn diese die Werte v_1 und v_2 liefert, liefert die Anwendung den Wert, den die durch den Operator o bezeichnete Operation für v_1 und v_2 liefert.
5. Die Ausführung einer Projektion $\#k\ e$ beginnt mit der Ausführung des Teilausdrucks e in V. Wenn diese ein Tupel mit mindestens k Positionen liefert, liefert die Projektion die k-te Komponente des Tupels.
6. Die Ausführung einer Prozeduranwendung $e_1\ e_2$ beginnt mit der Ausführung der Teilausdrücke e_1 und e_2 in V. Wenn diese die Prozedur p und den Wert v liefern, wird der Prozeduraufruf $p\ v$ ausgeführt. Die Prozeduranwendung liefert den so erhaltenen Wert.
7. Die Ausführung eines Konditionals *if* e_1 *then* e_2 *else* e_3 beginnt mit der Ausführung der Bedingung e_1 in V. Wenn e_1 den Wert *true* liefert, wird die Konsequenz e_2 in V ausgeführt und das Konditional liefert den so erhaltenen Wert. Wenn e_1 den Wert *false* liefert, wird die Alternative e_3 in V ausgeführt und das Konditional liefert den so erhaltenen Wert.
8. Die Ausführung eines Tupelausdrucks $(e_1, \ldots, e_n)$ beginnt mit der Ausführung der Teilausdrücke $e_1, \ldots, e_n$ in V. Wenn diese die Werte $v_1, \ldots, v_n$ liefern, liefert der Tupelausdruck das Tupel $(v_1, \ldots, v_n)$.
9. Die Ausführung eines Let-Ausdrucks *let P in e end* beginnt mit der Ausführung des Programms P in V. Wenn diese die Umgebung V' liefert, wird der Ausdruck e in V' ausgeführt. Der Let-Ausdruck liefert den so erhaltenen Wert.

2.7.2 Ausführung von Prozeduraufrufen

Bei einem Aufruf einer Prozedur $p = (\mathit{fun}\ f\ M = e,\ t,\ V)$ mit einem Wert v wird zuerst die Umgebung V' bestimmt, die die Variablen des Musters M gemäß dem Argument v bindet. Dann wird der Prozedurrumpf e in der Umgebung $(V + [f := p]) + V'$ ausgeführt. Der Prozeduraufruf liefert den so erhaltenen Wert. Die Bindung $f := p$ ermöglicht rekursive Prozeduraufrufe.

2.7.3 Ausführung von Deklarationen

Deklarationen werden wie folgt in einer Umgebung V ausgeführt:

1. Bei einer Deklaration $val\ M = e$ wird zuerst der Ausdruck e in V ausgeführt. Wenn das den Wert v liefert, wird die Umgebung V' bestimmt, die die Variablen des Musters M gemäß v bindet. Die Deklaration liefert dann die Umgebung $V + V'$.
2. Eine Prozedurdeklaration $fun\ f\ M = e$ oder $fun\ f\ M : t = e$ liefert die Umgebung

   $$V + [\, f := (fun\ f\ M' = e,\ t',\ V')]$$

   wobei M', t' und V' wie folgt bestimmt sind: M' ergibt sich aus M durch Löschen der Typen; t' ist der für die deklarierte Prozedur ermittelte Typ; und die Umgebung V' besteht aus den Bindungen von V, die die freien Bezeichner der Prozedurdeklaration binden.

2.7.4 Ausführung von Programmen

Die Ausführung eines Programms in einer Umgebung V geschieht wie folgt:

1. Wenn das Programm leer ist, wird die Umgebung V geliefert.
2. Wenn das Programm die Form $D\ P$ hat, wird zunächst die Deklaration D in V ausgeführt. Wenn das die Umgebung V' liefert, wird das Restprogramm P in V' ausgeführt und das Programm liefert die so erhaltene Umgebung.

Aufgabe 2.5 Machen Sie sich den Unterschied zwischen einer Prozeduranwendung und einem Prozeduraufruf klar. Hilfestellung: Bei Prozeduranwendungen handelt es sich um syntaktische Objekte. Bei Prozeduraufrufen handelt es sich dagegen um semantische Objekte, die für die Beschreibung der Ausführung von Prozeduranwendungen benötigt werden.

Aufgabe 2.6 Betrachten Sie das folgende Programm:

```
val x = 3+2
fun f (y:int) = x+y
fun g (y:int) : int = if y<x then 0 else y+g(y-1)
```

a) Geben Sie die Umgebung an, die die Ausführung des Programms in der Umgebung [] liefert (§ 2.7.4).

b) Geben Sie die Umgebung an, in der der Rumpf der Prozedur f bei der Ausführung des Aufrufs $f\,7$ ausgeführt wird (§ 2.7.2).

c) Geben Sie die Umgebung an, in der der Rumpf der Prozedur g bei der Ausführung des Aufrufs $g\,13$ ausgeführt wird (§ 2.7.2).

Aufgabe 2.7 Geben Sie einen geschlossenen Ausdruck an, der eine Prozedur $int \to int$ beschreibt, die zu x das Ergebnis x^2 liefert. Geben Sie die Tripeldarstellung der durch Ihren Ausdruck beschriebenen Prozedur an. Hinweis: Verwenden Sie einen Let-Ausdruck.

2.8 Verarbeitungsphasen eines Interpreters

Wir sehen uns jetzt genauer an, wie ein Interpreter ein Programm verarbeitet. Bisher wissen wir in etwa das Folgende. Als Eingabe bekommt der Interpreter eine Folge von Zeichen. Er prüft dann, ob es sich dabei um die Darstellung eines semantisch zulässigen Programms handelt. Wenn dies der Fall ist, führt er das Programm aus und teilt dem Benutzer die Ergebnisse mit.

Bei genauerer Betrachtung können wir vier aufeinanderfolgende Verarbeitungsphasen unterscheiden:

1. **Lexikalische Analyse.** Prüfe, ob die eingegebene Zeichenfolge als eine Folge von Wörtern aufgefasst werden kann.
2. **Syntaktische Analyse.** Prüfe, ob die von der lexikalischen Analyse gelieferte Wortfolge ein Programm beschreibt. Statt von syntaktischer Analyse spricht man auch von *Parsing*.
3. **Semantische Analyse.** Prüfe, ob das von der syntaktischen Analyse gelieferte Programm semantisch zulässig ist. Man bezeichnet dieses Vorgehen auch als *Elaboration*.
4. **Ausführung.** Führe das Programm aus. Diese Phase wird auch *Auswertung* oder *Evaluation* genannt.

In jeder Phase können Fehler entdeckt werden. Eine Phase wird nur dann ausgeführt, wenn bis dahin keine Fehler entdeckt wurden.

Man unterscheidet zwischen statischen und dynamischen Aspekten von Programmiersprachen. Die die Analysephasen betreffenden Aspekte bezeichnet man als **statisch** und die die Ausführungsphase betreffenden als **dynamisch**.

Die **Syntax einer Programmiersprache** legt die Regeln für die lexikalische und syntaktische Analyse fest. Entsprechend unterscheidet man zwischen der lexikalischen und der phrasalen Syntax einer Programmiersprache. Die **lexikalische Syntax** regelt, wie Wörter aus Zeichen gebildet werden, und die **phrasale Syntax** regelt, wie Phrasen aus Wörtern gebildet werden.

Die **Semantik einer Programmiersprache** legt die Regeln für die semantische Analyse und die Ausführung fest. Entsprechend unterscheidet man zwischen der statischen und der dynamischen Semantik einer Programmiersprache. Die **statische Semantik** formuliert Bedingungen, die semantisch zulässige Programme erfüllen müssen (beispielsweise Wohlgetyptheit). Die **dynamische Semantik** beschreibt, welche Ergebnisse die Ausführung von Programmen liefern soll.

Abbildung 2.5 veranschaulicht den Zusammenhang zwischen den Verarbeitungsphasen eines Interpreters und den verschiedenen Teilen der Syntax und Semantik graphisch.

Man unterscheidet zwischen **syntaktischen und semantischen Objekten.** Wörter und Phrasen sind syntaktische Objekte, Werte und Umgebungen sind semantische Objekte.

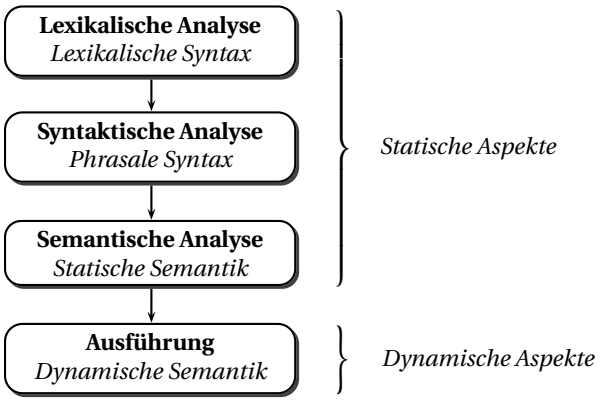


Abbildung 2.5: *Verarbeitungsphasen eines Interpreters*

Man unterscheidet auch zwischen **atomaren und zusammengesetzten Objekten**. Die unmittelbaren Teilobjekte eines zusammengesetzten Objekts bezeichnet man als seine **Komponenten**.

Die Syntax einer Programmiersprache regelt die äußere Form der Sprache. Die Semantik einer Programmiersprache ist mit der Bedeutung der syntaktischen Objekte der Sprache befasst.

2.9 Semantische Äquivalenz

Wir bezeichnen zwei Phrasen als **semantisch äquivalent**, wenn sie sich in Bezug auf die statische und dynamische Semantik nach außen hin gleich verhalten. Beispielsweise sind die Ausdrücke $x+x$ und $2*x$ für $x:int$ semantisch äquivalent. Auch die Prozedurdeklarationen

```
fun p (x:int, y:int) = x+y
fun p (y:int, x:int) = x+y
```

sind semantisch äquivalent.

Die Semantik von Programmiersprachen ist in Bezug auf einen abstrakten Interpreter definiert, der keiner Speicherbeschränkung unterliegt. Weiter nehmen wir an, dass mit beliebig großen ganzen Zahlen gerechnet werden kann. Semantische Äquivalenz ist in Bezug auf diese idealisierte Semantik definiert. Das sorgt dafür, dass die Ausdrücke

$$(x-y)*(x-y)$$

und

$$x*x-2*x*y+y*y$$

für $x, y : int$ semantisch äquivalent sind. Auf einem konkreten Interpreter werden sich die beiden Ausdrücke jedoch unterschiedlich verhalten, wenn man für x und y genügend große Zahlen wählt:

```
val x = 999999999
val y = x
val z = (x-y)*(x-y)
```
z = 0 : int

```
val z = x*x - 2*x*y + y*y
```
! Uncaught exception: Overflow

Zwei Prozeduren werden als **semantisch äquivalent** bezeichnet, wenn sie den gleichen Typ haben und für dieselben Argumente dieselben Ergebnisse liefern. Beispielsweise liefern die folgenden Deklarationen semantisch äquivalente Prozeduren:

```
fun p (x:int, y:int) = (x+y)*(x+y)
fun q (x:int, y:int) = x*x + 2*x*y + y*y
```

Bemerkungen

In diesem Kapitel haben wir uns eingehend mit der Syntax und Semantik programmiersprachlicher Objekte beschäftigt. Die Syntax regelt den Aufbau und die Darstellung von Phrasen. Die zweidimensionale Baumdarstellung stellt den Aufbau einer Phrase direkt dar. Bei der eindimensionalen Zeichendarstellung muss der Aufbau der dargestellten Phrase dagegen erst erschlossen werden. Im Allgemeinen müssen Klammern eingefügt werden, um für eine Phrase eine eindeutige Zeichendarstellung zu erhalten. Die statische Semantik regelt die Bindung von Bezeichnern, den typgerechten Aufbau von Phrasen und die Inferenz fehlender Typangaben. Die dynamische Semantik regelt, welche Ergebnisse Phrasen bei der Ausführung liefern sollen.

Verzeichnis

Syntax: Baum-, Wort- und Zeichendarstellung von Phrasen; lexikalische und phrasale Syntax; syntaktische Formen; Leerzeichen; Klammersparregeln; überflüssige Klammern.

Semantik: statische und dynamische; semantische Zulässigkeit; Bezeichnerbindungen, Umgebungen, Adjunktion; Ausführung oder Auswertung von Ausdrücken, Prozeduraufrufen, Deklarationen und Programmen; Wert eines Ausdrucks in einer Umgebung; semantische Äquivalenz.

Verarbeitungsphasen eines Interpreters: lexikalische, syntaktische und semantische Analyse; Ausführung.

Objekte: atomare, zusammengesetzte; Komponenten; syntaktische, semantische; statische, dynamische.

Wörter: Konstanten, Operatoren, Bezeichner, Schlüsselwörter.

Phrasen: Ausdrücke, Typen, Muster, Deklarationen, Programme; offene, geschlossene; wohlgetypte, semantisch zulässige; syntaktische Kategorien und syntaktische Gleichungen.

Ausdrücke: atomare; Anwendungen (Operator- und Prozeduranwendungen, Projektionen); Konditionale; Tupel- und Let-Ausdrücke.

Typen: *int*, *real*, *bool*, *unit*; Tupel- und Prozedurtypen; Typregeln.

Deklarationen: Kopf, Rumpf, deklarierte Bezeichner; Ergebnistyp bei rekursiven Prozedurdeklarationen; (kartesische) Val- und Argumentmuster, Variablen.

Werte: Zahlen, Boolesche Werte, Tupel, Prozeduren.

Tripeldarstellung von Prozeduren: Code, Typ, Umgebung.

Lexikalisches Bindungsprinzip.

3 Höherstufige Prozeduren

Viele Berechnungsaufgaben können durch Rückführung auf bekannte Berechnungstechniken gelöst werden. In diesem Kapitel lernen Sie höherstufige und polymorphe Prozeduren kennen, mit denen Berechnungstechniken programmiersprachlich formuliert werden können. Außerdem behandeln wir den Mechanismus der Typinferenz, der uns bei der Deklaration von Prozeduren viel Schreibarbeit erspart, indem er fehlende Typangaben automatisch ergänzt.

3.1 Abstraktionen und kaskadierte Prozeduren

Bisher haben wir Prozeduren mit Deklarationen beschrieben, die die beschriebene Prozedur an einen Bezeichner binden. Es ist aber oft einfacher, Prozeduren durch spezielle Ausdrücke zu beschreiben, die als **Abstraktionen** bezeichnet werden. Abstraktionen bestehen aus einem Argumentmuster und einem Ausdruck und haben die Form

fn ⟨*Argumentmuster*⟩ ⇒ ⟨*Ausdruck*⟩

Hier sind Beispiele:

```
fn (x:int) => x*x
```
fn : int → int

```
(fn (x:int) => x*x) 7
```
49 : int

```
fn (x:int, y:int) => x*y
```
*fn : int * int → int*

```
(fn (x:int, y:int) => x*y) (4,7)
```
28 : int

Abstraktionen ermöglichen die Beschreibung von Prozeduren, die Prozeduren als Ergebnis liefern:

```
fun mul (x:int) = fn (y:int) => x*y
```
val mul : int → (int → int)

```
mul 7
```
fn : int → int

```
it 3
```
21 : int

```
mul 7 5
```
35 : int

Prozeduren, die Prozeduren als Ergebnis liefern, werden als **kaskadierte Prozeduren** bezeichnet.[1] Wir haben jetzt zwei Möglichkeiten, eine mehrstellige Operation wie Multiplikation durch eine Prozedur darzustellen. Bei der **kaskadierten Darstellung** verwenden wir eine Prozedur des Typs $int \to (int \to int)$, die wie oben gezeigt deklariert werden kann. Bei der **kartesischen Darstellung** verwenden wir eine Prozedur des Typs $int * int \to int$, die wie folgt deklariert werden kann:

```
fun mul (x:int, y:int) = x*y
```
*val mul : int * int → int*

```
mul (7,5)
```
35 : int

Bei der kartesischen Darstellung werden die Argumente einer mehrstelligen Operation also mithilfe eines Tupels zu einem Argument zusammengefasst. Bei der kaskadierten Darstellung werden die Argumente dagegen jeweils für sich übergeben.

Im Zusammenhang mit kaskadierten Prozeduren sind zwei Klammersparregeln von Bedeutung:

$e_1\ e_2\ e_3$	$\rightsquigarrow$	$(e_1\ e_2)\ e_3$	Prozeduranwendung klammert links
$t_1 \to t_2 \to t_3$	$\rightsquigarrow$	$t_1 \to (t_2 \to t_3)$	Pfeil klammert rechts

Mit **kaskadierten Prozedurdeklarationen** können kaskadierte Prozeduren auch ohne Abstraktionen deklariert werden:

```
fun f (x:int) (y:int) (z:int) = x+y+z
```
val f : int → int → int → int

Diese Deklaration beschreibt dieselbe Prozedur wie

```
fun f (x:int) = fn (y:int) => fn (z:int) => x+y+z
```

Aufgabe 3.1 Deklarieren Sie eine Prozedur $mul : int \to int \to int \to int$, die das Produkt dreier Zahlen liefert. Deklarieren Sie *mul* auf 3 Arten: Mit einer kaskadierten Prozedurdeklaration, mit einer Prozedurdeklaration und zwei Abstraktionen, und mit einer Deklaration mit *val* und drei Abstraktionen.

[1] Kaskadierte Prozeduren werden im Englischen als *curried procedures* bezeichnet. Diese Bezeichnung nimmt Bezug auf Haskell B. Curry, der die kaskadierte Darstellung mehrstelliger Operationen bekannt machte. Erfunden wurde die kaskadierte Darstellung um 1920 von Moses Schönfinkel.

Aufgabe 3.2 Deklarieren Sie zwei Prozeduren, die die Operation *div* (ganzzahlige Division) kartesisch und kaskadiert darstellen.

Aufgabe 3.3 Geben Sie die Baumdarstellung des Ausdrucks

```
mul x y + mul x (y + 2) * 5
```

an. Überprüfen Sie die Richtigkeit Ihrer Darstellung mit einem Interpreter.

Aufgabe 3.4 Geben Sie die Baumdarstellungen der folgenden Typen an:

a) $int \to real \to int \to bool$

b) $int \to int * bool * int \to int$

Aufgabe 3.5 Geben Sie zu den folgenden Abstraktionen semantisch äquivalente Ausdrücke an, die ohne die Verwendung von Abstraktionen gebildet sind.

a) $fn\,(x\colon int) \Rightarrow x * x$

b) $fn\,(x\colon int) \Rightarrow fn\,(y\colon int) \Rightarrow x + y$

Hilfe: Verwenden Sie Let-Ausdrücke und Prozedurdeklarationen.

3.2 Tripeldarstellung und Rekursion

Wir betrachten jetzt Beispiele für die Tripeldarstellung kaskadierter Prozeduren. Wir beginnen mit dem Programm

```
fun f (x:int) (y:int) = x+y
val g = f 7
```

das die folgenden Bindungen liefert:

$$f := (fun\,f\;x\,y = x + y,\ \ int \to int \to\ int,\ \ [\,])$$
$$g := (fn\,y \Rightarrow x + y,\ \ int \to int,\ \ [x := 7])$$

Der Code der aus der kaskadierten Prozedur f abgeleiteten Prozedur g wird also durch eine Abstraktion dargestellt. Unser zweites Programm

```
fun f (x:int) (y:int) : int = f x y
val g = f 7
```

liefert eine endrekursive kaskadierte Prozedur f und eine daraus abgeleitete Prozedur g:

$$f := (fun\,f\;x\,y = f\;x\,y,\ \ int \to int \to\ int,\ \ [\,])$$
$$g := (fn\,y \Rightarrow f\;x\,y,\ \ int \to int,$$
$$\qquad [x := 7,\ \ f := (fun\,f\;x\,y = f\;x\,y,\ \ int \to int \to\ int,\ \ [\,])])$$

Beachten Sie, dass die abgeleitete Prozedur g nicht rekursiv ist. Sie wendet jedoch die rekursive Prozedur f an, die in ihrer Umgebung abgelegt ist.

Aufgabe 3.6 Geben Sie die Tripeldarstellung der Prozedur an, zu der der folgende Ausdruck auswertet:

```
(fn (x:int) => fn (b:bool) => if b then x else 7) (2+3)
```

Aufgabe 3.7 Geben Sie die Tripeldarstellung der Prozedur an, zu der der folgende Ausdruck auswertet:

```
let val a = 7
    fun f (x:int) = a + x
    fun g (x:int) (y:int) : int = g (f x) y
in
    g (f 5)
end
```

3.3 Höherstufige Prozeduren

Eine Prozedur heißt **höherstufig**, wenn eines ihrer Argumente eine Prozedur ist. Als erstes Beispiel betrachten wir eine Prozedur *sum*, die zu einer Prozedur f und einer Zahl $n \geq 0$ die Summe $0 + f\,1 \cdots + f\,n$ liefert:[2]

$$sum : (int \to int) \to int \to int$$
$$sum\ f\ n\ =\ 0 + f\,1 \cdots + f\,n$$

Wir deklarieren *sum* wie folgt:

```
fun sum (f:int->int) (n:int) : int =
    if n<1 then 0 else sum f (n-1) + f n
```

Die Summe der Zahlen von 1 bis 100 bekommen wir mit *sum* gemäß

```
sum (fn (i:int) => i) 100
```
5050 : int

Die Summe der Quadratzahlen 1^2 bis 10^2 erhalten wir wie folgt:

```
sum (fn (i:int) => i*i) 10
```
385 : int

[2] Frage: Warum schreiben wir statt $0 + f\,1 \cdots + f\,n$ nicht einfacher $f\,1 + \cdots + f\,n$? Antwort: Wir wollen die Rekursionsgleichungen der Prozedur *sum* genauer wiedergeben:

$sum\ f\ n = 0$ für $n < 1$
$sum\ f\ n = sum\ f\ (n-1) + f\,n$ für $n \geq 1$

Schließlich deklarieren wir mit *sum* eine Prozedur *gauss*, die zu $n \geq 0$ die Summe der Zahlen 0 bis n liefert.

```
val gauss = sum (fn (i:int) => i)
```
val gauss : int → int

Aufgabe 3.8 Deklarieren Sie eine Prozedur $prod : (int \to int) \to int \to int$, die für $n \geq 0$ die Gleichung $prod\ f\ n = 1 \cdot (f\ 1) \cdot \ldots \cdot (f\ n)$ erfüllt. Deklarieren Sie außerdem mithilfe von *prod* eine Prozedur $fac : int \to int$, die für $n \geq 0$ die Fakultät $n!$ berechnet (siehe Aufgabe 1.26 auf S. 22). Die Prozedur *fac* soll nicht rekursiv sein.

Aufgabe 3.9 Deklarieren Sie mithilfe der höherstufigen Prozedur *sum* eine Prozedur $sum' : (int \to int) \to int \to int \to int$, die für $k \geq 0$ die Gleichung $sum'\ f\ m\ k = 0 + f(m+1) \cdots + f(m+k)$ erfüllt. Die Prozedur sum' soll nicht rekursiv sein.

Aufgabe 3.10 Geben Sie die Baumdarstellungen der folgenden Typen an:

a) $(int \to int) \to int$

b) $int \to (int * bool \to int) \to int$

c) $(int \to bool) \to (bool \to real) \to int \to real$

Aufgabe 3.11 Geben Sie geschlossene Abstraktionen an, die die folgenden Typen haben:

a) $(int * int \to bool) \to int \to bool$

b) $(int \to bool \to real) \to int \to bool \to real$

c) $(int \to bool) \to (bool \to real) \to int \to real$

d) $(int \to real) \to (bool \to int) \to int * bool \to real * int$

Die Abstraktionen sollen nur mit Prozeduranwendungen, Tupeln und Bezeichnern gebildet werden. Konstanten und Operatoren sollen nicht verwendet werden.

Aufgabe 3.12 Schreiben Sie zwei Prozeduren

$cas : (int * int \to int) \to int \to int \to int$

$car : (int \to int \to int) \to int * int \to int$

sodass *cas* zur kartesischen Darstellung einer zweistelligen Operation die kaskadierte Darstellung und *car* zur kaskadierten Darstellung die kartesische Darstellung liefert. Erproben Sie *cas* und *car* mit Prozeduren, die das Maximum zweier Zahlen liefern:

```
fun maxCas (x:int) (y:int) = if x<y then y else x
fun maxCar (x:int, y:int) = if x<y then y else x
val maxCas' = cas maxCar
val maxCar' = car maxCas
```

Wenn Sie *cas* und *car* richtig geschrieben haben, verhält sich *maxCas'* genauso wie *maxCas* und *maxCar'* genauso wie *maxCar*. Hinweis: Die Aufgabe hat eine sehr einfache Lösung.

3.3.1 Bestimmte Iteration: Iter

Als zweites Beispiel für eine höherstufige Prozedur betrachten wir eine endrekursive Prozedur

$$iter : int \to int \to (int \to int) \to int$$

die zu einer Zahl $n \geq 0$ (der **Schrittzahl**), einer Zahl s (dem **Startwert**) und einer Prozedur f (der **Schrittfunktion**) die Zahl liefert, die man durch n-maliges Anwenden von f auf s erhält:

$$iter\ n\ s\ f = \underbrace{f(\ldots(f}_{n\text{-mal}}\ s)\ldots)$$

Wir deklarieren *iter* wie folgt:

```
fun iter (n:int) (s:int) (f:int->int) : int =
    if n<1 then s else iter (n-1) (f s) f
```

Die durch *iter* formulierte Berechnungstechnik wird als **bestimmte Iteration** bezeichnet.[3] Mit bestimmter Iteration können wir beispielsweise die Potenzen x^n endrekursiv berechnen, da diese ausgehend von 1 durch n-maliges Multiplizieren mit x bestimmt werden können:

$$x^n = 1 \underbrace{\cdot x \ \ldots \ \cdot x}_{n\text{-mal}}$$

```
fun power (x:int) (n:int) = iter n 1 (fn (a:int) => a*x)
val power : int → int → int

power 2 10
1024 : int
```

Die Prozedur *power* liefert dieselben Ergebnisse wie die in § 1.9 besprochene Prozedur *potenz*. Im Gegensatz zu *potenz* ist *power* jedoch nicht rekursiv. Stattdessen benutzt *power* die höherstufige Prozedur *iter*, die die für die Berechnung von Potenzen erforderliche Endrekursion zur Verfügung stellt.

Aufgabe 3.13 Deklarieren Sie eine Prozedur $mul : int \to int \to int$, die das Produkt zweier Zahlen x und $n \geq 0$ gemäß der Gleichung

$$x \cdot n = 0 \underbrace{+ x \ \ldots \ + x}_{n\text{-mal}}$$

durch Addieren berechnet. Die Prozedur *mul* soll mithilfe der Prozedur *iter* formuliert werden und nicht rekursiv sein.

[3] Bestimmte Iteration ist in vielen Programmiersprachen in der Form von For-Schleifen verfügbar.

3.3.2 Unbestimmte Iteration: First

Die Berechnungstechnik **unbestimmte Iteration** liefert zu einer Zahl s und einer Prozedur p die kleinste ganze Zahl $x \geq s$, für die $p\ true$ liefert.[4] Wir stellen dieses Berechnungskonstrukt durch die folgende endrekursive Prozedur zur Verfügung:

```
fun first (s:int) (p:int->bool) : int =
    if p s then s else first (s+1) p
```
val first : int → (int → bool) → int

Die durch *first* berechnete Funktion können wir wie folgt charakterisieren:

$$first\ s\ p = \min\{x \in \mathbb{Z} \mid x \geq s \text{ und } p\,x = true\}$$

Da für die natürliche Quadratwurzel (§ 1.10) die Gleichung

$$\lfloor\sqrt{n}\rfloor = \min\{k \in \mathbb{N} \mid k^2 > n\} - 1$$

gilt, können wir sie unmittelbar mit unbestimmter Iteration berechnen:

```
fun sqrt (x:int) = first 1 (fn (k:int) => k*k>x) - 1
```
val sqrt : int → int

```
sqrt 15
```
3 : int

Man spricht bei der durch die Prozedur *first* formulierten Berechnungstechnik von unbestimmter Iteration, da die Anzahl von Schritten, die die Prozedur *first* benötigt (anders als bei bestimmter Iteration) nicht a priori bekannt ist. Im Extremfall kann es sogar sein, dass die Prozedur *first* divergiert, obwohl der Test p für alle Argumente terminiert.

Aufgabe 3.14 Geben Sie eine Abstraktion e an, sodass die Ausführung des Ausdrucks *first x e* für alle x divergiert.

Aufgabe 3.15 Deklarieren Sie mit *first* eine Prozedur $divi : int \to int \to int$, die zu $x \geq 0$ und $m > 0$ dasselbe Ergebnis liefert wie Division mit dem Operator *div*. Hinweis: Für $x \geq 0$ und $m > 0$ liefert *div* die größte ganze Zahl k mit $k \cdot m \leq x$.

3.4 Bestimmte Iteration: Polymorphes Iter

Betrachten Sie nochmals die Beschreibung der Prozedur *iter*:

$$iter\ n\ s\ f\ = \underbrace{f(\ldots(f}_{n\text{-mal}}\ s)\ldots)$$

[4] In der Theoretischen Informatik wird unbestimmte Iteration als μ-Rekursion bezeichnet.

Offenbar können wir *iter* gemäß

$$iter\colon \underbrace{int}_{n} \to \underbrace{\alpha}_{s} \to \underbrace{(\alpha \to \alpha)}_{f} \to \alpha$$

typisieren, wobei α ein beliebiger Typ sein kann. Die bisherige Deklaration von *iter* in § 3.3.1 erfasst aber nur den Fall $\alpha = int$. Die Deklaration einer **polymorphen Prozedur** *iter*, die gemäß aller Typen $int \to \alpha \to (\alpha \to \alpha) \to \alpha$ anwendbar ist, gelingt wie folgt:

```
fun 'a iter (n:int) (s:'a) (f:'a->'a) : 'a =
    if n<1 then s else iter (n-1) (f s) f
```
val α iter : int → α → (α → α) → α

Bei der Eingabe in den Interpreter wird die **Typvariable** α durch das Wort `'a` beschrieben. Für die lexikalische Syntax sind Typvariablen eine eigene Klasse von Wörtern, die mit dem Hochkomma ' beginnen und dann mit Buchstaben fortgesetzt werden.

Mit der polymorphen Version von *iter* können wir Potenzen x^n sowohl für ganzzahliges als auch für reelles x berechnen:

```
fun power (x:int) (n:int) = iter n 1 (fn (a:int) => a*x)
```
val power : int → int → int

```
power 2 10
```
1024 : int

```
fun power' (x:real) (n:int) = iter n 1.0 (fn (a:real) => a*x)
```
val power' : real → int → real

```
power' 2.0 10
```
1024.0 : real

Beachten Sie, dass die Prozeduren *power* und *power'* ohne explizite Rekursion auskommen, da die für die Berechnung der Potenzen notwendige Rekursion über die Prozedur *iter* bereitgestellt wird.

Mit der polymorphen Prozedur *iter* können auch viele andere Funktionen ohne explizite Rekursion berechnet werden. Ein schönes Beispiel ist die Funktion, die zu n die Summe $0+1+2+3\ldots+n$ liefert. Wahrscheinlich sehen Sie nicht gleich, wie diese Funktion mit *iter* bestimmt werden kann. Der Trick besteht darin, mit dem Paar $(1,0)$ zu beginnen und die Funktion $(k,a) \mapsto (k+1, a+k)$ darauf n-mal anzuwenden:

$$(1,0) \to (2,1) \to (3,3) \to (4,6) \ldots \to (n+1, 0+1\ldots+n)$$

Diese Idee liefert die Prozedur

```
fun gauss (n:int) =
    #2(iter n (1,0) (fn (i:int, a:int) => (i+1, a+i)))
```
val gauss : int → int

```
gauss 10
```
55 : int

Aufgabe 3.16 Deklarieren Sie mit *iter* eine Prozedur *fac*, die zu $n \geq 0$ die n-te Fakultät $n!$ liefert.

Aufgabe 3.17 Deklarieren Sie mit *iter* eine Prozedur *sum*, die für $n \geq 0$ die Gleichung $sum\ n\ f = 0 + f\ 1 \ldots + f\ n$ erfüllt.

3.5 Polymorphe Typisierung

Wir wollen jetzt näher auf die programmiersprachlichen Aspekte polymorpher Prozeduren eingehen. Die für die polymorphe Prozedur *iter* möglichen Typen können wir durch das **Typschema**

$$\forall\alpha.\ int \to \alpha \to (\alpha \to \alpha) \to \alpha$$

beschreiben, das die Typvariable α durch "$\forall\alpha$." explizit **quantifiziert**. Die möglichen Typen für *iter* ergeben sich dann als **Instanzen** des Schemas:

$int \to int \to (int \to int) \to int$	$(\alpha = int)$
$int \to real \to (real \to real) \to real$	$(\alpha = real)$
$int \to int * int \to (int * int \to int * int) \to int * int$	$(\alpha = int * int)$

Hier ist eine polymorphe Prozedur, die die zweite Komponente eines Paars liefert:

```
fun ('a,'b) project2 (x:'a, y:'b) = y
```
*val (α, β) project2 : α * β → β*

```
project2 (1,~1)
```
~1 : int

```
project2 (1,true)
```
true : bool

Das Typschema dieser Prozedur quantifiziert zwei Typvariablen:

$$\forall\alpha\ \beta.\ \alpha * \beta \to \beta$$

Hier sind Instanzen dieses Schemas:

$int * int \to int$	$(\alpha = int,\ \beta = int)$
$int * real \to real$	$(\alpha = int,\ \beta = real)$
$(int * bool) * real \to real$	$(\alpha = int * bool,\ \beta = real)$

Aufgabe 3.18 Geben Sie Instanzen für die folgenden Typschemen an:

a) $\forall\alpha.\ \alpha \to \alpha$

b) $\forall\alpha\ \beta.\ \alpha \to \beta \to \alpha$

c) $\forall\alpha\ \beta\ \gamma.\ \alpha * \beta * \gamma$

d) $\forall\alpha.\ \alpha * \alpha$

Aufgabe 3.19 Deklarieren Sie polymorphe Prozeduren wie folgt:

$$\begin{aligned} id &: \forall\alpha.\ \alpha \to \alpha \\ com &: \forall\alpha\ \beta\ \gamma.\ (\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma \\ cas &: \forall\alpha\ \beta\ \gamma.\ (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma \\ car &: \forall\alpha\ \beta\ \gamma.\ (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma \end{aligned}$$

Ihre Prozeduren sollen nicht rekursiv sein. Machen Sie sich klar, dass die angegebenen Typschemen das Verhalten der Prozeduren eindeutig festlegen. Vergleichen Sie die Prozeduren *cas* und *car* mit denen aus Aufgabe 3.12 auf S. 53.

Aufgabe 3.20 Dieter Schlau ist ganz begeistert von polymorphen Prozeduren. Er deklariert die Prozedur

```
fun 'a pif (x:bool, y:'a, z:'a) = if x then y else z
val α pif : bool * α * α → α
```

und behauptet, dass man statt eines Konditionals stets die Prozedur *pif* verwenden kann:

$$\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \quad \rightsquigarrow \quad \textit{pif}(e_1, e_2, e_3)$$

Was übersieht Dieter? Denken Sie an rekursive Prozeduren und daran, dass der Ausdruck, der das Argument einer Prozeduranwendung beschreibt, vor dem Aufruf der Prozedur ausgeführt wird.

3.5.1 Monomorphe und polymorphe Bezeichner

Im Rahmen eines Programms heißt ein Bezeichner **polymorph**, wenn er mit einem Typschema getypt ist, und **monomorph**, wenn er normal mit einem Typ getypt ist. Eine Deklaration heißt **polymorph**, wenn sie mindestens einen polymorphen Bezeichner deklariert, und **monomorph**, wenn sie nur monomorphe Bezeichner deklariert. Eine Prozedur heißt **monomorph**, wenn sie nur gemäß eines Typs angewendet werden darf, und **polymorph**, wenn sie gemäß aller Instanzen eines Typschemas angewendet werden darf.

Argumentvariablen von Prozeduren werden in Standard ML immer monomorph getypt. Polymorphe Bezeichner können also nur mithilfe von Deklarationen eingeführt werden. Normalerweise erfolgt die Deklaration polymorpher Bezeichner mithilfe von Prozedurdeklarationen, aber auch Deklaration mit *val* können polymorphe Bezeichner einführen.

Die sogenannte **Identitätsprozedur** kann wie folgt deklariert werden:

```
fun 'a id (x:'a) = x
```

Diese Deklaration typt den Bezeichner *id* mit dem Schema $\forall\alpha.\ \alpha \to \alpha$ und die Argumentvariable x mit dem Typ α. Damit ist *id* polymorph und x monomorph getypt. Der Bezeichner *id* kann gemäß jeder Instanz seines Typschemas benutzt werden. Beispielsweise wird er in der Deklaration

```
val x = id 5
```

gemäß der Instanz $int \to int$ benutzt, und in der Deklaration

```
val x = id 5.0
```

gemäß der Instanz $real \to real$. Schließlich benutzt die Deklaration

```
val x = id(id 3, id 5.0)
```

den Bezeichner *id* gleichzeitig gemäß der Instanzen $int * real \to int * real$, $int \to int$ und $real \to real$.

Aufgabe 3.21 Geben Sie ein Typschema für den Bezeichner f an, sodass der Ausdruck $(f\ 1,\ f\ 1.0)$ wohlgetypt ist.

Aufgabe 3.22 Warum gibt es keinen Typen t, sodass die Abstraktion $fn\, f : t \Rightarrow (f\ 1,\ f\ 1.0)$ wohlgetypt ist?

Aufgabe 3.23 (Let und Abstraktionen) Dieter Schlau ist begeistert. Scheinbar kann man Let-Ausdrücke mit Val-Deklarationen auf Abstraktion und Applikation zurückführen:

$$let\ val\ x = e\ in\ e'\ end \quad \rightsquigarrow \quad (fn\ x\colon\ t \Rightarrow e')\, e$$

Auf der Heimfahrt von der Uni kommen ihm jedoch Zweifel an seiner Entdeckung, da ihm plötzlich einfällt, dass Argumentvariablen nicht polymorph getypt werden können. Können Sie ein Beispiel angeben, das zeigt, dass Dieters Zweifel berechtigt sind? Hilfe: Geben Sie einen semantisch zulässigen Let-Ausdruck an, dessen Übersetzung gemäß des obigen Schemas scheitert, da Sie keinen passenden Typ t für die Argumentvariable x finden können.

3.5.2 Ambige Deklarationen

Betrachten Sie die Deklaration

```
val f = id id
```

Für das rechte Auftreten des Bezeichners *id* können wir jede Instanz $\alpha \to \alpha$ des Typschemas für *id* wählen, wenn wir dazu passend $(\alpha \to \alpha) \to (\alpha \to \alpha)$ für das linke Auftreten

von *id* benutzen. Das legt den Schluss nahe, dass der Bezeichner *f* polymorph mit dem Schema $\forall\alpha.\ \alpha \to \alpha$ getypt werden kann.

Standard ML typt die obige Deklaration jedoch monomorph. Da aus der Deklaration selbst aber kein eindeutiger Typ für *f* bestimmt werden kann, stellt der Interpreter den Typ von *f* mithilfe einer **freien Typvariable** dar:

```
val f = id id
val f : α → α
!Warning: Free type variable α  (value polymorphism)
```

Wenn eine nachfolgende Deklaration Information über den Typ von *f* liefert, wird die freie Typvariable entsprechend **instanziiert**:

```
val z = f 5
val z = 5 : int
! Warning: free type variable α instantiated to int
```

Wenn man beide Deklarationen zusammen eingibt, kann der Typ von *f* sofort eindeutig bestimmt werden:

```
val f = id id
val z = f 5
val f : int → int
val z = 5 : int
```

Beachten Sie, dass das Programm

```
val f = id id
val z = f 5
val u = f 5.0
! Type clash: expression of type real cannot have type int
```

nicht wohlgetypt ist, da bei der Analyse der Deklaration von *u* der Typ von *f* bereits auf $int \to int$ festgelegt ist.

Es gibt einige Spezialfälle, bei denen Standard ML mit *val* deklarierte Bezeichner polymorph typt. Das ist insbesondere immer dann der Fall, wenn die Deklaration keine Anwendung einer Prozedur oder eines Operators enthält. Hier ist ein Beispiel:

```
val g = id
val α g : α → α
```

Bei dieser Deklaration wird *g* so wie *id* mit dem Schema $\forall\alpha.\ \alpha \to \alpha$ getypt.

Sie wollen jetzt natürlich wissen, warum Standard ML bestimmte Val-Deklarationen monomorph typisiert, obwohl nichts gegen eine polymorphe Typisierung zu sprechen scheint. Der Grund dafür sind die sogenannten Speicheroperationen, die wir aber erst in § 15.1 einführen werden.

Ein Paar Worte zu Standard ML

ML ist eine theorienahe Sprache, die in der programmiersprachlichen Forschung und in der Informatikausbildung eine wichtige Rolle spielt. Entwickelt wurde ML ab 1974 von Robin Milner und anderen für die Programmierung des interaktiven Beweissystems LCF (logic of computable functions). ML war die erste Programmiersprache mit polymorpher Typisierung und typsicheren Ausnahmen. Das Akronym ML steht für meta language, ein Begriff, der sich aus dem Zusammenhang mit LCF ergibt. Robin Milner wurde 1991 mit dem ACM Turing-Preis ausgezeichnet, der höchsten Auszeichnung in der Informatik, vergleichbar dem Nobelpreis oder der Fields-Medaille in der Mathematik.

Es gibt viele Dialekte von ML. Standard ML ist das Ergebnis einer von Robin Milner geleiteten Arbeitsgruppe zur mathematisch präzisen Definition einer Standardversion von ML. Die erste Version der Sprachbeschreibung wurde 1990 vorgelegt, 1997 wurden einige Punkte revidiert. Die Sprachdefinition von Standard ML [3] gilt als Musterbeispiel für die präzise Beschreibung von Programmiersprachen.

ML gehört zur Familie der funktionalen Programmiersprachen. Für diese Sprachen ist kennzeichnend, dass sie Prozeduren durch Gleichungen beschreiben und dass Prozeduren mathematische Funktionen berechnen. Ein wichtiger Grund, warum wir in diesem Buch Standard ML benutzen, besteht darin, dass Standard ML neben seinem funktionalen Kern auch über Speicheroperationen verfügt, wie sie in imperativen und objektorientierten Programmiersprachen (z.B. C und Java) eine prominente Rolle spielen.

Wir merken uns, dass Prozedurdeklarationen immer monomorph oder polymorph getypt werden. Dagegen gibt es für Val-Deklarationen drei Möglichkeiten: monomorph, ambig oder polymorph. Bei **ambigen Deklarationen** handelt es sich um Deklarationen, die in der bisher eingeführten Teilsprache eigentlich polymorph getypt werden könnten, aber von Standard ML mithilfe einer freien Typvariablen monomorph behandelt werden. Eine eigentlich polymorphe Deklaration wird in Standard ML immer dann als ambig behandelt, wenn ihre Ausführung die Ausführung einer Prozedur- oder Operatoranwendung beinhaltet.

3.6 Typinferenz

Standard ML ist gemäß eines Abkürzungsprinzips entworfen, das es erlaubt, die Typangaben für die Argumentvariablen und die Ergebnisse von Prozeduren wegzulassen. Fehlende Typangaben werden automatisch und in eindeutiger Weise mit einem Verfahren ergänzt, das als **Typinferenz** bezeichnet wird. Hier sind Beispiele:

```
fun con x y = if x then y else false
```
val con : bool → bool → bool

```
fun sum f n = if n<1 then 0 else sum f (n-1) + f n
```
val sum : (int → int) → int → int

```
val sqsum = sum (fn i => i*i)
```
val sqsum : int → int

```
fun id x = x
```
val α id : α → α

```
fun iter n s f = if n<1 then s else iter (n-1) (f s) f
```
val α iter : int → α → (α → α) → α

```
fun power x n = iter n 1 (fn a => a*x)
```
val power : int → int → int

Die Deklarationen von *id* und *iter* zeigen, dass Typinferenz fehlende Typangaben möglichst allgemein ergänzt.

Da die Operatoren für Zahlen überladen sind, kann es sein, dass bestimmte Bezeichner sowohl mit *int* als auch mit *real* getypt werden können. Wenn es eine Wahl gibt, entscheidet sich Typinferenz immer für *int*, um ein eindeutiges Ergebnis liefern zu können:

```
fun plus x y = x+y
```
val plus : int → int → int

Wenn Sie stattdessen den Typ *real → real → real* für *plus* haben wollen, müssen Sie die Deklaration mit einer Typangabe versehen, die den mit *int* gebildeten Typ ausschließt:

```
fun plus (x:real) y = x+y
```
val plus : real → real → real

Diese Deklaration gibt den Typ für die Argumentvariable *x* vor. Wenn Sie wollen, können Sie auch den Ergebnistyp der Prozedur angeben:

```
fun plus x y : real = x+y
```
val plus : real → real → real

Sie können auch die Typen für beide Argumentvariablen und den Ergebnistyp angeben:

```
fun plus (x:real) (y:real) : real = x+y
```
val plus : real → real → real

Es ist möglich, beliebige Teilausdrücke mit einer Typangabe zu versehen:

```
fn (x,y) => (x+y : real) * x
```
*fn : real * real → real*

```
(fn x => x) : bool -> bool
```
fn : bool → bool

Auch die mit einer Val-Deklaration eingeführten Bezeichner können mit Typangaben versehen werden:

```
val f : real -> real = fn x => x
```
val f : real → real

Aufgabe 3.24 Deklarieren Sie eine Prozedur *first* : $int \to (int \to bool) \to int$, die zu x und p die kleinste Zahl $y \geq x$ mit $p\,y = true$ liefert. Verzichten Sie dabei vollständig auf Typangaben.

Aufgabe 3.25 Geben Sie die Typschemen an, mit denen die Bezeichner p und q des folgenden Programms typisiert werden.

```
fun p f (x,y) = f x y
fun q f g x = g (f x)
```

Aufgabe 3.26 Geben Sie Deklarationen an, die monomorph getypte Bezeichner wie folgt deklarieren:

a: $int * unit * bool$

b: $unit * (int * unit) * (real * unit)$

c: $int \to int$

d: $int * bool \to int$

e: $int \to real$

f: $int \to real \to real$

g: $(int \to int) \to bool$

Verzichten Sie dabei auf explizite Typangaben und verwenden Sie keine Operator- und Prozeduranwendungen.
Hinweis: Für einige der Deklarationen ist die Verwendung eines Konditionals essentiell. Die Typregel für Konditionale verlangt, dass die Konsequenz und die Alternative den gleichen Typ haben (siehe § 2.6). Außerdem ist für einige der Deklarationen die Verwendung von Tupeln und Projektionen erforderlich, um Werte vergessen zu können, die nur zur Steuerung der Typinferenz konstruiert wurden.

Aufgabe 3.27 (Projektionen) Im Zusammenhang mit fehlenden Typangaben kann die Verwendung von Projektionen problematisch sein. Beispielsweise kann Typinferenz das Programm *fun f x = #1x* nicht typisieren. Können Sie erklären, warum das so ist?

3.7 Typen und Gleichheit

In Standard ML ist der Gleichheitstest (=): $t * t \to bool$ nicht für alle Typen t verfügbar. Typen, für die der Test verfügbar ist, heißen **Typen mit Gleichheit**. Dazu gehören *int*, *bool* und *unit*. Für Prozedurtypen $t_1 \to t_2$ ist der Gleichheitstest generell nicht verfügbar.

Für einen Produkttypen $t_1 * \cdots * t_n$ ist der Gleichheitstest genau dann verfügbar, wenn er für alle Komponententypen $t_1, \ldots, t_n$ verfügbar ist.

Um bei Typschemen zwischen Typen mit und ohne Gleichheit unterscheiden zu können, gibt es zwei Arten von Typvariablen. Die bisher verwendeten Typvariablen mit nur einem Hochkomma (z.B. *'a*) können zu beliebigen Typen instanziiert werden. Dagegen können Typvariablen mit zwei Hochkommas (z.B. *''a*) nur zu Typen mit Gleichheit instanziiert werden. Hier sind Beispiele:

```
fun eq x y = x = y
```
val ''a eq : ''a → ''a → bool

```
fun neq x y = x <> y
```
val ''a eq : ''a → ''a → bool

```
fun f (x,y,z) = if x=y then x else z
```
*val ''a f : ''a * ''a * ''a → ''a*

```
(fn x => 2*x) = (fn x => 2*x)
```
! Type clash: int → int is not an equality type

Aufgabe 3.28 Deklarieren Sie eine Identitätsprozedur *''a ideq* : *''a* → *''a*, deren Typschema auf Typen mit Gleichheit eingeschränkt ist. Verzichten Sie dabei auf explizite Typangaben.

3.8 Bezeichnerbindung

Bezeichner dienen im Rahmen eines Programms als Namen für Werte. Bezeichner werden mit Deklarationen und Abstraktionen eingeführt. Im Rahmen der semantischen Analyse werden Bezeichner an Typen oder Typschemen gebunden. Während der Ausführung eines Programms werden Bezeichner an Werte gebunden. Wir wollen jetzt genauer verstehen, wie diese Bindungsmechanismen funktionieren.

3.8.1 Lexikalische Bindungen

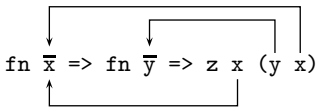
Im Rahmen einer Phrase unterscheiden wir zunächst zwischen **definierenden** und **benutzenden Bezeichnerauftreten**. Bei der Ausführung führen definierende Auftreten neue Bindungen ein, während benutzende Auftreten gemäß bestehender Bindungen Werte liefern. Als Beispiel betrachten wir den Ausdruck

```
fn x => fn y => z x (y x)
```

der zwei definierende und vier benutzende Bezeichnerauftreten enthält. Wenn wir die definierenden Auftreten durch Überstreichen markieren, bekommen wir das folgende Bild:

fn $\overline{x}$ => fn $\overline{y}$ => z x (y x)

Neben definierenden und benutzenden Bezeichnerauftreten unterscheiden wir zwischen **gebundenen** und **freien Bezeichnerauftreten**. Definierende Bezeichnerauftreten sind immer auch gebundene Bezeichnerauftreten. Dagegen können benutzende Bezeichnerauftreten gebunden oder frei sein. Ein benutzendes Bezeichnerauftreten ist genau dann gebunden, wenn ihm ein definierendes Auftreten zugeordnet werden kann, das bei der Ausführung für die Einführung des zu benutzenden Wertes zuständig ist. Wir sagen dann, dass das benutzende Auftreten **lexikalisch** an das definierende Auftreten **gebunden** ist und bezeichnen diese Zuordnung als eine **lexikalische Bindung**. Wenn wir die lexikalischen Bindungen unseres Beispielausdrucks durch Pfeile darstellen, bekommen wir das folgende Bild:



Jeder Pfeil führt von einem gebundenen benutzenden Bezeichnerauftreten zu dem definierenden Bezeichnerauftreten, an das es lexikalisch gebunden ist. Das Auftreten von z ist das einzige freie Bezeichnerauftreten.

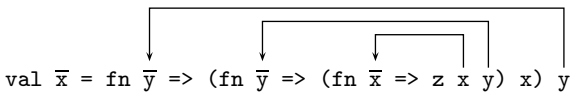
Im Rahmen einer Phrase ist ein Bezeichnerauftreten genau dann frei, wenn es benutzend ist und lexikalisch an kein definierendes Bezeichnerauftreten gebunden ist. Eine Phrase ist **offen**, wenn sie freie Bezeichnerauftreten enthält, und **geschlossen**, wenn sie keine freien Bezeichnerauftreten enthält (vergleiche § 2.4).

3.8.2 Konsistente Umbenennung und Bereinigung

Eine Phrase heißt **bereinigt**, wenn sie zwei Bedingungen erfüllt:

1. Kein Bezeichner hat mehr als ein definierendes Auftreten.
2. Wenn ein Bezeichner ein definierendes Auftreten hat, dann hat er kein freies Auftreten.

Die lexikalischen Bindungen einer bereinigten Phrase sind besonders leicht zu erkennen. Der obige Beispielausdruck ist bereinigt. Hier ist ein Beispiel für die lexikalischen Bindungen einer unbereinigten Phrase:



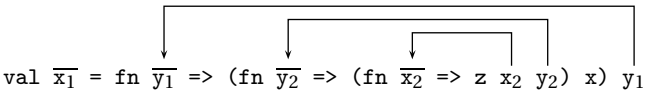
In dieser Phrase treten die Bezeichner x und z frei auf. Außerdem gibt es 2 definierende Auftreten des Bezeichners x.

```
let
   val x = 2*x
   val x = 2*x
   fun f x = if x<2 then x
             else f(x-1)
   val f = fn x => f x
in
   f x - y
end
```

```
let
   val x̄1 = 2*x
   val x̄2 = 2*x1
   fun f̄1 x̄3 = if x3<2 then x3
               else f1(x3-1)
   val f̄2 = fn x̄4 => f1 x4
in
   f2 x2 - y
end
```

Abbildung 3.1: *Bereinigung eines Let-Ausdrucks*

Wenn die gebundenen Bezeichner einer Phrase so umbenannt werden, dass sich die lexikalischen Bindungen nicht ändern, sprechen wir von einer **konsistenten Umbenennung** der Phrase. Die gerade betrachtete Phrase können wir durch konsistente Umbenennung in eine semantisch äquivalente bereinigte Phrase überführen:



Konsistente Umbenennung einer Phrase liefert semantisch äquivalente Phrasen, da die konkrete Wahl der gebundenen Bezeichner semantisch irrelevant ist. Jede Phrase lässt sich durch konsistente Umbenennung in eine semantisch äquivalente bereinigte Phrase überführen.

Abbildung 3.1 zeigt die **Bereinigung** eines Let-Ausdrucks. Darunter verstehen wir die Kennzeichnung der definierenden Bezeichnerauftreten durch Überstreichen und die konsistente Umbenennung aller gebundenen Bezeichnerauftreten durch Indizieren, sodass eine semantisch äquivalente bereinigte Phrase entsteht.

Aufgabe 3.29 Betrachten Sie den Ausdruck

```
(fn x => (fn y => (fn x => y) x) y) x
```

a) Geben Sie die Baumdarstellung des Ausdrucks an.

b) Markieren Sie die definierenden Bezeichnerauftreten durch Überstreichen.

c) Stellen Sie die lexikalischen Bindungen durch Pfeile dar.

d) Geben Sie alle Bezeichner an, die in dem Ausdruck frei auftreten.

e) Bereinigen Sie den Ausdruck durch Indizieren der gebundenen Bezeichnerauftreten.

Aufgabe 3.30 Betrachten Sie das Programm

```
val (x, y) = (2, 3)
fun f x = fn x => #1(x,y)
val y = x*y
fun g g = f x g
```

Klären Sie zuerst mithilfe eines Interpreters, wie die Deklaration der Prozedur *g* zu verstehen ist. Bereinigen Sie dann das Programm durch Überstreichen der definierenden und Indizieren der gebundenen Bezeichnerauftreten.

3.8.3 Statische und dynamische Bindungen

Neben lexikalischen Bindungen gibt es **statische und dynamische Bindungen**. Statische Bindungen werden bei der semantischen Analyse einer Phrase gebildet und binden Bezeichner an Typen (**monomorphe Bindung**) oder Typschemen (**polymorphe Bindung**). Dynamische Bindungen werden bei der Ausführung einer Phrase gebildet und binden Bezeichner an Werte. Die lexikalischen Bindungen einer Phrase beschreiben den strukturellen Rahmen für die statischen und dynamischen Bindungen. Die lexikalischen Bindungen werden bei der semantischen Analyse einer Phrase ermittelt.

Als Beispiel betrachten wir das bereinigte und geschlossene Programm

```
val x = 4*5
fun f y = if y then x else ~x
fun id z = z
```

Bei der statischen Analyse des Programms wird für jedes definierende Bezeichnerauftreten ein Typ oder ein Typschema bestimmt (statische Bindungen):

$$x : int$$
$$f : bool \to int$$
$$y : bool$$

$$id : \forall \alpha.\ \alpha \to \alpha$$
$$z : \alpha$$

Bei der Ausführung werden die durch das Programm deklarierten Bezeichner an Werte gebunden (dynamische Bindungen):

$$x := 20$$
$$f := (fun\ f\ y = if\ y\ then\ x\ else\ \sim x,\ \ bool \to int,\ \ [x := 20])$$
$$id := (fun\ id\ z = z,\ \ \forall \alpha.\ \alpha \to \alpha,\ \ [])$$

Die Ausgabe eines Interpreters liefert Information über die statischen und dynamischen Bindungen der deklarierten Bezeichner:

$$val\ x = 20 : int$$
$$val\ f : bool \to int$$
$$val\ \alpha\ id : \alpha \to \alpha$$

Aufgabe 3.31 Betrachten Sie die bereinigte Deklaration

```
fun f (x, y) = (fn z => (fn u => (fn v => u) z) y) x
```

a) Geben Sie die Baumdarstellung der Deklaration an.
b) Stellen Sie die lexikalischen Bindungen der Deklaration durch Pfeile dar.
c) Geben Sie die statischen Bezeichnerbindungen an, die durch die semantische Analyse bestimmt werden.
d) Geben Sie eine möglichst einfache, semantisch äquivalente Deklaration an, die ohne Abstraktionen gebildet ist.

3.9 Spezifikation polymorpher Prozeduren

Im Folgenden werden wir bei der Spezifikation polymorpher Prozeduren der Einfachheit halber auf die explizite Quantifizierung der Typvariablen verzichten. Beispielsweise werden wir statt

$$car : \forall\, \alpha\ \beta\ \gamma.\ (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$

einfach die folgende Zeile schreiben:

$$car : (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$

Auch bei den Interpreterausgaben werden wir auf die explizite Quantifizierung der Typvariablen verzichten, also statt

$$val\ (\alpha, \beta, \gamma)\ car : (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$

einfach nur

$$val\ car : (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$

schreiben. Das Weglassen der Quantifizierung wird zu keinen Schwierigkeiten führen, da bei den benötigten Spezifikationen immer alle vorkommenden Typvariablen quantifiziert sind.

3.10 Abgeleitete Formen: Andalso, Orelse, Op

Im Gegensatz zu natürlichen Sprachen wie Deutsch oder Englisch sind Programmiersprachen das Ergebnis sorgfältig durchdachter Entwurfsprozesse. Gut entworfene Programmiersprachen basieren auf relativ wenigen Prinzipien und Regeln, die dafür möglichst allgemein durchgezogen sind. Ein wichtiges Strukturierungsmittel ist die Aufteilung in eine **Kernsprache** und in daraus **abgeleitete Formen**. Abstraktionen und polymorphe Typisierung gehören zur Kernsprache. Kaskadierte Deklarationen und Typinferenz zählen dagegen zu den abgeleiteten Formen.

In diesem Abschnitt führen wir einige nützliche abgeleitete Formen ein. Wir beginnen mit zwei abkürzenden Formen für konditionale Ausdrücke:

$$\begin{array}{lcl} e_1 \; \mathit{andalso} \; e_2 & \rightsquigarrow & \mathit{if} \; e_1 \; \mathit{then} \; e_2 \; \mathit{else} \; \mathit{false} \\ e_1 \; \mathit{orelse} \; e_2 & \rightsquigarrow & \mathit{if} \; e_1 \; \mathit{then} \; \mathit{true} \; \mathit{else} \; e_2 \end{array}$$

Ein Beispiel für die Verwendung von *andalso* und *orelse* ist die Prozedur

```
fun forall m n p = m>n orelse (p m andalso forall (m+1) n p)
val forall : int → int → (int → bool) → bool
```

die testet, ob eine Prozedur p für alle Zahlen zwischen m und n den Wert *true* liefert.

Mit dem durch das Unterstrichzeichen "_" dargestellten **Wildcard-Muster** können Argumentvariablen anonymisiert werden, die im Rumpf einer Prozedur nicht benötigt werden:

```
fun snd (_, y) = y
val snd : α * β → β
```

Op-Ausdrücke sind Abkürzungen für Abstraktionen, die durch Operatoren beschriebene Operationen als Prozeduren zur Verfügung stellen:

$$\begin{array}{lcl} \mathit{op}+ & \rightsquigarrow & \mathit{fn}\,(x,y) \Rightarrow x+y \\ \mathit{op}< & \rightsquigarrow & \mathit{fn}\,(x,y) \Rightarrow x<y \\ \mathit{op}= & \rightsquigarrow & \mathit{fn}\,(x,y) \Rightarrow x=y \\ \mathit{op}\;\mathit{div} & \rightsquigarrow & \mathit{fn}\,(x,y) \Rightarrow x\;\mathit{div}\;y \\ \mathit{op}\sim & \rightsquigarrow & \mathit{fn}\,x \Rightarrow \sim x \end{array}$$

Da die meisten Operatoren überladen sind, ist der Verzicht auf die Typangaben bei den durch die Op-Ausdrücke beschriebenen Abstraktionen wesentlich:

```
op+(8,9)
17 : int

op+(8.0, 9.0)
17.0 : real

op+
fn : int * int → int
```

Aufgabe 3.32 (Exists) Deklarieren Sie eine Prozedur

$$\mathit{exists} : \mathit{int} \to \mathit{int} \to (\mathit{int} \to \mathit{bool}) \to \mathit{bool}$$

die testet, ob eine Prozedur für mindestens eine Zahl zwischen zwei gegebenen Zahlen den Wert *true* liefert. Orientieren Sie sich an der Prozedur *forall*.

Aufgabe 3.33 (Andalso als Prozedur) Warum können Sie einen Ausdruck e_1 *andalso* e_2 nicht immer durch eine Anwendung der Prozedur

```
fun andalso' x y = if x then y else false
```

ersetzen? Hinweis: Lösen Sie zuerst Aufgabe 3.20 auf S. 58.

3.11 Beispiel: Primzahlberechnung

Primzahlen spielen in der Informatik bei der Verschlüsselung von Daten eine wichtige Rolle. Hier wollen wir einige Prozeduren vorstellen, die Primzahlen auf elegante Weise mithilfe höherstufiger Prozeduren berechnen.

Wir beginnen mit der Definition von Primzahlen. Eine ganze Zahl $k \neq 0$ **teilt** eine ganze Zahl n, wenn es eine ganze Zahl n' mit $n = k \cdot n'$ gibt. Eine ganze Zahl $n \geq 2$ heißt **Primzahl**, wenn sie von keiner positiven ganzen Zahl außer 1 und n geteilt wird. Die ersten 25 Primzahlen sind 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97. Primzahlen haben unter anderem die folgenden Eigenschaften:

1. Es gibt unendlich viele Primzahlen.
2. Eine ganze Zahl $n \geq 2$ ist genau dann eine Primzahl, wenn sie durch keine Primzahl p mit $p^2 \leq n$ teilbar ist.
3. Jede ganze Zahl $n \geq 2$ lässt sich als Produkt von Primzahlen darstellen. Wenn man die Faktoren der Größe nach anordnet, ist diese Darstellung, die als **Primzerlegung** bezeichnet wird, eindeutig. Beispielsweise hat 1989 die Primzerlegung $1989 = 3 \cdot 3 \cdot 13 \cdot 17$.

Im Folgenden verwenden wir die höherstufigen Prozeduren *first* und *iter* aus § 3.3 sowie die höherstufige Prozedur *forall* aus § 3.10.

Wir beginnen mit einer Prozedur, die testet, ob eine Zahl eine Primzahl ist:

```
fun prime x = x>=2 andalso
              forall 2 (sqrt x) (fn k => x mod k <> 0)
val prime : int → bool
```

Die Prozedur *sqrt* liefert die natürliche Quadratwurzel und wurde in § 3.3 mithilfe der höherstufigen Prozedur *first* deklariert.

Als Nächstes deklarieren wir eine Prozedur *nextprime*, die für $x \geq 1$ die erste Primzahl $p > x$ liefert:

```
fun nextprime x = first (x+1) prime
val nextprime : int → int
```

Schließlich deklarieren wir eine Prozedur *nthprime*, die für $n \geq 1$ die n-te Primzahl liefert:

```
fun nthprime n = iter n 1 nextprime
```
val nthprime : int → int

```
nthprime 100
```
541 : int

Aufgabe 3.34 Deklarieren Sie semantisch äquivalente Prozeduren zu *sqrt*, *prime*, *nextprime* und *nthprime*, die ohne Rückgriff auf höherstufige Prozeduren formuliert sind. Realisieren Sie die erforderlichen Akkus mit Hilfsprozeduren.

Aufgabe 3.35 Schreiben Sie eine Prozedur *reduce* : $int \to int \to int * int$, die zu zwei Zahlen $n, p \geq 2$ das eindeutig bestimmte Paar (m,k) liefert, sodass $n = m \cdot p^k$ gilt und m nicht durch p teilbar ist.

3.12 Komposition von Prozeduren

Hier ist eine höherstufige Prozedur *compose*, die zu zwei Prozeduren f und g eine Prozedur liefert, die zuerst g und dann f anwendet:

```
fun compose f g x = f (g x)
```
val compose : (α → β) → (γ → α) → γ → β

Man sagt, dass *compose f g* die **Komposition von** f **und** g liefert.

```
fun plus x y = x+y
```
val plus : int → int → int

```
fun times x y = x*y
```
val times : int → int → int

```
val foo = compose (plus 2) (times 3)
```
val foo : int → int

```
foo 4
```
14 : int

Der Buchstabe "`o`" bezeichnet in Standard ML einen polymorphen Operator, der die Komposition zweier Prozeduren liefert:

```
val foo = (plus 2) o (times 3)
```
val foo : int → int

```
foo 7
```
23 : int

```
op o
```
*fn : (α → β) * (γ → α) → γ → β*

Die Tatsache, dass der Buchstabe "`o`" einen Operator bezeichnet, erweist sich gelegentlich als Stolperstein, da "`o`" infolgedessen nicht als Bezeichner verwendet werden kann.

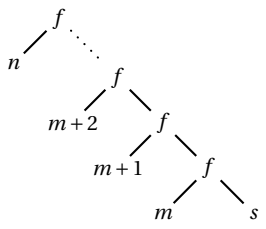
3.13 Bestimmte Iteration: Iterup und Iterdn

Abschließend stellen wir noch zwei Varianten von *iter*:

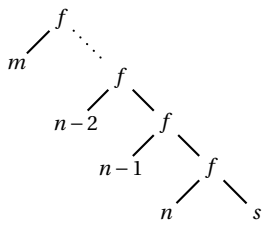
```
fun iterup m n s f = if m>n then s else iterup (m+1) n (f(m,s)) f
```
*val iterup : int → int → α → (int * α → α) → α*

```
fun iterdn n m s f = if n<m then s else iterdn (n-1) m (f(n,s)) f
```
*val iterdn : int → int → α → (int * α → α) → α*

Die Funktionsweise von *iterup* und *iterdn* lässt sich am besten grafisch verdeutlichen:[5]



iterup m n s f



iterdn n m s f

Das erste Argument der Schrittfunktionen von *iterup* und *iterdn* wird als **Index** und das zweite als **Akku** bezeichnet.

Aufgabe 3.36 Deklarieren Sie mit *iterup* eine Prozedur

a) *power*, die zu x und n die Potenz x^n liefert.

b) *fac*, die zu $n \geq 0$ die n-te Fakultät $n!$ liefert.

c) *sum*, die zu f und n die Summe $0 + f\,1 \ldots + f\,n$ liefert.

d) *iter′*, die zu n, s und f dasselbe Ergebnis liefert wie *iter n s f*.

Aufgabe 3.37 Deklarieren Sie mithilfe der Prozedur *iter* eine Prozedur

a) *iterup′*, die zu m, n, s und f dasselbe Ergebnis wie *iterup m n s f* liefert.

b) *iterdn′*, die zu n, m, s und f dasselbe Ergebnis wie *iterdn n m s f* liefert.

[5] Die hier verwendete Baumdarstellung unterscheidet sich von der in Kapitel 2 dadurch, dass Prozeduranwendungen wie Operatoranwendungen dargestellt werden.

Bemerkungen

Die Berechnungstechniken bestimmte und unbestimmte Iteration werden auch als **Rekursionsschemen** bezeichnet, da viele Berechnungsaufgaben mit ihrer Hilfe ohne explizite Rekursion formuliert werden können. Beide Iterationsformen sind endrekursiv.

Verzeichnis

Rekursionsschemen: Bestimmte und unbestimmte Iteration.

Bestimmte Iteration: *iter*, *iterdn*, *iterup*; Schrittzahl, Startwert, Schrittfunktion; Index und Akku.

Unbestimmte Iteration: *first*.

Prozeduren: kaskadierte, höherstufige, monomorphe, polymorphe; Abstraktionen; kaskadierte und kartesische Darstellung mehrstelliger Operationen.

Polymorphe Typisierung: Typschemen, freie und quantifizierte Typvariablen, Instanzierung und Instanzen; polymorphe, monomorphe und ambige Deklarationen; Spezifikation polymorpher Prozeduren.

Typen mit Gleichheit: Typvariablen mit einem und mit zwei Hochkommas.

Bezeichnerbindung: definierende und benutzende Auftreten; gebundene und freie Auftreten; freie Bezeichner einer Phrase; offene, geschlossene und bereinigte Phrasen; konsistente Umbenennung, Bereinigung; lexikalische, statische und dynamische Bindungen; monomorphe und polymorphe Bindungen.

Kernsprache und abgeleitete Formen: Typinferenz, kaskadierte Deklarationen, andalso, orelse, Wildcard-Muster, Op-Ausdrücke.

Komposition von Prozeduren.

4 Listen und Strings

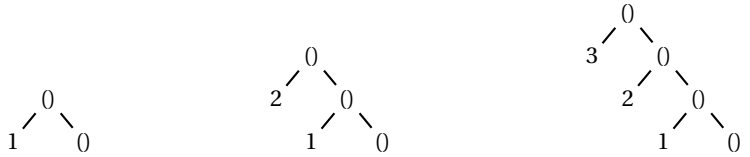
In diesem Kapitel behandeln wir zwei Datenstrukturen, deren Werte als Listen und Strings bezeichnet werden. Listen und Strings ist gemeinsam, dass es sich um zusammengesetzte Werte handelt, die Folgen von Werten darstellen.

4.1 Die Datenstruktur der Listen

Sei ein Typ t gegeben. Die **Listen über** t werden gemäß einer **rekursiven Konstruktionsvorschrift** gebildet:

1. Das leere Tupel $()$ ist eine Liste über t.
2. Wenn x ein Wert des Typs t ist und xs eine Liste über t, dann ist das Paar (x, xs) eine Liste über t.

Listen werden also schrittweise durch Paarbildung konstruiert. Hier sind Beispiele für Listen über *int*:



Aus den Beispielen ist ersichtlich, dass Listen Folgen darstellen. Dieser Zusammenhang begründet eine zusätzliche Schreibweise für Listen:

$$\begin{aligned} [] &= () \\ [x] &= (x, ()) \\ [x_1, x_2, x_3] &= (x_1, (x_2, (x_3, ()))) \\ [x_1, \ldots, x_n] &= (x_1, \ldots (x_n, ()) \ldots) \end{aligned}$$

Gegeben eine Liste $[x_1, \ldots, x_n]$, bezeichnet man die Werte $x_1, \ldots, x_n$ als die **Elemente der Liste**. Eine Liste heißt **nichtleer**, wenn sie wenigstens ein Element hat. Die Liste $[]$ wird als **leere Liste** bezeichnet. Bei einer nichtleeren Liste (x, xs) bezeichnet man x als den **Kopf** und xs als den **Rumpf** der Liste. Beispielsweise hat die Liste $[1,2,3]$ den Kopf 1 und den Rumpf $[2,3]$.

Eine Liste $[x_1,\ldots,x_n]$ hat die **Länge** n. Die leere Liste hat die Länge 0. Bei $[1,1,1]$ handelt es sich um eine Liste der Länge 3, die nur ein Element hat (die Zahl 1). Die Länge einer Liste xs bezeichnen wir mit $|xs|$.

Die **Konkatenation** zweier Listen ist wie folgt definiert:

$$[x_1,\ldots,x_m]@[y_1,\ldots,y_n] := [x_1,\ldots,x_m,y_1,\ldots,y_n]$$

Hier sind Beispiele:

$$\begin{aligned} |[]| &= 0 \\ |[7,3,3,2]| &= 4 \\ [2,5,1]@[4,5] &= [2,5,1,4,5] \\ [3]@[] &= [3] \\ |xs@ys| &= |xs|+|ys| \end{aligned}$$

Manchmal ist es hilfreich, sich eine Liste bildhaft als einen Stapel von Spielkarten vorzustellen. Der Kopf der Liste entspricht dabei der obersten Karte des Stapels. Der Rumpf der Liste entspricht dem Stapel, den man erhält, wenn man die oberste Karte wegnimmt. Die Länge der Liste ist die Anzahl der Karten im Stapel. Konkatenation setzt zwei Stapel aufeinander.

4.1.1 Listentypen

Wir zeigen anhand von Beispielen, wie Listen in Standard ML dargestellt werden:

```
[1, 2, 3]
```
[1, 2, 3] : int list

```
[2, 3] @ [4, 5, 6]
```
[2, 3, 4, 5, 6] : int list

```
op@
```
*fn : α list * α list → α list*

```
[1.0, 2.0, 3.0]
```
[1.0, 2.0, 3.0] : real list

```
[(2,3), (5,1), (7,2)]
```
*[(2,3), (5,1), (7,2)] : (int * int) list*

```
[[2, 3], [4, 5, 6], []]
```
[[2, 3], [4, 5, 6], []] : int list list

Eine Liste über einem Typ t hat also den Typ t *list*. Die von Standard ML verwendete umgekehrte Schreibweise für Listentypen ist gewöhnungsbedürftig: Statt *int list list* würde man eigentlich *list*(*list*(*int*)) erwarten.

Wenn t ein Typ mit Gleichheit ist, dann ist auch der Listentyp *t list* ein Typ mit Gleichheit:

```
[1,2,3,4] = [1] @ [2,3,4]
```
true : bool

4.1.2 Nil und Cons

Wenn wir Listen mit runden Klammern beschreiben, bekommen wir die falschen Typen:

```
(1,(2,()))
```
*(1,(2,())) : int * (int * unit)*

Um dieses Problem zu umgehen, gibt es eine eigene Konstante *nil* für die leere Liste und einen speziellen Operator :: für die Konstruktion nichtleerer Listen:

```
val xs = 1::(2::nil)
```
val xs = [1, 2] : int list

```
val ys = 0::xs
```
val ys = [0, 1, 2] : int list

Der Operator :: heißt **Cons**. Sowohl *nil* als auch Cons sind polymorph getypt:

$$nil : \forall\alpha.\ \alpha\ list$$

$$:: : \forall\alpha.\ \alpha * \alpha\ list \to \alpha\ list$$

Die Operatoren :: und @ klammern im Gegensatz zu den arithmetischen Operatoren rechts:

$$e_1 :: e_2 :: e_3 \quad \rightsquigarrow \quad e_1 :: (e_2 :: e_3)$$

$$e_1 @ e_2 @ e_3 \quad \rightsquigarrow \quad e_1 @ (e_2 @ e_3)$$

Außerdem klammern @ und :: gleichberechtigt:

$$e_1 :: e_2 @ e_3 \quad \rightsquigarrow \quad e_1 :: (e_2 @ e_3)$$

$$e_1 @ e_2 :: e_3 \quad \rightsquigarrow \quad e_1 @ (e_2 :: e_3)$$

Hier sind Beispiele:

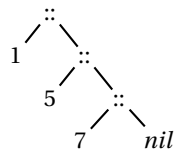
```
1::2::3::nil = [1,2,3]
```
true : bool

```
1::2::3::nil = 1::[2,3]
```
true : bool

```
[1,2,3] @ [3,5,6] @ [2,6,9]
```
[1, 2, 3, 3, 5, 6, 2, 6, 9] : int list

```
1::2::3::nil @ 4::5::nil
```
[1, 2, 3, 4, 5] : int list

Unter der **Baumdarstellung einer Liste** versteht man die zweidimensionale Darstellung der Liste mit *nil* und Cons. Beispielsweise hat die Liste $[1,5,7]$ die Baumdarstellung



Aufgabe 4.1 Geben Sie einen Ausdruck an, der die Liste $[7,2,4]$ klammerfrei mit Cons und *nil* beschreibt. Geben Sie die Baumdarstellung ihres Ausdrucks an. Unterscheidet sich die Baumdarstellung des Ausdrucks von der Baumdarstellung der Liste?

Aufgabe 4.2 Betrachten Sie den Ausdruck $1::2::nil@3::4::nil$.

a) Geben Sie die Baumdarstellung des Ausdrucks an.

b) Geben Sie die Baumdarstellung der beschriebenen Liste an.

c) Geben Sie die beschriebene Liste mit „[…]“ an.

Aufgabe 4.3 Macht es für die dargestellten Listen einen Unterschied, wie die folgenden Ausdrücke geklammert sind?

a) $(e_1::e_2)@e_3$ oder $e_1::(e_2@e_3)$.

b) $(e_1@e_2)@e_3$ oder $e_1@(e_2@e_3)$.

c) $(e_1::e_2)::e_3$ oder $e_1::(e_2::e_3)$.

4.1.3 Regelbasierte Prozeduren

Wir wollen jetzt eine polymorphe Prozedur $length : \alpha\ list \to int$ schreiben, die die Länge einer Liste liefert. Wie üblich suchen wir zuerst nach passenden Rekursionsgleichungen:

$$|nil| = 0$$
$$|x::xr| = 1 + |xr|$$

Die erste Gleichung liefert die Länge der leeren Liste. Die zweite Gleichung liefert die Länge nichtleerer Listen. Die damit verbundene Rekursion terminiert, da die Schachtelungstiefe (also die Länge) der Liste jeweils um eins reduziert wird.

Bei der Umsetzung der Rekursionsgleichungen in eine Prozedur gehen wir einen neuen Weg. Statt die beiden Gleichungen durch ein Konditional zu einer Gleichung zu verbinden, machen wir davon Gebrauch, dass man Prozeduren in Standard ML auch direkt durch mehrere Rekursionsgleichungen beschreiben kann:

```
fun length nil     = 0
  | length (x::xr) = 1 + length xr
```
length : α list → int

```
length [1,1,1,1]
```
4 : int

Wir sprechen von einer **regelbasierten Prozedur** mit zwei **Regeln**. Normale Prozeduren kann man als regelbasierte Prozeduren mit nur einer Regel auffassen.

4.2 Append, Rev, Concat und Tabulate

Wir zeigen jetzt anhand weiterer Beispiele, wie man mit Listen und regelbasierten Prozeduren programmiert.

Append

Die Konkatenation zweier Listen können wir gemäß der Gleichungen

$$nil @ ys = ys$$
$$(x :: xr) @ ys = x :: (xr @ ys)$$

bestimmen. Die durch die zweite Gleichung eingeführte Rekursion terminiert, da die Länge der linken Liste jeweils um eins reduziert wird. Die Umsetzung der Gleichungen in eine regelbasierte Prozedurdeklaration ist einfach:

```
fun append (nil,   ys) = ys
  | append (x::xr, ys) = x::append(xr,ys)
```
*val append : α list * α list → α list*

```
append([2,3], [6,7,8])
```
[2, 3, 6, 7, 8] : int list

Interpreter realisieren den Konkatenationsoperator @ übrigens mit einer *append* entsprechenden rekursiven Prozedur.

Rev

Die Prozedur

```
fun rev nil     = nil
  | rev (x::xr) = rev xr @ [x]
```
val rev : α list → α list

reversiert eine Liste durch Umkehrung der Elementreihenfolge:

```
rev [1, 2, 3, 4]
```
[4, 3, 2, 1] : int list

Die Rekursion terminiert, da die Länge der Liste jeweils um eins reduziert wird.

Concat

Die Prozedur

```
fun concat nil     = nil
  | concat (x::xr) = x @ concat xr
```
val concat : α list list → α list

liefert zu einer Liste von Listen die Konkatenation der Elementlisten:

$$concat\ [xs_1, \ldots, xs_n] \;=\; xs_1 @ \ldots @ xs_n$$

Tabulate

Die höherstufige Prozedur

```
fun tabulate (n,f) = iterdn (n-1) 0 nil (fn (i,xs) => f i::xs)
```
*val tabulate : int * (int → α) → α list*

liefert für eine Zahl $n \geq 0$ und eine Prozedur f eine Liste der Länge n wie folgt:

$$tabulate(n, f) \;=\; [f(0), \ldots, f(n-1)]$$

Hier sind Beispiele:

```
tabulate(5, fn x => x)
```
[0, 1, 2, 3, 4] : int list

```
tabulate(5, fn x => x*x)
```
[0, 1, 4, 9, 16] : int list

Vordeklarierte Prozeduren

Damit man die grundlegenden Prozeduren für Listen nicht jedes Mal neu deklarieren muss, sind diese beim Start des Interpreters bereits **vordeklariert**:

length	: $\alpha\ list \to int$
rev	: $\alpha\ list \to \alpha\ list$
List.concat	: $\alpha\ list\ list \to \alpha\ list$
List.tabulate	: $int * (int \to \alpha) \to \alpha\ list$

Aufgabe 4.4 (Enum) Schreiben Sie mithilfe der Prozedur *iterdn* (§ 3.13) eine Prozedur $enum : int \to int \to int\ list$, die zu zwei Zahlen $m \leq n$ die Liste $[m, \ldots, n]$ liefert. Beispielsweise soll $enum\ 3\ 6 = [3,4,5,6]$ gelten. Für $m > n$ soll *enum* die leere Liste liefern.

Aufgabe 4.5 (Member) Schreiben Sie eine polymorphe Prozedur

$$member : ''a \to ''a\ list \to bool$$

die testet, ob ein Wert als Element in einer Liste vorkommt.

4.3 Map, Filter, Exists und All

Wir besprechen jetzt die vordeklarierten Prozeduren

$$\begin{array}{ll} \mathit{map} & : (\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list} \\ \mathit{List.filter} & : (\alpha \to \mathit{bool}) \to \alpha\ \mathit{list} \to \alpha\ \mathit{list} \\ \mathit{List.exists} & : (\alpha \to \mathit{bool}) \to \alpha\ \mathit{list} \to \mathit{bool} \\ \mathit{List.all} & : (\alpha \to \mathit{bool}) \to \alpha\ \mathit{list} \to \mathit{bool} \end{array}$$

Diesen höherstufigen Prozeduren ist gemeinsam, dass sie zunächst auf eine Prozedur und dann auf eine Liste angewendet werden.

Map

Die Prozedur

```
fun map f nil     = nil
  | map f (x::xr) = (f x) :: (map f xr)
```
map : (α → β) → α list → β list

wendet eine Prozedur f auf alle Elemente einer Liste an:

$$\mathit{map}\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

Hier sind Beispiele:

```
map (fn x => 2*x) [2, 4, 11, 34]
```
[4, 8, 22, 68] : int list

```
map op~ [2, 4, 11, 34]
```
[~2, ~4, ~11, ~34] : int list

```
fun minus5 xs = map (fn x => x-5) xs
```
val minus5 : int list → int list

```
minus5 [3, 6, 99, 72]
```
[~2, 1, 94, 67] : int list

```
map rev [[3, 4, 5], [2, 3], [7, 8, 9]]
```
[[5, 4, 3], [3, 2], [9, 8, 7]] : int list list

```
map length [[3, 4, 5], [2, 3], [7, 8, 9]]
```
[3, 2, 3] : int list

Aufgabe 4.6 Wie typisiert die ambige Deklaration *val f = map rev* den Bezeichner *f*? Wie müssen Sie die Deklaration umschreiben, damit *f* polymorph typisiert wird?

Filter

Die Prozedur

```
fun filter f nil     = nil
  | filter f (x::xr) = if f x then x :: filter f xr
                       else filter f xr
```
val filter : (α → bool) → α list → α list

liefert zu einer Liste die Liste der Elemente, für die eine gegebene Prozedur *true* liefert:

```
filter (fn x => x<0) [0, ~1, 2, ~3, ~4]
```
[~1, ~3, ~4] : int list

```
filter (fn x => x>=0) [0, ~1, 2, ~3, ~4]
```
[0, 2] : int list

Exists und All

Die Prozedur

```
fun exists f nil     = false
  | exists f (x::xr) = f x orelse exists f xr
```
exists : (α → bool) → α list → bool

testet, ob eine gegebene Prozedur für mindestens ein Element einer Liste *true* liefert:

```
exists (fn x => x<0) [1, 2, 3]
```
false : bool

```
exists (fn x => x<0) [1, ~2, 3]
```
true : bool

Die Prozedur

```
fun all f nil     = true
  | all f (x::xr) = f x andalso all f xr
```
all : (α → bool) → α list → bool

testet, ob eine gegebene Prozedur für alle Elemente einer Liste *true* liefert:

```
all (fn x => x>0) [1, 2, 3]
```
true : bool

```
all (fn x => x>0) [1, ~2, 3]
```
false : bool

Aufgabe 4.7 (Member) Schreiben Sie mithilfe von *List.exists* eine Prozedur *member* : ′*a* → ′*a list* → *bool*, die testet, ob ein Wert als Element in einer Liste vorkommt.

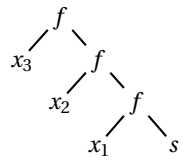
4.4 Faltung

Faltung ist ein Rekursionsschema, das für Listen eine vergleichbar wichtige Rolle spielt wie bestimmte Iteration für Zahlen. Faltung wird in Standard ML durch eine vordeklarierte Prozedur

$$foldl : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$$

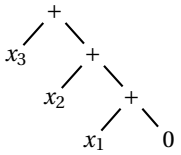
realisiert. Der Ausdruck $foldl\ f\ s$ beschreibt dabei eine Prozedur, die Listen gemäß einer **Verknüpfung** f und einem **Startwert** s wie folgt faltet:

$$foldl\ f\ s\ [x_1, x_2, x_3] \ = \ f(x_3, f(x_2, f(x_1, s))) \ =$$



Um ein konkreteres Beispiel zu haben, falten wir die Liste $[x_1, x_2, x_3]$ mit der Verknüpfung + und dem Startwert 0:

$$foldl\ op{+}\ 0\ [x_1, x_2, x_3] \ = \ x_3 + (x_2 + (x_1 + 0)) \ =$$



Diese Faltung summiert die Elemente der Liste. Also deklariert

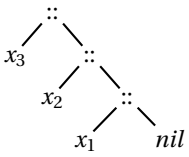
```
fun sum xs = foldl op+ 0 xs
```
val sum : int list → int

eine Prozedur, die die Elemente ganzzahliger Listen summiert:

```
sum [4,3,6,2,8]
```
23 : int

Unser zweites Beispiel reversiert eine Liste durch Falten mit der Verknüpfung Cons und dem Startwert *nil*:

$$foldl\ op{::}\ nil\ [x_1, x_2, x_3] \ = \ x_3 :: (x_2 :: (x_1 :: nil)) \ =$$



Also liefert der Einzeiler

```
fun rev xs = foldl op:: nil xs
```
rev : α list → α list

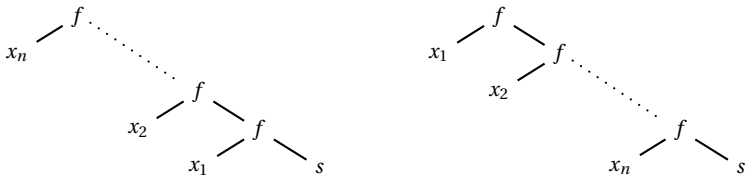
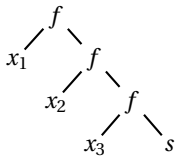


Abbildung 4.1: *Grafische Darstellung von foldl und foldr*

eine Prozedur, die Listen reversiert.

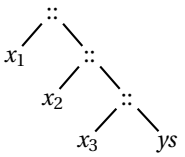
Eine Variante von *foldl* ist die Prozedur *foldr*, die Listen von rechts nach links faltet:

$$foldr\ f\ s\ [x_1,x_2,x_3] \;=\; f(x_1,\,f(x_2,\,f(x_3,s))) \;=$$



Aus dem Beispiel

$$foldr\ op{::}\ ys\ [x_1,x_2,x_3] \;=\; x_1 :: (x_2 :: (x_3 :: ys)) \;=$$



können wir ersehen, dass der Einzeiler

```
fun append (xs,ys) = foldr op:: ys xs
val append : α list * α list → α list
```

eine Prozedur liefert, die Listen konkateniert.

Abbildung 4.1 stellt die durch *foldl* und *foldr* realisierten Rekursionsschemen grafisch dar. Aus dieser Darstellung ist ersichtlich, dass sich *foldl* und *foldr* nur durch die Reversion der Argumentliste unterscheiden:

$$\begin{aligned} foldr\ f\ s\ [x_1,\ldots,x_n] &= foldl\ f\ s\ [x_n,\ldots,x_1] \\ foldl\ f\ s\ [x_1,\ldots,x_n] &= foldr\ f\ s\ [x_n,\ldots,x_1] \end{aligned}$$

Abbildung 4.2 deklariert einige der bisher eingeführten Listenprozeduren mithilfe der Faltungsprozeduren.[1]

Zu guter Letzt zeigen wir noch, wie die Faltungsprozeduren deklariert werden können:

[1] Die hier verwendete Sprechweise ist bequem aber ungenau. Korrekt müssten wir sagen: Abbildung 4.2 deklariert zu einigen der bisher eingeführten Listenprozeduren semantisch äquivalente Prozeduren mithilfe der Faltungsprozeduren.

```
fun length xs = foldl (fn (x,n) => n+1) 0 xs

fun append (xs,ys) = foldr op:: ys xs

fun rev xs = foldl op:: nil xs

fun concat xs = foldr op@ nil xs

fun map f = foldr (fn (x,yr) => (f x)::yr) nil

fun filter f = foldr (fn (x,ys) => if f x then x::ys else ys) nil
```

Abbildung 4.2: *Rückführung von Listenprozeduren auf Faltung*

```
fun foldl f s nil     = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
```
*val foldl : (α * β → β) → β → α list → β*

```
fun foldr f s nil     = s
  | foldr f s (x::xr) = f(x, foldr f s xr)
```
*val foldr : (α * β → β) → β → α list → β*

Wenn man zwischen *foldl* und *foldr* wählen kann, sollte man sich aus Effizienzgründen für das endrekursive *foldl* entscheiden.

Aufgabe 4.8 (Produkt) Schreiben Sie mit *foldl* eine Prozedur *prod* : *int list* → *int*, die das Produkt der Elemente einer Liste liefert. Für die leere Liste soll 1 geliefert werden.

Aufgabe 4.9 (Member) Schreiben Sie mithilfe von *foldl* eine polymorphe Prozedur *member* : $''a \to ''a\ list \to bool$, die testet, ob ein Wert als Element in einer Liste vorkommt.

Aufgabe 4.10 (Count) Schreiben Sie mithilfe von *foldl* eine polymorphe Prozedur *count* : $''a \to ''a\ list \to int$, die zählt, wie oft ein Wert in einer Liste als Element vorkommt. Beispielsweise soll *count* 5 $[2,5,3,5] = 2$ gelten.

Aufgabe 4.11 (Dezimaldarstellung) Die Dezimaldarstellung einer natürlichen Zahl ist die Liste ihrer Ziffern. Beispielsweise hat 7856 die Dezimaldarstellung $[7,8,5,6]$.

a) Deklarieren Sie eine Prozedur *dec* : *int* → *int list*, die die Dezimaldarstellung einer natürlichen Zahl liefert. Verwenden Sie *div* und *mod*.

b) Deklarieren Sie mithilfe von *foldl* eine Prozedur *num* : *int list* → *int*, die zu einer Dezimaldarstellung die dargestellte Zahl liefert.

Aufgabe 4.12 Deklarieren Sie die Faltungsprozedur *foldr* mithilfe der Faltungsprozedur *foldl*. Verwenden Sie dabei keine weitere rekursive Hilfsprozedur.

Aufgabe 4.13 Deklarieren Sie die Faltungsprozedur *foldl* mithilfe der Faltungsprozedur *foldr*. Gehen Sie wie folgt vor:

a) Deklarieren Sie *append* mithilfe von *foldr*.

b) Deklarieren Sie *rev* mithilfe von *foldr* und *append*.

c) Deklarieren Sie *foldl* mithilfe von *foldr* und *rev*.

d) Deklarieren Sie *foldl* nur mithilfe von *foldr*.

Aufgabe 4.14 (Challenge) Die Rückführung von *foldl* auf *foldr* gelingt auf besonders elegante Weise, wenn man *foldr* auf einen prozeduralen Startwert anwendet.

a) Finden Sie zwei Abstraktionen e und e', die die folgende Gleichung erfüllen:

$$foldl\ f\ s\ xs = (foldr\ e\ e'\ xs)\ s$$

Die Abstraktionen sollen keine Hilfsprozeduren verwenden. Machen Sie sich klar, dass $foldr\ e\ e'\ xs$ eine zu $fn\ s \Rightarrow foldl\ f\ s\ xs$ äquivalente Prozedur liefern muss. Daher ergibt sich e' aus dem Spezialfall $xs = nil$ und e aus dem Spezialfall $xs = x :: xr$.

b) Überzeugen Sie sich davon, dass Ihre Abstraktionen die obige Gleichung auch dann erfüllen, wenn *foldl* mit *foldr* vertauscht wird.

c) Deklarieren Sie die Prozedur *foldl* gemäß der obigen Gleichung mithilfe der Prozedur *foldr*.

4.5 Hd, Tl, Null, Nth und das Werfen von Ausnahmen

Die vordeklarierten Prozeduren

$$hd : \alpha\ list \to \alpha$$
$$tl : \alpha\ list \to \alpha\ list$$

liefern den Kopf (engl. head) und den Rumpf (engl. tail) nichtleerer Listen:

```
hd [1,2,3,4,5]
```
1 : int

```
tl [1,2,3,4,5]
```
[2, 3, 4, 5] : int list

Wenn *hd* und *tl* auf die leere Liste angewendet werden, **werfen sie die Ausnahme** *Empty*:

```
hd nil
```
! Uncaught exception: Empty

```
tl nil
```
! Uncaught exception: Empty

Die Prozeduren *hd* und *tl* können wie folgt deklariert werden:

```
fun hd nil     = raise Empty
  | hd (x::xr) = x
val hd : α list → α
```

```
fun tl nil     = raise Empty
  | tl (x::xr) = xr
val tl : α list → α list
```

Das Werfen der Ausnahme *Empty* bei der Anwendung auf die leere Liste erfolgt mit dem Ausdruck *raise Empty*. Da dieser Ausdruck keinen Wert liefert, kann er jeden Typ annehmen. Wenn bei der Ausführung eines Programms eine Ausnahme geworfen wird, bricht der Interpreter die Ausführung des Programms mit einer entsprechenden Fehlermeldung ab. Später werden wir ein Sprachkonstrukt kennenlernen, mit dem geworfene Ausnahmen gefangen werden können.

Test auf die leere Liste

Die vordeklarierte Prozedur

```
fun null nil     = true
  | null (x::xr) = false
val null : α list → bool
```

testet, ob eine Liste leer ist.

Mithilfe der Prozeduren *hd*, *tl* und *null* können wir Listenprozeduren auch ohne die Verwendung von mehreren Regeln deklarieren. Wir zeigen das am Beispiel von *append*:

```
fun append (xs,ys) = if null xs then ys
                     else hd xs :: append(tl xs, ys)
val append : α list * α list → α list
```

Auf den ersten Blick scheint es, dass man statt *null xs* auch *xs* = *nil* schreiben kann. Das ist aber keineswegs der Fall, da *xs* = *nil* im Gegensatz zu *null xs* nur dann zulässig ist, wenn der Elementtyp von *xs* ein Typ mit Gleichheit ist (siehe § 3.7). Folglich liefert die Deklaration

```
fun append (xs,ys) = if xs=nil then ys
                     else hd xs :: append(tl xs, ys)
val append : ''a list * ''a list → ''a list
```

eine Prozedur mit einem auf Gleichheitstypen eingeschränkten Typschema.

Positionen und Nth

Beim Programmieren ist es üblich, die **Positionen einer nichtleeren Liste** mit null beginnend durchzunummerieren: $[x_0, \ldots, x_m]$. Eine Liste der Länge n hat dann die Positionen 0 bis $n-1$. Unter dem n**-ten Element** einer Liste versteht man dementsprechend das Element an der Position n.

Hier ist eine unter *List.nth* vordeklarierte Prozedur, die das n-te Element einer Liste liefert:

```
fun nth(xs,n) = if n<0 orelse null xs then raise Subscript
                else if n=0 then hd xs else nth(tl xs, n-1)
```
*val nth : α list * int → α*

```
nth ([3,4,5], 0)
```
3 : int

```
nth ([3,4,5], 2)
```
5 : int

```
nth ([3,4,5], 3)
```
! Uncaught exception: Subscript

Die Ausnahme *Subscript* wird zur Signalisierung von Grenzüberschreitungen verwendet.

Die mit null beginnende Nummerierung beißt sich mit der üblichen mathematischen Nummerierung, die bei eins beginnt. Sie müssen also immer darauf achten, welche Nummerierung gerade verwendet wird.

Aufgabe 4.15 (Last) Deklarieren Sie eine Prozedur *last* : α *list* $\rightarrow \alpha$, die das Element an der letzten Position einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme *Empty* geworfen werden.

Aufgabe 4.16 (Max) Schreiben Sie mit *foldl* eine Prozedur *max*: *int list* → *int*, die das größte Element einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme *Empty* geworfen werden. Verwenden Sie die vordefinierte Prozedur *Int.max*: *int* ∗ *int* → *int*.

Aufgabe 4.17 (Take und Drop) Schreiben Sie zwei polymorphe Prozeduren *take* und *drop*, die gemäß α *list* ∗ *int* → α *list* getypt sind und die folgende Spezifikation erfüllen:

a) *take*(xs, n) liefert die ersten n Elemente der Liste *xs*. Falls $n < 0$ oder $|xs| < n$ gilt, soll die Ausnahme *Subscript* geworfen werden.

b) *drop*(xs, n) liefert die Liste, die man aus xs erhält, wenn man die ersten n Elemente weglässt. Falls $n < 0$ oder $|xs| < n$ gilt, soll die Ausnahme *Subscript* geworfen werden.

Hinweis: Verwenden Sie *orelse* und die Prozeduren *hd*, *tl* und *null*.

4.6 Regelbasierte Prozeduren und Musterabgleich

Durch die Verwendung mehrerer Regeln kann bei vielen Prozeduren für Listen auf die Verwendung von Konditionalen und der Prozeduren *null*, *hd* und *tl* verzichtet werden. Ein typisches Beispiel ist die Prozedur *rev*, die Listen gemäß zweier Regeln reversiert:

```
fun rev nil     = nil
  | rev (x::xr) = rev xr @ [x]
val rev : α list → α list
```

Die Regeln der Prozedur stellen eine Fallunterscheidung dar. Bei der Ausführung eines Aufrufs wird die zum Argument des Aufrufs passende Regel gewählt.

Prozeduren mit nur einer Regel entsprechen den Prozeduren, die wir bisher kennengelernt haben. Insofern ist jede Prozedur regelbasiert.

Eine **Regel** besteht aus einem Bezeichner, einem Muster und einem Ausdruck, der als **Rumpf** der Regel bezeichnet wird:

$$\langle \textit{Bezeichner} \rangle \ \langle \textit{Muster} \rangle \ = \ \underbrace{\langle \textit{Ausdruck} \rangle}_{\text{Rumpf}}$$

Das Muster einer Regel entscheidet darüber, ob die Regel auf ein Argument anwendbar ist. Das Muster kann Bezeichner einführen, die als **Variablen** des Musters und der Regel bezeichnet werden. Ein Bezeichner darf nur einmal in einem Muster auftreten, und dieses Auftreten ist immer definierend.

Wenn ein Wert gemäß eines Musters aufgebaut ist, sagen wir, dass der Wert das Muster **trifft**. Umgekehrt sagen wir auch, dass das Muster den Wert trifft. Wenn ein Muster einen Wert trifft, können die Variablen des Musters an Teilwerte des Wertes gebunden werden. Beispielsweise trifft das Muster $(x :: xs)$ den Wert $[7,8,9]$ und bindet dabei x und xs an 7 und $[8,9]$.

Der Prozess, der entscheidet, ob ein Muster einen Wert trifft und dabei die Variablen des Musters bindet, wird als **Musterabgleich** bezeichnet. Im Englischen spricht man von **pattern matching**.

Muster dürfen Konstanten sowie runde und eckige Klammern enthalten. Außerdem können Prozeduren mit mehr als zwei Regeln formuliert werden. Hier ist eine Prozedur mit 4 Regeln:

```
fun test [] = 0
  | test [(x,y)] = x+y
  | test [(x,5), (7,y)] = x*y
  | test (_::_::ps) = test ps
val test : (int * int) list → int
```

Aufgabe 4.18 Welche der Muster der 4 Regeln der Prozedur *test* treffen den Wert $[(2,5),\ (7,3)]$? An welche Werte werden die Variablen der Muster dabei gebunden?

Aufgabe 4.19 Entscheiden Sie für jeden der folgenden Werte, ob er das Muster $(x,\ y :: \_ :: z,\ (u,3))$ trifft. Geben Sie bei einem Treffer die Bindungen für die Variablen des Musters an.

a) $(7,\ [1],\ (3,3))$

b) $([1,2],\ [3,4,5],\ (11,3))$

4.6.1 Disjunkte und überlappende Regeln

Eine Menge von Mustern heißt **disjunkt**, wenn die Muster jeweils verschiedene Werte treffen. Präziser können wir sagen, dass eine Menge von Mustern disjunkt ist, wenn es keinen Wert gibt, der mehr als ein Muster der Menge trifft. Beispielsweise sind die Muster

$$[],\quad [(x,y)],\quad [(x,5),\ (7,y)],\quad (\_ :: (x,\_) :: \_ :: ps)$$

disjunkt. Muster, die nicht disjunkt sind, bezeichnen wir als **überlappend**. Hier sind zwei überlappende Muster:

$$[(x,5),\ (7,y)],\quad (\_ :: (x,\_) :: ps)$$

Eine Menge von Regeln heißt **disjunkt** [**überlappend**], wenn ihre Muster disjunkt [überlappend] sind. Wenn die Regeln einer Prozedur disjunkt sind, spielt die Reihenfolge, in der sie angegeben sind, keine Rolle. Statt

```
fun length nil     = 0
  | length (_::xr) = 1 + length xr
```

können wir also genauso gut

```
fun length (_::xr) = 1 + length xr
  | length nil     = 0
```

schreiben.

Wenn die Regeln einer Prozedur überlappend sind, ist ihre Reihenfolge von Bedeutung. In diesem Fall wird eine Regel nämlich nur dann angewendet, wenn keine der vor ihr stehenden Regeln anwendbar ist. Hier ist eine Prozedur, die von dieser Tatsache Gebrauch macht, um zu testen, ob eine Liste mindestens zwei Positionen hat:

```
fun test (_::_::_) = true
  | test     _     = false
val test : α list → bool
```

Mit überlappenden Regeln kann man auch Prozeduren für Zahlen mit mehreren Regeln schreiben. Beispielsweise liefert die Prozedur

```
fun power (x, 0) = 1
  | power (x, n) = x * power(x, n-1)
val power : int * int → int
```

für x und $n \geq 0$ die Potenz x^n. Für negative n divergiert die Prozedur.

Die Rekursionsgleichungen einer Prozedur lassen sich in Standard ML allerdings nur dann direkt durch Regeln darstellen, wenn sie ohne Anwendungsbedingungen formuliert sind.[2] Wenn dagegen wie bei

$$\begin{aligned} &potenz(x,n) = 1 && \text{für } n < 1 \\ &potenz(x,n) = x \cdot potenz(x,n-1) && \text{für } n \geq 1 \end{aligned}$$

Anwendungsbedingungen vorkommen, muss man die Rekursionsgleichungen durch ein Konditional verbinden:

```
fun potenz (x,n) = if n<1 then 1 else x*potenz(x,n-1)
```

Ein weiteres Beispiel für eine Prozedur mit überlappenden Regeln ist

```
fun or (false, false) = false
  | or       _        = true
```

*val or : bool * bool → bool*

Diese Prozedur liefert entsprechend dem logischen oder für zwei Boolesche Werte genau dann *true*, wenn mindestes einer der Werte *true* ist.

Aufgabe 4.20 Deklarieren Sie eine zu *or* äquivalente Prozedur, die mit disjunkten Regeln formuliert ist. Verwenden Sie dabei kein Konditional.

4.6.2 Erschöpfende Regeln

Eine Menge von Mustern **erschöpft** einen Typ, wenn jeder Wert des Typs mindestens eines der Muster trifft. Die Regeln einer Prozedur heißen **erschöpfend**, wenn ihre Muster den Argumenttyp der Prozedur erschöpfen.

Sie sollten nur Prozeduren mit erschöpfenden Regeln verwenden. Wenn Sie einen Fehler machen und die Regeln Ihrer Prozedur nicht erschöpfend sind, warnt Sie der Interpreter:

```
fun test nil = 0
  | test [_] = 1
val test : α list → int
! Warning: pattern matching is not exhaustive
```

Wenn die Prozedur *test* aber nur auf Listen mit höchstens einem Element angewendet werden soll, sollten Sie eine **Ausnahmeregel** hinzufügen, die bei einem Verstoß eine geeignete Ausnahme wirft:

[2] In der Programmiersprache Haskell können die Regeln von Prozeduren auch mit Anwendungsbedingungen formuliert werden.

```
exception SomethingWrong

fun test nil = 0
  | test [_] = 1
  | test _   = raise SomethingWrong
```
val test : α list → int

```
test [1,2]
```
! Uncaught exception: SomethingWrong

Auf das Deklarieren von Ausnahmen werden wir in § 6.5 näher eingehen.

4.6.3 Regelbasierte Abstraktionen und Case-Ausdrücke

Auch Abstraktionen können mit mehreren Regeln gebildet werden. Beispielsweise beschreibt

```
fn true  => false
 | false => true
```
fn : bool → bool

eine Prozedur, die einen Booleschen Wert negiert.

Ein **Case-Ausdruck**

$$case\ e\ of\ M_1 \Rightarrow e_1 \mid \ldots \mid M_n \Rightarrow e_n$$

ist eine abgeleitete Form, die die durch die Prozeduranwendung

$$(fn\ M_1 \Rightarrow e_1 \mid \ldots \mid M_n \Rightarrow e_n)\ e$$

realisierte Fallunterscheidung lesbarer darstellt. Hier ist ein Beispiel:

```
fun test xs y = case rev xs
                of _::x::_ => x<y
                |     _    => false
```
val test : int list → int → bool

Die Deklaration liefert eine Prozedur, die testet, ob das vorletzte Argument einer Liste kleiner als eine gegebene Zahl ist.

Übrigens ist ein Konditional *if* e_1 *then* e_2 *else* e_3 in Standard ML eine abgeleitete Form, die auf die Prozeduranwendung (*fn true* $\Rightarrow e_2 \mid$ *false* $\Rightarrow e_3$) e_1 zurückgeführt wird.

4.6.4 Kaskadierte Prozedurdeklarationen mit mehreren Regeln

Kaskadierte Prozedurdeklarationen mit mehreren Regeln werden mithilfe von Case-Ausdrücken auf kaskadierte Prozedurdeklarationen mit nur einer Regel zurückgeführt. Beispielsweise wird die Deklaration

```
fun or false false = false
  | or   _     _   = true
val or : bool → bool → bool
```

auf die Deklaration

```
fun or x y = case (x,y)
             of (false, false) => false
             |  (  _  ,   _  ) => true
```

zurückgeführt. Die Wahl der für die Ausführung eines Aufrufs zu verwendenden Regel erfolgt also erst dann, wenn beide Argumente vorliegen (x und y). Wenn wir auch die restlichen abgeleiteten Formen eliminieren, bekommen wir die Deklaration

```
fun or (x:bool) = fn (y:bool) =>
                     (fn (false , false ) => false
                      |  (a:bool, b:bool) => true ) (x,y)
```

4.7 Strings

Unter einem String versteht man in der Programmierung eine Zeichenfolge. Standard ML hat spezielle Konstanten für Strings:

```
"Saarbrücken"
"Saarbrücken" : string
```

```
"4567899999999 is a large number"
"4567899999999 is a large number" : string
```

```
"true ist ein Boolescher Wert"
"true ist ein Boolescher Wert" : string
```

Die Konstante `""` stellt den **leeren String**, dar.

Auch für Zeichen hat Standard ML einen eigenen Typ und eigene Konstanten:

```
#"M"
#"M" : char
```

```
[#"S", #"M", #"L"]
[#"S", #"M", #"L"] : char list
```

Die vordefinierten Prozeduren

val explode : string → char list

val implode : char list → string

konvertieren zwischen Strings und Zeichenlisten:

```
explode("Hallo")
```
[#"H", #"a", #"l", #"l", #"o"] : char list

```
implode [#"H", #"a", #"l", #"l", #"o"]
```
"Hallo" : string

Die Operation ^ liefert die **Konkatenation zweier Strings**:

```
"Programmier" ^ "sprache"
```
"Programmiersprache" : string

```
"Aller " ^ "Anfang " ^ "ist schwer."
```
"Aller Anfang ist schwer." : string

```
op^
```
*fn : string * string → string*

Mit *explode* und *implode* kann man eine semantisch äquivalente Prozedur zu *op^* schreiben:

```
fn (s,s') => implode(explode s @ explode s')
```

Aufgabe 4.21 Schreiben Sie eine Prozedur *size* : *string* → *int*, die die Länge eines Strings liefert (z.B. *size* "*hut*" = *3*).

Aufgabe 4.22 Schreiben Sie eine Prozedur *reverse* : *string* → *string*, die Strings reversiert (z.B. *reverse* "*hut*" = "*tuh*").

4.7.1 Zeichenstandards

Ein **Zeichenstandard** legt eine bestimmte Anzahl von Zeichen fest und regelt, wie die Zeichen grafisch darzustellen sind. Zusätzlich werden die Zeichen mit 0 beginnend durchnummeriert. Beispiele für gebräuchliche und international festgelegte Zeichenstandards sind ASCII (2^7 Zeichen), Latin-1 (2^8 Zeichen) und Unicode (2^{16} Zeichen). Die Standard ML Sprachdefinition legt sich auf keinen Zeichenstandard fest.

Die meisten Standard ML Interpreter benutzen den Zeichenstandard Latin-1. Wir tun das in diesem Buch auch. Die vordefinierten Prozeduren

val ord : char → int

val chr : int → char

konvertieren gemäß des vereinbarten Zeichenstandards zwischen Zeichen und natürlichen Zahlen:

```
ord #"1"
```
49 : int

```
chr 49
```
#"1" : char

```
map ord (explode "019abz")
```
[48, 49, 57, 97, 98, 122] : int list

```
chr 255
```
#"\255" : char

```
chr 256
```
! Uncaught exception: Chr

Es gibt einige Zeichen, die in den Konstanten für Zeichen und Strings nicht in der offensichtlichen Weise geschrieben werden können. Das gilt beispielsweise für das Anführungszeichen, das Tabulatorzeichen und das Zeichen für Zeilenwechsel. Diese **Sonderzeichen** werden mithilfe des Zeichens \ (sprich backslash) geschrieben:

\"	Anführungszeichen (")
\\	Backslash (\ )
\t	Tabulator
\n	Zeilenwechsel

Hier sind Beispiele für Konstanten, die mit Backslash dargestellte Sonderzeichen enthalten:

```
"\"\\\t\na"
```
"\"\\\t\na" : string

```
explode "\"\\\t\na"
```
[#"\"", #"\\", #"\t", #"\n", #"a"] : char list

Das Zeichen \ bewirkt, dass das nachfolgende Zeichen anders als üblich interpretiert wird. Zeichen mit dieser Funktion werden als **Fluchtsymbole** (engl. escape characters) bezeichnet.

Aufgabe 4.23 Schreiben Sie eine Prozedur *isDigit* : *char* → *bool*, die mithilfe der Prozedur *ord* testet, ob ein Zeichen eine der Ziffern $0,\ldots,9$ ist. Nützen Sie dabei aus, dass *ord* die Ziffern durch aufeinander folgende Zahlen darstellt.

Aufgabe 4.24 Schreiben Sie eine Prozedur *toInt* : *string* → *int*, die zu einem String, der nur aus Ziffern besteht, die durch ihn dargestellte Zahl liefert (z.B. *toInt* "*123*" = *123*). Falls der String Zeichen enthält, die keine Ziffern sind, soll die Ausnahme *Domain* geworfen werden.

4.7.2 Lexikalische Ordnung

Die Vergleichsoperatoren $<$, $\leq$, $>$ und $\geq$ sind auch für Zeichen und Strings definiert.[3] Die Vergleiche für Zeichen entsprechen den Vergleichen für die durch den Zeichenstandard zugeordneten Zahlen. Die Vergleiche für Strings sind **lexikalisch** gemäß der Ordnung für Zeichen definiert:

```
"Adam" < "Eva"
```
true : bool

```
"Adam" < "Adamo"
```
true : bool

Das Ergebnis des Vergleichs

```
"rufen" < "Zukunft"
```
false : bool

ist zunächst überraschend, da „rufen“ im Lexikon vor „Zukunft“ steht. Die Erklärung ist einfach: Große Buchstaben werden mit kleineren Zahlen dargestellt als kleine Buchstaben:

```
map ord (explode "ABab")
```
[65, 66, 97, 98] : int list

Aufgabe 4.25 Schreiben Sie mithilfe der Prozeduren *explode* und *ord* eine Prozedur *less* : *string* → *string* → *bool*, die für zwei Strings s, s' dasselbe Ergebnis liefert wie $s < s'$.

Bemerkungen

In diesem Kapitel haben wir am Beispiel von Listen erstmals eine rekursive Datenstruktur kennengelernt, deren Werte durch Konstruktoren gebildet werden. Solche Datenstrukturen spielen in der Programmierung eine wichtige Rolle und werden uns später in allgemeiner Form beschäftigen.

Die Darstellung von Listen durch die Konstruktoren *nil* und Cons bildet die Grundlage für die regelbasierte Deklaration von Prozeduren für Listen. Regelbasierte Prozeduren gehören zum Sprachkern von Standard ML.

Für Listen haben wir das Rekursionsschema Faltung kennengelernt, auf das viele Prozeduren für Listen zurückgeführt werden können.

Wir haben auch den Typ *string* betrachtet, dessen Werte Zeichenfolgen darstellen und als Strings bezeichnet werden. Entsprechend ihrer praktischen Bedeutung gibt es spezielle Konstanten für Strings. Außerdem gibt es einen Typ *char*, bei dessen Werten es

[3] Insgesamt sind die Vergleichsoperationen vierfach überladen, da sie auch für die Typen *int* und *real* definiert sind.

sich um die für Strings zulässigen Zeichen handelt. Wenn wir wollen, können wir den Typ *string* als eine Variante des Typs *char list* auffassen. Operationen auf Strings können mithilfe der Prozeduren *explode* und *implode* programmiert werden, die Strings in Zeichenlisten und Zeichenlisten in Strings übersetzen.

Verzeichnis

Listen: Listen über t; Kopf, Rumpf, Elemente, Positionen, und Länge $|xs|$; Konkatenieren ($xs@ys$), Reversieren und Falten; Darstellung mit eckigen Klammern; Darstellung mit *nil* und Cons; Baumdarstellung.

Vordeklarierte Prozeduren: hd, tl, null; length, rev, map; List.nth, List.tabulate, List.all, List.exists, List.filter, List.concat; foldl, foldr.

Ausnahmen werfen: raise Empty, raise Subscript.

Muster und Musterabgleich: Variable; Muster trifft Wert und Wert trifft Muster; disjunkte, überlappende und erschöpfende Muster.

Regelbasierte Prozeduren: Regeln mit Muster, Rumpf und Variablen; disjunkte, überlappende und erschöpfende Regeln; Ausnahmeregel; regelbasierte Abstraktionen und kaskadierte Prozedurdeklarationen mit mehreren Regeln.

Case-Ausdrücke.

Strings: explode, implode, ord, chr, Konkatenation ($s\hat{}s'$), Zeichenstandards, Sonderzeichen, Fluchtsymbole, lexikalische Ordnung.

5 Sortieren

In diesem Kapitel geht es in erster Linie um Algorithmen, die Listen durch Umordnen ihrer Elemente sortieren. Wir realisieren diese Algorithmen durch polymorphe Prozeduren, die zu einer Vergleichsprozedur für α eine Sortierprozedur für Listen über α liefern. Außerdem beschäftigen wir uns mit der Darstellung von Mengen durch strikt sortierte Listen und entwickeln einen Algorithmus für die Primzerlegung natürlicher Zahlen.

5.1 Sortieren durch Einfügen

Eine Liste $[x_1,\ldots,x_n]$ über *int* heißt **sortiert**, wenn sie ihre Elemente in aufsteigender Ordnung enthält: $x_1 \le \cdots \le x_n$. Jede Liste kann durch Umordnung ihrer Elemente in eine sortierte Liste überführt werden, die eindeutig bestimmt ist. Die zu einer Liste *xs* gehörige sortierte Liste heißt **Sortierung von** *xs* und wird mit *sort xs* bezeichnet. Hier sind Beispiele:

$$\begin{aligned} \mathit{sort}\ [] &= [] \\ \mathit{sort}\ [x] &= [x] \\ \mathit{sort}\ [3,4,1,4,2,3] &= [1,2,3,3,4,4] \end{aligned}$$

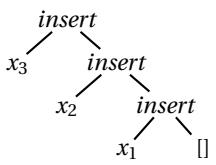
Eine Liste zu **sortieren** bedeutet, die Anordnung der Elemente so zu verändern, dass die Elemente in aufsteigender Ordnung erscheinen. Zwei Listen sind genau dann bis auf die Anordnung ihrer Elemente gleich, wenn sie die gleiche Sortierung haben.

Es gibt verschiedene Sortieralgorithmen. Im Folgenden beschreiben wir einen Algorithmus, der als Sortieren durch Einfügen bezeichnet wird.

Sortieren durch Einfügen beruht auf einer Operation *insert*, die einen Wert so in eine sortierte Liste einfügt, dass sich wieder eine sortierte Liste ergibt:

$$\mathit{insert}(3,\ [0,1,2,3,4,5]) = [0,1,2,3,3,4,5]$$

Eine Liste wird nun dadurch sortiert, dass ihre Elemente schrittweise in die sortierten Listen eingefügt werden, die sich ausgehend von der leeren Liste ergeben:

$\mathit{sort}\ [x_1,x_2,x_3]\ =$ 

$= \mathit{foldl}\ \mathit{insert}\ []\ [x_1,x_2,x_3]$

```
fun insert (x, nil)   = [x]
  | insert (x, y::yr) = if x<=y then x::y::yr else y::insert(x,yr)
```
*val insert : int * int list → int list*

```
fun isort xs = foldl insert nil xs
```
val isort : int list → int list

Abbildung 5.1: *Sortieren durch Einfügen*

Die Prozedur für die Einfügeoperation *insert* ergibt sich aus den folgenden Gleichungen:

$$\begin{aligned} \mathit{insert}(x,\ []) &= [x] \\ \mathit{insert}(x,\ y::yr) &= x::y::yr && \text{für } x \le y \\ \mathit{insert}(x,\ y::yr) &= y::\mathit{insert}(x,yr) && \text{für } x > y \end{aligned}$$

Abbildung 5.1 zeigt zwei Prozeduren, die die Einfügeoperation und den Sortieralgorithmus realisieren.

Die Essenz von Sortieren durch Einfügen kann durch die folgende Gleichung formuliert werden:

$$\mathit{sort}\,(x::xr) = \mathit{insert}(x,\ \mathit{sort}\ xr)$$

Wir unterscheiden zwischen **aufsteigender** und **absteigender Sortierung**. Bisher haben wir aufsteigende Sortierung betrachtet. Eine Liste $[x_1,\ldots,x_n]$ über *int* heißt **absteigend sortiert**, wenn sie ihre Elemente in absteigender Ordnung enthält: $x_1 \ge \cdots \ge x_n$. Wenn wir im Folgenden den Begriff der Sortierung nicht weiter qualifizieren, meinen wir stets aufsteigende Sortierung.

Aufgabe 5.1 Schreiben Sie eine Prozedur *isort'*, die eine Liste in absteigender Ordnung sortiert.

Aufgabe 5.2 Schreiben Sie eine Prozedur *sorted* : *int list* → *bool*, die testet, ob eine Liste aufsteigend sortiert ist. Verwenden Sie dabei keine Hilfsprozedur.

Aufgabe 5.3 Schreiben Sie eine Prozedur *perm* : *int list* → *int list* → *bool*, die testet, ob zwei Listen bis auf die Anordnung ihrer Elemente gleich sind. Verwenden Sie dabei die Prozedur *isort* und die Tatsache, dass *int list* ein Typ mit Gleichheit ist.

Aufgabe 5.4 Schreiben Sie eine Prozedur *issort* : *int list* → *int list*, die eine Liste sortiert und dabei Mehrfachauftreten von Elementen eliminiert. Beispielsweise soll für $[3,1,3,1,0]$ die Liste $[0,1,3]$ geliefert werden.

Aufgabe 5.5 Beim Einfügen eines Elements in eine sortierte Liste kann es sein, dass *insert* das neue Element gleich an den Anfang der sortierten Liste setzen kann. Können Sie charakterisieren, unter welchen Umständen das der Fall ist? Wie muss eine Liste sortiert sein, damit jeder Einfügeschritt von *isort* diese Eigenschaft hat?

```
fun pisort compare =
    let
        fun insert (x, nil)   = [x]
          | insert (x, y::yr) = case compare(x,y) of
                GREATER => y::insert(x,yr)
              |    _    => x::y::yr
    in
        foldl insert nil
    end
```

*val pisort : (α * α → order) → α list → α list*

Abbildung 5.2: *Polymorphes Sortieren durch Einfügen*

Umgekehrt gibt es den Fall, dass *insert* das neue Element erst am Ende der Liste einfügen kann und daher die Liste per Rekursion vollständig durchlaufen muss. Können Sie charakterisieren, unter welchen Umständen das der Fall ist? Wie muss eine Liste sortiert sein, damit jeder Einfügeschritt von *isort* diese Eigenschaft hat?
Erproben Sie *isort* mit den Listen

```
val xs = List.tabulate(5000, fn x => x)
val ys = rev xs
```

Können Sie den beträchtlichen Laufzeitunterschied beim Sortieren der beiden Listen erklären?

5.2 Polymorphes Sortieren

Wir wollen jetzt eine polymorphe Sortierprozedur schreiben, die Listen über einem beliebigen Typ gemäß einer beliebigen Ordnung sortieren kann. Ordnungen werden in Standard ML durch sogenannte **Vergleichsprozeduren** dargestellt. Eine Vergleichsprozedur liefert zu zwei Werten einen der drei Werte *LESS*, *EQUAL* und *GREATER* des Typs *order*. Die Vergleichsprozedur für die Standardordnung der ganzen Zahlen ist über $Int.compare : int * int \rightarrow order$ verfügbar:

$$\begin{aligned} Int.compare(2,6) &= LESS \\ Int.compare(6,2) &= GREATER \\ Int.compare(6,6) &= EQUAL \end{aligned}$$

Ausgehend von *isort* ist es nicht schwer, eine polymorphe Sortierprozedur *pisort*, so wie in Abbildung 5.2 gezeigt, zu deklarieren. Die Hilfsprozedur *insert* haben wir lokal im Rumpf von *pisort* deklariert, da sie die als Argument übergebene Ordnung benötigt. Hier sind Beispiele für die Anwendung von *pisort*:

```
pisort Int.compare
```
fn : int list → int list

```
pisort Int.compare [5, 2, 2, 13, 4, 9, 9, 13, ~2]
```
[~2, 2, 2, 4, 5, 9, 9, 13, 13] : int list

```
pisort Real.compare [5.0, 2.0, 2.0, 13.0, 4.0, 9.0]
```
[2.0, 2.0, 4.0, 5.0, 9.0, 13.0] : real list

Einige Interpreter stellen eine polymorphe Sortierprozedur unter *List.sort* bereit.

5.3 Inverse und lexikalische Ordnungen

Wir deklarieren jetzt eine Prozedur

$$invert\colon\ (\alpha * \alpha \to order) \to \alpha * \alpha \to order$$

die zu einer Ordnung die **inverse Ordnung** liefert:

```
fun invert (compare : 'a * 'a -> order) (x,y) = compare (y,x)
```

Durch Invertieren der Ordnung können wir mit *pisort* statt aufsteigend auch absteigend sortieren:

```
pisort (invert Int.compare) [5, 2, 2, 13, 4, 9, 9, 13, ~2]
```
[13, 13, 9, 9, 5, 4, 2, 2, ~2] : int list

In einem Lexikon sind die Schlagwörter nach einer Ordnung sortiert, die aus der Ordnung für die Zeichen des Wortes abgeleitet ist. Weiter links stehende Zeichen haben dabei mehr Gewicht als weiter rechts stehende. Das dieser Ordnung zugrunde liegende Prinzip können wir auf Listen übertragen. Abbildung 5.3 zeigt eine Prozedur

$$lex\colon\ (\alpha * \alpha \to order) \to \alpha\ list * \alpha\ list \to order$$

die zu einer Ordnung für α die **lexikalische Ordnung** für α *list* liefert:

```
pisort (lex Int.compare) [[4,1], [], [4], [4,1,~8]]
```
[[], [4], [4, 1], [4, 1, ~8]] : int list list

Die charakteristische Eigenschaft einer lexikalischen Ordnung besteht darin, dass die Positionen zweier anzuordnender Listen solange paarweise verglichen werden, bis die erste Position erreicht ist, in der keine Gleichheit vorliegt. Diese Position entscheidet dann über die Anordnung der Listen. Wenn eine Liste in allen Positionen mit einer längeren Liste übereinstimmt, kommt die kürzere Liste vor der längeren (z.B. $[1,2]$ vor $[1,2,3]$).

Eine zu *lex* äquivalente Prozedur ist unter *List.collate* vordeklariert.

```
fun lex (compare : 'a * 'a -> order) p = case p of
    (nil, _::_) => LESS
  | (nil, nil)  => EQUAL
  | (_::_, nil) => GREATER
  | (x::xr, y::yr) => case compare(x,y) of
        EQUAL => lex compare (xr,yr)
      |   s   => s
```

*lex : (α * α → order) → α list * α list → order*

Abbildung 5.3: *Lexikalische Ordnung für Listen*

Aufgabe 5.6 Welche Typschemen werden für die Prozeduren *invert* und *lex* abgeleitet, wenn man in den obigen Deklarationen die Typangaben weglässt?

Aufgabe 5.7 Deklarieren Sie mithilfe von *List.collate* eine Prozedur

stringCompare : *string* ∗ *string* → *order*

die dasselbe Ergebnis liefert wie *String.compare* (siehe § 4.7.2).

Aufgabe 5.8 Deklarieren Sie eine Prozedur

intPairCompare : (*int* ∗ *int*) ∗ (*int* ∗ *int*) → *order*

die die lexikalische Ordnung für Paare des Typs *int* ∗ *int* darstellt. Zum Beispiel soll [(3,4), (3,5), (4,0)] gemäß dieser Ordnung sortiert sein.

Aufgabe 5.9 Deklarieren Sie eine Prozedur

lex: (α ∗ α → *order*) → (β ∗ β → *order*) → (α ∗ β) ∗ (α ∗ β) → *order*

die zu zwei Ordnungen für die Typen α und β die lexikalische Ordnung für den Produkttyp α ∗ β liefert. Deklarieren Sie mit Ihrer Prozedur *lex* eine Prozedur, die die lexikalische Ordnung für Paare des Typs *int* ∗ *real* darstellt, die in der ersten Position absteigend und in der zweiten Position aufsteigend sortiert. Zum Beispiel soll [(3,4.0), (2,2.0), (2,3.0)] gemäß dieser Ordnung sortiert sein.

5.4 Sortieren durch Mischen

Sortieren durch Einfügen hat den Nachteil, dass es für manche Listen unangemessen langsam ist (siehe Übungsaufgabe 5.5 auf S. 100). Wir entwickeln jetzt eine zweite Sortierprozedur, die alle Listen mit angemessener Laufzeit sortiert.

Sortieren durch Mischen beruht auf einer als Mischen bezeichneten Operation *merge*, die zwei sortierte Listen in eine sortierte Liste kombiniert:

merge([2,3,5,8], [1,2,4,6,7]) = [1,2,2,3,4,5,6,7,8]

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys))
                     (nil, nil) xs
```
*val split : α list → α list * α list*

```
fun merge (nil  , ys   ) = ys
  | merge (xs   , nil  ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)
```
*val merge : int list * int list → int list*

```
fun msort []  = []
  | msort [x] = [x]
  | msort xs  = let val (ys,zs) = split xs
                in  merge(msort ys, msort zs) end
```
val msort : int list → int list

Abbildung 5.4: *Sortieren durch Mischen*

Die Essenz von Sortieren durch Mischen wird durch die Gleichung

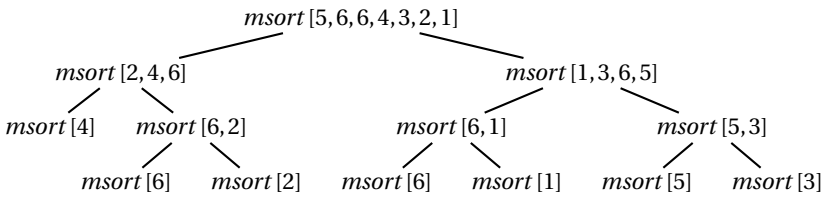
$$sort\,(xs\,@\,ys) \;=\; merge(sort\,xs,\; sort\,ys)$$

formuliert. Entsprechend dieser Gleichung können wir eine Liste wie folgt sortieren:

1. Zerlege die Liste in zwei etwa gleich große Teillisten.
2. Sortiere die beiden Teillisten durch Rekursion.
3. Kombiniere die sortierten Teillisten mit *merge* in eine sortierte Liste.

Abbildung 5.4 formuliert eine Prozedur *msort*, die Listen durch Mischen sortiert. Die Prozedur *split* ist für das Zerlegen einer Liste in zwei Teillisten zuständig. Sie beginnt mit zwei leeren Listen und fügt die Elemente der zu zerlegenden Liste alternierend in diese ein. Die Alternierung ergibt sich dadurch, dass die Teillisten bei jeder Einfügung vertauscht werden.

Die der Prozedur *msort* zugrunde liegende Rekursion unterscheidet sich von den bisher gesehenen Rekursionen dadurch, dass im Rekursionsfall statt einem zwei rekursive Aufrufe zu erledigen sind: *merge*(*msort ys, msort zs*). Man spricht von **binärer** (zwei rekursive Aufrufe) und **linearer Rekursion** (ein rekursiver Aufruf). Bei binärer Rekursion ist die Hierarchie der für die Ausführung eines Aufrufs insgesamt erforderlichen Aufrufe nicht mehr linear, sondern baumartig. Die Darstellung dieser Hierarchie wird als **Rekursionsbaum** bezeichnet. Hier ist der Rekursionsbaum für den Aufruf *msort* [5,6,6,4,3,2,1]:



Die Rekursion von *msort* terminiert, da die von *split* gelieferten Teillisten kürzer als die Ausgangslisten sind. Für die gute Laufzeit von *msort* ist es wesentlich, dass die von *split* gelieferten Teillisten jeweils nur halb so groß wie die Vorgängerlisten sind.

Nichtlineare Rekursion wird auch als **Baumrekursion** bezeichnet.

Aufgabe 5.10 Schreiben Sie eine polymorphe Sortierprozedur, die durch Mischen sortiert.

Aufgabe 5.11 Vergleichen Sie die Laufzeiten der Prozeduren *isort* und *msort* für die Listen

```
val xs = List.tabulate(5000, fn x => x)
val ys = rev xs
```

Aufgabe 5.12 (Partition) Deklarieren Sie eine Prozedur

$$partition : int \to int\ list \to int\ list * int\ list$$

die zu einer Zahl x und einer Liste xs zwei Listen us und vs wie folgt liefert:

1. $sort\,(us @ vs) = sort\ xs$.
2. Alle Elemente von us sind kleiner als x und alle Elemente von vs sind größer oder gleich x.

Schreiben Sie *partition* mit *foldl*. Orientieren Sie sich dabei an der Prozedur *split*.

Aufgabe 5.13 (Quicksort) Quicksort ist ein klassischer Sortieralgorithmus, der die zu sortierende Liste gemäß der Prozedur *partition* aus Aufgabe 5.12 in zwei Teillisten zerlegt und ihre durch Rekursion berechneten Sortierungen mit Konkatenation wieder zusammenfügt. Die Essenz von Quicksort wird durch die bedingte Gleichung

$$sort\,(x :: xr) = (sort\ us) @ [x] @ (sort\ vs) \quad \text{für} \quad (us, vs) = partition\ x\ xr$$

beschrieben. Deklarieren Sie eine Prozedur *qsort*, die Listen über *int* gemäß Quicksort sortiert.

Aufgabe 5.14 (Dreifach-Partition) Schreiben Sie eine Prozedur

$$partition\colon\ (\alpha * \alpha \to order) \to \alpha\ list \to \alpha\ list * \alpha\ list * \alpha\ list$$

die eine Liste xs gemäß einer Prozedur f in drei Listen us, vs, ws zerlegt, sodass f für die Elemente von us *LESS*, für die Elemente von vs *EQUAL* und für die Elemente von ws *GREATER* liefert.

5.5 Endliche Mengen und strikte Sortierung

Unter einer **Menge** versteht man im mathematischen Sprachgebrauch eine Zusammenfassung von mathematischen Objekten. Die zu einer Menge zusammengefassten Objekte werden als die **Elemente** der Menge bezeichnet. Die Notation $x \in X$ besagt, dass das

Objekt x ein Element der Menge X ist. Man sagt, dass eine Menge ihre Elemente **enthält** oder auch dass eine Menge aus ihren Elementen **besteht**.

Zu endlich vielen Objekten $x_1, \ldots, x_n$ existiert stets genau eine Menge, die genau diese Objekte als Elemente hat. Diese Menge wird mit $\{x_1, \ldots, x_n\}$ bezeichnet. Insbesondere existiert genau eine **leere Menge** $\emptyset$, die keine Elemente hat. Beachten Sie, dass $\{1,2\} = \{2,1\} = \{1,2,1\}$ gilt, da Mengen im Gegensatz zu Tupeln und Listen keine Ordnung für ihre Elemente beinhalten und Elemente auch nicht mehrfach auftreten können. Wir müssen also zwischen der Notation $\{1,2,1\}$ und der durch sie beschriebenen Menge unterscheiden.

Generell gilt, dass eine Menge vollständig durch ihre Elemente bestimmt ist. Diese grundlegende Eigenschaft von Mengen kann durch das **Gleichheitsaxiom** formuliert werden: Zwei Mengen X und Y sind genau dann gleich ($X = Y$), wenn jedes Element von X ein Element von Y ist und jedes Element von Y ein Element von X ist. Aus dem Gleichheitsaxiom folgt, dass es zu gegebenen Objekten immer nur höchstens eine Menge gibt, die genau diese Objekte als Elemente enthält.

Seien X und Y Mengen. Wir definieren einige gebräuchliche Begriffe und Notationen:

- X ist eine **Teilmenge** von Y, in Zeichen $X \subseteq Y$, wenn jedes Element von X ein Element von Y ist. Gemäß dem Gleichheitsaxiom gilt $X = Y$ genau dann, wenn $X \subseteq Y$ und $Y \subseteq X$ gilt.
- Der **Schnitt** $X \cap Y$ ist die Menge, die genau aus den Objekten besteht, die sowohl Element von X als auch Element von Y sind. Beispielsweise gilt $\{1,2,3\} \cap \{1,3,4\} = \{1,3\}$.
- Die **Vereinigung** $X \cup Y$ ist die Menge, die genau aus den Objekten besteht, die Element mindestens einer der Mengen X und Y sind. Beispielweise gilt $\{1,2,3\} \cup \{1,3,4\} = \{1,2,3,4\}$.
- Die **Differenz** $X - Y$ ist die Menge, die genau aus den Elementen von X besteht, die keine Elemente von Y sind. Beispielweise gilt $\{1,2,3\} - \{1,3,4\} = \{2\}$.

Es gibt viele Möglichkeiten, Mengen durch die Datenstrukturen einer Programmiersprache darzustellen. Dabei gibt es keine beste Darstellung, die für alle Anwendungen gleichermaßen gut ist, sondern verschiedene Darstellungen sind für verschiedene Anwendungen gut. Aus diesem Grund sind Mengen in Programmiersprachen nicht direkt als Datenstruktur verfügbar, sondern müssen je nach Anwendung mit den vorhandenen Datenstrukturen realisiert werden.

Was die Darstellung von Mengen anbetrifft, ist unser programmiersprachliches Repertoire noch sehr begrenzt. Immerhin sind wir in der Lage, endliche Mengen durch Listen darzustellen. Diese Darstellung beschreiben wir durch die Gleichung

$$\mathrm{Set}\,[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$$

die jeder Liste xs eine endliche Menge $\mathrm{Set}\,xs$ zuordnet. Bei $\mathrm{Set}\,xs$ handelt es sich gerade um die Menge, die aus den Elementen der Liste xs besteht. Beispielsweise gilt

Set [2,1,3,1] = {1,2,3}. Die Darstellung von Mengen durch Listen ist nicht eindeutig, da Mehrfachauftreten bei der Darstellung von nichtleeren Mengen erlaubt sind. Beispielsweise gilt {1} = Set [1] = Set [1,1] = Set [1,1,1]. Dieses Problem können wir vermeiden, wenn wir für die Darstellung von Mengen nur **strikt sortierte** Listen, das heißt ohne Mehrfachauftreten, zulassen. Beispielsweise hat die Menge {2,1} dann nur noch die Darstellung [1,2].

Aufgabe 5.15 (Striktes Sortieren durch Mischen)

a) Schreiben Sie eine polymorphe Prozedur *smerge*, die zwei strikt sortierte Listen zu einer strikt sortierten Liste kombiniert. Beispielsweise soll gelten: *smerge Int.compare* ([1,3], [2,3]) = [1,2,3].

b) Schreiben Sie eine polymorphe Prozedur *ssort*, die Listen strikt sortiert. Beispielsweise soll *ssort Int.compare* [5,3,2,5] = [2,3,5] gelten.

c) Machen Sie sich klar, dass es sich bei *ssort Int.compare* um eine Prozedur handelt, die zu einer Liste *xs* die eindeutig bestimmte strikt sortierte Liste liefert, die dieselbe Menge wie *xs* darstellt.

Aufgabe 5.16 Seien endliche Mengen ganzer Zahlen gemäß *Int.compare* durch strikt sortierte Listen dargestellt. Deklarieren Sie Prozeduren wie folgt:

a) *member* : *int* → *int list* → *bool* testet, ob eine Zahl Element einer Menge ist.

b) *union* : *int list* → *int list* → *int list* liefert die Vereinigung zweier Mengen.

c) *intersection* : *int list* → *int list* → *int list* liefert den Schnitt zweier Mengen.

d) *difference* : *int list* → *int list* → *int list* liefert die Differenz zweier Mengen.

e) *eqset* : *int list* → *int list* → *bool* testet, ob zwei Mengen gleich sind.

f) *subset* : *int list* → *int list* → *bool* testet für zwei Mengen X und Y, ob $X \subseteq Y$. Realisieren Sie *subset* mit *member* und alternativ mit *eqset* und *intersection* gemäß der Äquivalenz $X \subseteq Y \iff X = X \cap Y$.

Aufgabe 5.17 Seien endliche Mengen durch unsortierte Listen dargestellt. Deklarieren Sie Prozeduren wie folgt:

a) *member* : $''a \to ''a$ *list* → *bool* testet, ob ein Wert Element einer durch eine Liste dargestellten Menge ist.

b) *subset* : $''a$ *list* → $''a$ *list* → *bool* testet für zwei Listen *xs* und *ys*, ob die durch *xs* dargestellte Menge eine Teilmenge der durch *ys* dargestellten Menge ist.

c) *eqset* : $''a$ *list* → $''a$ *list* → *bool* testet, ob zwei Listen dieselbe Menge darstellen.

Aufgabe 5.18 (Potenzmenge) Seien endliche Mengen durch Listen ohne Mehrfachauftreten dargestellt. Deklarieren Sie eine Prozedur *power* : α *list* → α *list list*, die zu einer Menge X die Menge liefert, die genau die Teilmengen von X enthält. Dabei ist die Reihenfolge beliebig, aber es soll keine Mehrfachauftreten geben. Beispielsweise wäre es korrekt, wenn *power* für [1,2] die Liste [[1,2],[2],[1],[]] liefert.

5.6 Primzerlegung

Jede natürliche Zahl größer 2 lässt sich als Produkt von Primzahlen darstellen:

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$147 = 3 \cdot 7 \cdot 7$$

Diese als **Primzerlegung** bezeichnete Darstellung wird eindeutig, wenn man die Faktoren der Größe nach anordnet. Wir stellen Primzerlegungen durch aufsteigend sortierte Listen dar und bezeichnen die Primzerlegung einer Zahl $n \geq 2$ mit $pf\ n$:

$$pf\ 60 = [2,2,3,5]$$
$$pf\ 147 = [3,7,7]$$

Für die Bestimmung von Primzerlegungen gibt es einen eleganten Algorithmus, der die zu zerlegende Zahl schrittweise durch größer werdende Primzahlen teilt. Wir realisieren den Algorithmus durch eine rekursive Prozedur pf' mit zwei Argumenten. Für $n = 50$ soll sich das folgende Ausführungsprotokoll ergeben:

$$\begin{aligned} pf\ 50 &= pf'(2,50) \\ &= 2 :: pf'(2,25) \\ &= 2 :: pf'(3,25) \\ &= 2 :: pf'(4,25) \\ &= 2 :: pf'(5,25) \\ &= 2 :: 5 :: pf'(5,5) \\ &= [2,5,5] \end{aligned}$$

Das erste Argument von pf' dient als Akku (§ 1.10) und realisiert einen Zähler, der mit 2 beginnend schrittweise um 1 erhöht wird. Der Rest des Algorithmus lässt sich am einfachsten dadurch erklären, dass wir die Funktion definieren, die die Prozedur pf' berechnet. Der Einfachheit halber wollen wir diese Funktion auch mit pf' bezeichnen. Gemäß dem obigen Ausführungssprotokoll gibt es für die Definition der Funktion pf' eigentlich nur die Möglichkeit

$$pf'(k,n) = pf\ n \qquad \text{für } A(k,n)$$

wobei $A(k,n)$ eine noch zu bestimmende **Anwendungsbedingung** darstellt, die die Argumente beschreibt, für die die Funktion pf' definiert sein soll. Die Anwendungsbedingung muss die folgenden Eigenschaften haben:

1. Für $n \geq 2$ gilt $A(2,n)$.
2. Für die Funktion pf' gibt es Rekursionsgleichungen.

Die erste Bedingung sorgt dafür, dass wir die Primzerlegung für n als $pf'(2,n)$ berechnen können. Die zweite Bedingung sorgt dafür, dass wir eine Prozedur schreiben können, die die Funktion pf' berechnet. Um das Aufstellen der Rekursionsgleichungen zu

erleichtern, ist es naheliegend, die Anwendungsbedingung möglichst restriktiv zu formulieren. Allerdings müssen wir dabei mindestens die von der ersten Bedingung geforderten Argumente zulassen. Diese Überlegungen führen zusammen mit dem obigen Ausführungsprotokoll zu der folgenden Definition von $A(k,n)$:

> k und n sind natürliche Zahlen mit $2 \leq k \leq n$
> und n ist durch keine Zahl u mit $2 \leq u < k$ teilbar

Gemäß dieser Definition gelten für alle Zahlen k und n, für die $A(k,n)$ gilt, die folgenden Eigenschaften:

1. Wenn $k^2 > n$, dann ist n eine Primzahl.
2. Wenn n durch k teilbar ist, dann gilt $A(k, n/k)$ und k ist die kleinste Primzahl, die n restfrei teilt.
3. Wenn n nicht durch k teilbar ist und $k < n$, dann gilt $A(k+1, n)$.

Aus diesen Eigenschaften ergeben sich die folgenden Rekursionsgleichungen:

$$
\begin{aligned}
pf'(k,n) &= [n] && \text{für } k^2 > n \\
pf'(k,n) &= k :: pf'(k, n/k) && \text{für } n \; mod \; k = 0 \text{ und } k^2 \leq n \\
pf'(k,n) &= pf'(k+1, n) && \text{für } n \; mod \; k \neq 0 \text{ und } k^2 \leq n
\end{aligned}
$$

Die Rekursionsgleichungen sind terminierend, da im Rekursionsfall die Differenz $n-k$ kleiner wird und die Anwendungsbedingung $n-k \geq 0$ garantiert. Die Umsetzung der Gleichungen in eine Prozedur ist Routine:

```
fun pf'(k,n) = if k*k>n then [n] else
               if n mod k = 0 then k::pf'(k, n div k)
               else pf'(k+1,n)
fun pf n = pf'(2,n)
val pf : int → int list

pf 1989
[3, 3, 13, 17] : int list
```

Die Anwendungsbedingung $A(k,n)$ hat die wichtige Eigenschaft, dass sich ihre Gültigkeit bei einem Aufruf der Prozedur pf' auf die rekursiven Aufrufe von pf' fortpflanzt (folgt aus den Eigenschaften (2) und (3)). Bedingungen mit dieser Eigenschaft bezeichnen wir als **Invarianten**. Allgemein versteht man unter einer Invariante eine Eigenschaft, deren Gültigkeit sich im Verlauf einer Programmausführung nicht ändert.

Das Beispiel der Primzerlegung zeigt uns, dass die Entwicklung von Algorithmen eine interessante Angelegenheit ist, bei der mathematische Argumente eine wichtige Rolle spielen.

Aufgabe 5.19 (Natürliche Quadratwurzel) Entwickeln Sie einen Algorithmus, der $\lfloor\sqrt{n}\rfloor$ für natürliche Zahlen n mit einer endrekursiven Hilfsprozedur *sqrt* bestimmt. Für $n = 6$ soll sich das folgende Ausführungssprotokoll ergeben:

$$\left\lfloor\sqrt{6}\right\rfloor = sqrt(1,6) = sqrt(2,6) = sqrt(3,6) = 2$$

Definieren Sie die Funktion, die die Prozedur *sqrt* berechnet, mit einer Anwendungsbedingung $A(k,n)$. Geben Sie für diese Funktion Rekursionsgleichungen an.

Aufgabe 5.20 (Liste der ersten n Primzahlen) Deklarieren Sie eine Prozedur *primelist* : $int \to int\ list$, die zu $n \geq 0$ die Liste der ersten n Primzahlen in aufsteigender Ordnung liefert. Verwenden Sie die Prozedur *nextprime* aus § 3.11. Hinweis: Verwenden Sie ein Akku und berechnen Sie die Liste der ersten n Primzahlen ab einer gegebenen Primzahl.

Aufgabe 5.21 (Iterlist) Deklarieren Sie eine höherstufige Prozedur wie folgt:

$$iterlist\colon\ (\alpha \to \alpha) \to int \to \alpha \to \alpha\ list$$

$$iterlist\ f\ n\ x = \underbrace{[x,\ f\,x,\ f(f\,x),\ \ldots\ ]}_{\text{Länge } n}$$

Überzeugen Sie sich davon, dass die Prozedur

```
fun primelist n = iterlist nextprime n 2
```

die Liste der ersten n Primzahlen liefert.

5.7 Ein überraschender Laufzeitunterschied

Die Ausführung eines Prozeduraufrufs benötigt eine gewisse Zeit. Man spricht von der **Laufzeit** eines Prozeduraufrufs. Wenn eine Prozedur für bestimmte Argumente eine zu hohe Laufzeit hat (zum Beispiel 100 Jahre), ist sie für diese Argumente unbrauchbar. In der Praxis müssen wir Prozeduren also so schreiben, dass ihre Laufzeit für die relevanten Argumente akzeptabel ist.

Wir haben zwei unterschiedliche Prozeduren zum Reversieren von Listen kennengelernt:

```
fun rev nil     = nil
  | rev (x::xr) = rev xr @ [x]
```

```
fun rev' xs = foldl op:: nil xs
```

Welche der beiden Prozeduren rechnet schneller? Anfänger neigen dazu, die Prozedur *rev* für schneller zu halten. Das Gegenteil ist jedoch der Fall: *rev'* ist sehr viel schneller als *rev*. Wenn wir beide Prozeduren auf eine Liste mit 10000 Elementen anwenden, können wir einen deutlichen Laufzeitunterschied beobachten. Der Unterschied wird größer, wenn wir die Prozeduren auf längere Listen anwenden. Für eine Liste der Länge 100000

liefert *rev* auch nach einigen Minuten noch kein Ergebnis, wohingegen die Prozedur *rev'* das Ergebnis sofort liefert.

Der Grund für die hohe Laufzeit von *rev* liegt in den Kosten für die durch den Operator @ bezeichnete Konkatenationsprozedur. Diese muss mit einer rekursiven Prozedur realisiert werden, die der Prozedur *append* aus § 4.2 entspricht. Die Kosten für eine Konkatenation *xs@ys* wachsen also mit der Länge von *xs*. Das bedeutet, dass die Kosten für die Konkatenationen die restlichen Kosten von *rev* bei weitem überschreiten. Wir werden diese Situation in Kapitel 11 genauer analysieren.

Bei der vordeklarierten Prozedur *rev* handelt es sich selbstverständlich um eine schnelle Prozedur zum Reversieren von Listen.

Aufgabe 5.22 Überzeugen Sie sich mit einem Interpreter von dem Laufzeitunterschied zwischen den oben deklarierten Prozeduren *rev* und *rev'*. Beginnen Sie mit der Liste *val xs = List.tabulate(10000, fn x ⇒ x)*.

Bemerkungen

Sortieralgorithmen gehören zum Grundrepertoire der Informatik. Wir haben Sortieren durch Einfügen, Sortieren durch Mischen und Quicksort betrachtet. Außerdem haben wir gezeigt, wie man polymorphe Sortierprozeduren schreibt, die Objekte eines beliebigen Typs gemäß einer beliebigen Ordnung sortieren können. Dabei haben wir gelernt, wie man Ordnungen durch Prozeduren darstellt, und wie man inverse und lexikalische Ordnungen konstruiert. Mit Sortieren durch Mischen haben wir erstmals einen baumrekursiven Algorithmus kennengelernt.

Wir haben uns auch mit der Darstellung von Mengen durch strikt sortierte Listen beschäftigt. Außerdem haben wir einen Algorithmus für die Primzerlegung natürlicher Zahlen entwickelt. Die Korrektheit des Algorithmus haben wir mit einer Argumentation gezeigt, bei der eine auch als Invariante bezeichnete Anwendungsbedingung von entscheidender Bedeutung war.

Schließlich haben wir noch das Thema **Effizienz** angesprochen. Ein Programm oder eine Prozedur wird als effizient bezeichnet, wenn es seine Aufgabe in Bezug auf Laufzeit und **Speicherbedarf** vergleichsweise sparsam erfüllt. Für das Reversieren von Listen haben wir eine schnelle und eine langsame Prozedur betrachtet, für das Sortieren von Listen haben wir eine Prozedur betrachtet, die Listen je nach Anordnung schnell oder langsam sortiert. Außerdem haben wir eine Prozedur kennengelernt, die Listen unabhängig von ihrer Anordnung schnell sortiert.

Vorerst werden wir das Thema Effizienz noch zurückstellen. In Kapitel 11 werden wir dann aber eine Theorie einführen, mit der fundierte Aussagen über die Laufzeit von Prozeduren gemacht werden können. Und in Kapitel 16 werden wir auf den Speicherbedarf von Programmen eingehen.

Verzeichnis

Sortieren und Ordnungen: sortierte Liste, Sortierung einer Liste; Sortieren durch Einfügen, Sortieren durch Mischen, Quicksort; polymorphe Sortierprozeduren; Vergleichsprozeduren; inverse und lexikalische Ordnungen.

Vordeklarierte Prozeduren: Int.compare, Real.compare; List.sort, List.collate.

Strikte Sortierung und endliche Mengen: Gleichheitsaxiom; $x \in X$; Schnitt $X \cap Y$, Vereinigung $X \cup Y$, Differenz $X - Y$; Teilmengen $X \subseteq Y$.

Binäre Rekursion und Rekursionsbäume: lineare Rekursion, Baumrekursion.

Effizienz, Laufzeit, Speicherbedarf.

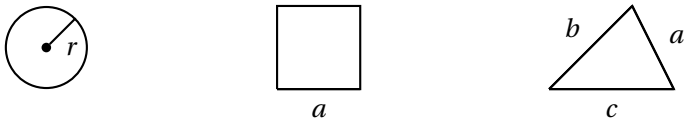
Primzerlegung und Invarianten.

6 Konstruktoren und Ausnahmen

In diesem Kapitel erweitern wir unser programmiersprachliches Repertoire um die Möglichkeit, neue Typen zu deklarieren, deren Werte mithilfe von sogenannten Konstruktoren dargestellt werden. Damit eröffnen sich viele neue Anwendungen. Zum Beispiel können wir arithmetische Ausdrücke als Werte darstellen und sie mit einer Prozedur symbolisch differenzieren. Außerdem behandeln wir in diesem Kapitel das Programmieren mit Ausnahmen.

6.1 Konstruktoren

Als einleitendes Beispiel betrachten wir die Darstellung geometrischer Objekte, die gemäß dreier Formen gebildet sind:



Kreis	Quadrat	Dreieck
$(1, r)$	$(2, a)$	$(3, (a, b, c))$

Wir stellen die Objekte durch Paare dar, deren erste Komponente mit einer der Zahlen 1, 2, 3 anzeigt, um welche Form es sich handelt:

- Ein Paar $(1, r)$ stellt einen Kreis mit dem Radius r dar.
- Ein Paar $(2, a)$ stellt ein Quadrat mit der Seitenlänge a dar.
- Ein Paar $(3, (a, b, c))$ stellt ein Dreieck mit den Seitenlängen a, b, c dar.

Die erste Komponente dieser Paare bezeichnen wir als **Variantennummer** und die zweite als **Datum**. In Standard ML können wir diese Darstellung geometrischer Objekte mit der folgenden **Typdeklaration** formulieren:

```
datatype shape =
    Circle   of real
  | Square   of real
  | Triangle of real * real * real
```

Die Deklaration führt einen neuen Typ *shape* ein, dessen Werte mithilfe der **Konstruktoren** *Circle*, *Square* und *Triangle* beschrieben werden können:

```
Circle 4.0
```
Circle 4.0 : shape

```
Square 3.0
```
Square 3.0 : shape

```
Triangle (4.0, 3.0, 5.0)
```
Triangle (4.0, 3.0, 5.0) : shape

Die Konstruktoren können wie Prozeduren benutzt werden und haben die folgenden Typen:

$$\begin{array}{ll} \mathit{Circle} & : \mathit{real} \to \mathit{shape} \\ \mathit{Square} & : \mathit{real} \to \mathit{shape} \\ \mathit{Triangle} & : \mathit{real} * \mathit{real} * \mathit{real} \to \mathit{shape} \end{array}$$

Typen, die so wie *shape* mit dem Schlüsselwort *datatype* deklariert sind, bezeichnen wir als **Konstruktortypen**. Wenn ein Konstruktortyp n Konstruktoren hat, sagen wir, dass er n **Varianten** hat. Zur i-ten Variante eines Konstruktortyps gehören alle Werte, die mit dem i-ten Konstruktor beschrieben werden können.

Semantisch gesehen sind die Werte von Konstruktortypen wie oben beschrieben Paare, die eine Variantennummer mit einem Datum kombinieren. Sie werden aber grundsätzlich mithilfe der entsprechenden Konstruktoren dargestellt. Diese Konvention machen wir uns auch bei der grafischen Darstellung dieser Werte zu Eigen:

Ein Konstruktortyp ist ein Typ mit Gleichheit, wenn die Argumenttypen aller Konstruktoren Typen mit Gleichheit sind. Also ist *shape* ein Typ ohne Gleichheit, da *real* ein Typ ohne Gleichheit ist.

Konstruktoren und Muster

Konstruktoren können in den Mustern von Regeln verwendet werden, um die Anwendbarkeit der Regel auf die dem Konstruktor entsprechende Variante zu beschränken. Wir führen das am Beispiel einer Prozedur vor, die den Flächeninhalt eines geometrischen Objekts liefert:

```
fun area (Circle r)        = Math.pi*r*r
  | area (Square a)        = a*a
  | area (Triangle(a,b,c)) = let val s = (a+b+c)/2.0
                             in Math.sqrt(s*(s-a)*(s-b)*(s-c))
                             end
```
val area : shape → real

```
area (Square 3.0)
```
9.0 : real

```
area (Triangle(6.0, 6.0, Math.sqrt 72.0))
```
18.0 : real

Für jede der drei Varianten von *shape* hat *area* eine eigene Regel. Die Muster der Regeln sind wie Konstruktoranwendungen gebildet. Wenn *area* beispielsweise auf einen mit *Square* konstruierten Wert angewendet wird, kommt die zweite Regel zur Anwendung. Dabei wird die Variable *a* an das Datum des Wertes gebunden (also die Kantenlänge des Quadrats).

Aufgabe 6.1 Deklarieren Sie eine Prozedur $variant : shape \to int$, die die Variantennummer eines geometrischen Objekts liefert. Beispielsweise soll $variant(Square\ 3.0) = 2$ gelten.

Aufgabe 6.2 Deklarieren Sie eine Prozedur $scale : real \to shape \to shape$, die ein Objekt gemäß einem Faktor skaliert (d.h. vergrößert oder verkleinert). Beispielsweise soll $scale\ 0.5\ (Square\ 3.0) = Square\ 1.5$ gelten.

Groß- und Kleinschreibung von Bezeichnern

Aus der Sicht der lexikalischen Syntax handelt es sich bei *shape*, *Circle*, *Square* und *Triangle* um Bezeichner. Im Rahmen der statischen Semantik wird jedem Bezeichnerauftreten eine **Lesart** zugewiesen, die besagt, ob das Auftreten einen Typen, einen Konstruktor oder einen Wert bezeichnet.

Um die Lesbarkeit von Programmen zu verbessern, wählen wir für Konstruktoren grundsätzlich Bezeichner, die mit einem Großbuchstaben beginnen, und für Typen und Werte Bezeichner, die mit einem Kleinbuchstaben beginnen.

6.2 Enumerationstypen

Die Deklaration

```
datatype day = Monday | Tuesday | Wednesday
             | Thursday | Friday | Saturday | Sunday
```

führt einen Typ *day* ein, dessen Werte die verschiedenen Wochentage darstellen. Die Werte des Typs werden durch **nullstellige Konstruktoren** beschrieben, die wie Konstanten verwendet werden können:

```
fun weekend Saturday = true
  | weekend Sunday   = true
  | weekend _        = false
```
val weekend : day → bool

```
weekend Saturday
```
true : bool

```
map weekend [Monday, Wednesday, Friday, Saturday, Sunday]
```
[false, false, false, true, true] : bool list

Nullstellige Konstruktoren beschreiben Werte, die nur aus einer Variantennummer bestehen. Bei den Werten des Typs *day* handelt es sich also um die Zahlen 1 bis 7. Konstruktortypen, die so wie *day* nur mit nullstelligen Konstruktoren gebildet sind, werden als **Enumerationstypen** bezeichnet.

Wir haben bereits zwei vordeklarierte Enumerationstypen kennengelernt:

```
datatype bool  = false | true

datatype order = LESS | EQUAL | GREATER
```

Wir unterscheiden zwischen nullstelligen Konstruktoren (wie beim Typ *day*) und **einstelligen Konstruktoren** (wie beim Typ *shape*). Es gibt keine mehrstelligen Konstruktoren.

Aufgabe 6.3 In § 4.6.3 haben Sie gelernt, dass das Konditional eine abgeleitete Form ist. Wissen Sie noch, auf welchen Ausdruck der Kernsprache ein Konditional *if* e_1 *then* e_2 *else* e_3 reduziert?

6.3 Typsynonyme

Punkte in der Ebene können wir durch Paare aus zwei reellen Zahlen darstellen. Wenn wir mit dieser Darstellung arbeiten, kann es sinnvoll sein, mithilfe der Deklaration

```
type point = real * real
```

ein **Typsynonym** *point* einzuführen. Dabei handelt es sich nicht um einen neuen Typ, sondern lediglich um eine neue Bezeichnung für einen bereits existierenden Typ. Hier ist ein Beispiel für eine sinnvolle Verwendung des Typsynonyms *point*:

```
datatype object = Circle   of point * real
                | Triangle of point * point * point
```

Bei der Ausgabe ist dem Interpreter frei gestellt, ob und wo er Typsynonyme verwendet. Beispielsweise kann ein Interpreter für die Deklaration

```
fun mirror ((x,y):point) = (x,~y)
```

eine der folgenden Ausgaben liefern:

*val mirror : point → real * real*
*val mirror : real * real → real * real*

Bei den in diesem Buch gezeigten Interpreterausgaben nehmen wir uns die Freiheit, Typsynonyme gemäß unseren Intentionen zu verwenden:

```
fun mirror ((x,y):point) = (x,~y)
```
val mirror : point → point

Statt eines Typsynonyms können wir auch einen neuen Typ für Punkte einführen:

```
datatype point = Point of real * real
```

Damit zwingen wir den Interpreter, den Typ *point* bei der Ausgabe zu verwenden:

```
Point (2.0, 3.0)
```
Point (2.0, 3.0) : point

```
fun mirror (Point(x,y)) = Point(x,~y)
```
val mirror : point → point

Aufgabe 6.4 Wie wird die Deklaration *fun mirror* $(x,y) = (x,\sim y)$ getypt?

6.4 Darstellung arithmetischer Ausdrücke

Mithilfe von Konstruktortypen können wir syntaktische Objekte als Werte darstellen. Als Beispiel betrachten wir Ausdrücke, die mit Konstanten und Variablen sowie Addition und Multiplikation gebildet sind (z.B. $x(y+7)+3$).

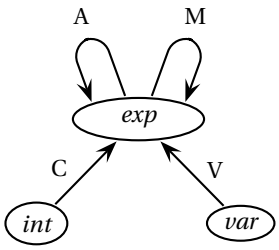
Variablen stellen wir durch Strings dar:

```
type var = string
```

Ausdrücke (engl. expressions) stellen wir durch Werte des Konstruktortyps

```
datatype exp = C of int | V of var | A of exp * exp | M of exp * exp
```

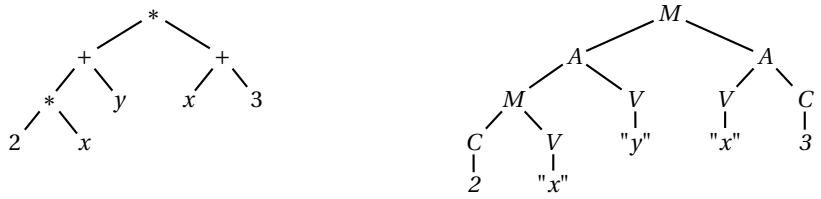
dar. Jede der vier Ausdrucksformen (Konstante, Variable, Summe, Produkt) wird mithilfe eines Konstruktors dargestellt. Da die Konstruktoren für Summen und Produkte Ausdrücke zu größeren Ausdrücken zusammensetzen, ist die Deklaration des Typs *exp* **rekursiv**. Grafisch können wir uns die Struktur des Typs *exp* wie folgt veranschaulichen:



Als Beispiel betrachten wir die Darstellung des Ausdrucks $(2x+y)(x+3)$:

```
val e = M(A(M(C 2, V "x"), V "y"), A(V "x", C 3))
```

Es ist aufschlussreich, der Baumdarstellung des Ausdrucks die Baumdarstellung des Werts *e* gegenüberzustellen:



6.4.1 Komponenten und Teilausdrücke

Können Sie präzise sagen, wie sich die Begriffe Komponente und Teilausdruck aus § 2.1 auf die hier betrachteten Ausdrücke übertragen? Versuchen Sie, die Komponenten und Teilausdrücke unseres Beispielausdrucks anzugeben.

Eine Möglichkeit, den Begriff Komponente für unsere Ausdrücke präzise zu definieren, ist die Angabe einer Prozedur, die die Komponenten eines Ausdrucks als Elemente einer Liste liefert:

```
fun components (A(e,e')) = [e, e']
  | components (M(e,e')) = [e, e']
  | components     _     = nil
```
val components : exp → exp list

Die Definition eines Begriffs durch eine Prozedur hat den Vorteil der *Ausführbarkeit*: Wenn Sie wissen wollen, was die Komponenten eines Ausdrucks sind, können Sie einen Interpreter fragen:

```
components (A(C 3, V "z"))
```
[C 3, V "z"] : exp list

Die Teilausdrücke eines Ausdrucks *e* sind rekursiv wie folgt definiert:

1. *e* ist ein Teilausdruck von *e*.
2. Jeder Teilausdruck jeder Komponente von *e* ist ein Teilausdruck von *e*.

Eine ausführbare Version dieser Definition erhalten wir durch eine Prozedur, die zu einem Ausdruck die Liste aller seiner Teilausdrücke liefert:

```
fun subexps e = e::
    (case e of
         A(e1,e2) => subexps e1 @ subexps e2
       | M(e1,e2) => subexps e1 @ subexps e2
       |    _     => nil)
```
val subexps : exp → exp list

Beachten Sie, dass die Prozedur *subexps* binär rekursiv ist. Die Terminierung von *subexps* ergibt sich aus der Tatsache, dass die rekursiven Anwendungen auf die Komponenten des aktuellen Ausdrucks erfolgen. Also werden mit fortschreitender Rekursion immer kleinere Ausdrücke behandelt. Machen Sie sich klar, dass der Rekursionsbaum eines Aufrufs *subexps e* dieselbe Form hat wie die Baumdarstellung des Ausdrucks *e*.

Eine Rekursion, die so wie bei *subexps* über die Komponenten zusammengesetzter Objekte verläuft, wird als **strukturelle Rekursion** bezeichnet. Strukturell rekursive Prozeduren haben wir erstmals im Zusammenhang mit Listen kennengelernt (z.B. *length*). Da nichtleere Listen nur eine Komponente haben, die wieder eine Liste ist, sind strukturell rekursive Prozeduren für Listen linear rekursiv.

Aufgabe 6.5 Deklarieren Sie eine Prozedur $vars : exp \to var\ list$, die zu einem Ausdruck eine Liste liefert, die die in dem Ausdruck vorkommenden Variablen enthält. Orientieren Sie sich an der Prozedur *subexps*.

Aufgabe 6.6 Deklarieren Sie eine Prozedur $count : var \to exp \to int$, die zählt, wie oft eine Variable in einem Ausdruck auftritt. Beispielsweise tritt x in $x + x$ zweimal auf.

Aufgabe 6.7 Deklarieren Sie eine Prozedur $check : exp \to exp \to bool$, die für zwei Ausdrücke e und e' testet, ob e ein Teilausdruck von e' ist.

6.4.2 Darstellung von Umgebungen

Um den Wert eines Ausdrucks bestimmen zu können, benötigen wir eine Umgebung, die den im Ausdruck vorkommenden Variablen Werte zuweist (siehe § 2.4). Beispielsweise liefert der Ausdruck $(2x + y)(x + 3)$ in der Umgebung $[x := 5, y := 3]$ den Wert 104, und in der Umgebung $[x := 0, y := 2]$ den Wert 6.

Umgebungen (engl. environments) stellen wir durch Prozeduren dar, die den Wert von Variablen liefern:

```
type env = var -> int
```

Wenn eine Umgebung den Wert einer Variablen nicht kennt, soll sie die Ausnahme *Unbound* werfen:

```
exception Unbound
```

Die Umgebung $[x := 5,\ y := 3]$ können wir wie folgt darstellen:

```
val env = fn "x" => 5 | "y" => 3 | _ => raise Unbound
```

Die obigen Deklarationen binden den Bezeichner *env* zweifach: Einerseits bezeichnet *env* einen Typ, andererseits einen Wert (eine Prozedur). Diese Doppelbindung ist zulässig, da der Kontext eines Bezeichnerauftretens immer eindeutig festlegt, ob ein Typ oder ein Wert benötigt wird.

Hier ist eine strukturell rekursive Prozedur, die den Wert eines Ausdrucks in einer Umgebung liefert:

```
fun eval env (C c)     = c
  | eval env (V v)     = env v
  | eval env (A(e,e')) = eval env e + eval env e'
  | eval env (M(e,e')) = eval env e * eval env e'
val eval : env → exp → int

eval env e
104 : int
```

Aufgabe 6.8 Schreiben Sie eine Prozedur *instantiate* : *env* → *exp* → *exp*, die zu einer Umgebung V und einem Ausdruck e den Ausdruck liefert, den man aus e erhält, indem man die in e vorkommenden Variablen gemäß V durch Konstanten ersetzt. Beispielsweise soll für die oben deklarierte Umgebung *env* und den Ausdruck $A(V\,\text{"}x\text{"},\ V\,\text{"}y\text{"})$ der Ausdruck $A(C\,5,\ C\,3)$ geliefert werden. Orientieren Sie sich an der Prozedur *eval*.

Aufgabe 6.9 (Symbolisches Differenzieren) Sie sollen eine Prozedur schreiben, die Ausdrücke nach der Variable x ableitet. Hier ist ein Beispiel:

$$(x^3 + 3x^2 + x + 2)' = 3x^2 + 6x + 1$$

Ausdrücke sollen gemäß des folgenden Typs dargestellt werden:

```
datatype exp = C of int                 c
             | X                        x
             | A of exp * exp           u + v
             | M of exp * exp           u · v
             | P of exp * int           u^n
```

a) Schreiben Sie eine Deklaration, die den Bezeichner u an die Darstellung des Ausdrucks $x^3 + 3x^2 + x + 2$ bindet. Der Operator + soll dabei links klammern.

b) Schreiben Sie eine Prozedur *derive* : *exp* → *exp*, die die Ableitung eines Ausdrucks gemäß den folgenden Regeln berechnet:

$$\begin{aligned} c' &= 0 \\ x' &= 1 \\ (u+v)' &= u' + v' \\ (u \cdot v)' &= u' \cdot v + u \cdot v' \\ (u^n)' &= n \cdot u^{n-1} \cdot u' \end{aligned}$$

Die Ableitung darf vereinfachbare Teilausdrücke enthalten (z.B. $0 \cdot u$).

c) Schreiben Sie eine Prozedur *simplifyTop* : *exp* → *exp*, die versucht, einen Ausdruck auf oberster Ebene durch die Anwendung einer der folgenden Regeln zu vereinfa-

chen:

$$\begin{array}{rcl@{\qquad\qquad}rcl} 0+u & \to & u & u+0 & \to & u \\ 0\cdot u & \to & 0 & u\cdot 0 & \to & 0 \\ 1\cdot u & \to & u & u\cdot 1 & \to & u \\ u^0 & \to & 1 & u^1 & \to & u \end{array}$$

Wenn keine der Regeln auf oberster Ebene anwendbar ist, soll der Ausdruck unverändert zurückgeliefert werden.

d) Schreiben Sie eine Prozedur *simplify* : *exp* → *exp*, die einen Ausdruck gemäß der obigen Regeln solange vereinfacht, bis keine Regel mehr anwendbar ist. Gehen Sie bei zusammengesetzten Ausdrücken wie folgt vor:
   1. Vereinfachen Sie zuerst die Komponenten.
   2. Vereinfachen Sie dann den Ausdruck mit den vereinfachten Komponenten mithilfe von *simplifyTop*.

Aufgabe 6.10 (Konstruktordarstellung natürlicher Zahlen) In dieser Aufgabe stellen wir die natürlichen Zahlen mit den Werten des Konstruktortyps

```
datatype nat = O | S of nat
```

dar: $0 \mapsto O$, $1 \mapsto SO$, $2 \mapsto S(SO)$, $3 \mapsto S(S(SO))$, und so weiter.

a) Deklarieren Sie eine Prozedur *code* : *int* → *nat*, die die Darstellung einer natürlichen Zahl liefert.

b) Deklarieren Sie eine Prozedur *decode* : *nat* → *int*, sodass *decode*(*code n*) = *n* für alle $n \in \mathbb{N}$ gilt.

c) Deklarieren Sie für *nat* kaskadierte Prozeduren *add*, *mul* und *less*, die den Operationen +, ∗ und < für natürliche Zahlen entsprechen. Verwenden Sie dabei keine Operationen für *int*.

Aufgabe 6.11 (Konstruktordarstellung ganzer Zahlen) In dieser Aufgabe stellen wir die ganzen Zahlen mit den Werten des Konstruktortyps

```
datatype integer = N of nat | P of nat
```

dar: $0 \mapsto PO$, $1 \mapsto P(SO)$, $2 \mapsto P(S(SO))$, $-1 \mapsto NO$, $-2 \mapsto N(SO)$, und so weiter. Der Typ *nat* sei dabei wie in Aufgabe 6.10 auf S. 121 definiert.

a) Deklarieren Sie eine Prozedur *code*′ : *int* → *integer*, die die Darstellung einer ganzen Zahl liefert. Verwenden Sie die Prozedur *code* für *nat*.

b) Deklarieren Sie eine Prozedur *decode*′ : *integer* → *int*, die zu einer Darstellung die dargestellte Zahl liefert. Verwenden Sie die Prozedur *decode* für *nat*.

c) Deklarieren Sie für *integer* eine kaskadierte Prozedur *add*′, die der Addition für ganze Zahlen entspricht. Verwenden Sie die Prozedur *add* für *nat*, aber keine Operationen für *int*. Sie benötigen 8 Regeln für *add*′. Wenn Sie die Kommutativität der Addition ausnutzen und die letzte Regel die Argumente vertauschen lassen, können Sie mit 6 Regeln auskommen.

6.5 Ausnahmen

Wir haben bereits gesehen, dass die Ausführung des Ausdrucks

```
raise Empty
! Uncaught exception: Empty
```

keinen Wert liefert, sondern die Ausnahme *Empty* wirft. Jetzt lernen wir, wie man neue Ausnahmen deklariert und wie man geworfene Ausnahmen wieder fängt.

Zunächst machen wir die überraschende Entdeckung, dass es sich bei Ausnahmen um Werte eines Typs *exn* handelt:

```
Empty
Empty : exn
```

Dabei ist der Typ *exn* als Konstruktortyp zu verstehen, und *Empty* als nullstelliger Konstruktor. Der Konstruktortyp *exn* hat die Besonderheit, dass man ihn durch die Deklaration neuer Konstruktoren erweitern kann. Beispielsweise führt die Deklaration

```
exception New
exn New : exn
```

einen neuen nullstelligen **Ausnahmekonstruktor** *New* ein. Man kann auch einstellige Ausnahmekonstruktoren deklarieren:

```
exception Newer of int
exn Newer : int → exn
```

Die folgenden Beispiele zeigen, dass Ausnahmekonstruktoren wie normale Konstruktoren verwendet werden können:

```
(Overflow, New, Newer)
(Overflow, New, fn) : exn * exn * (int → exn)

fun test New       = 0
  | test (Newer x) = x
  | test _         = ~1
val test : exn → int

test Overflow
~1 : int

test (Newer 13)
13 : int
```

6.5.1 Werfen von Ausnahmen

Mit einem Ausdruck der Form

raise ⟨*Ausdruck*⟩

können Ausnahmen **geworfen** werden. Der auf das Schlüsselwort *raise* folgende Ausdruck muss dabei den Typ *exn* haben. Hier ist ein Beispiel:

```
raise New
```
!Uncaught exception: New

Da Raise-Ausdrücke keinen Wert liefern, können sie jeden Typ annehmen:

```
fun f x y = if x then y else raise New
```
val f : bool→α→α

6.5.2 Fangen von Ausnahmen

Wenn die Auswertung eines Ausdrucks terminiert, liefert sie entweder einen Wert oder wirft eine Ausnahme. Mit Ausdrücken der Form

⟨*Ausdruck*⟩ *handle* ⟨*Regel*⟩ | ... | ⟨*Regel*⟩

können geworfene Ausnahmen mithilfe von Regeln gefangen werden:

```
(raise New) handle New => ()
```
() : unit

```
(raise Newer 7) handle Newer x => x
```
7 : int

```
fun test f = f() handle Newer x => x | Overflow => ~1
```
val test : (unit → int) → int

```
test (fn () => raise Newer 6)
```
6 : int

```
fun fac n = if n<1 then 1 else n*fac(n-1)
```
val fac : int → int

```
fac 15
```
! Uncaught exception: Overflow

```
test (fn () => fac 15)
```
~1 : int

Eine Prozedur, die die Adjunktion (§ 2.7) zweier gemäß § 6.4.2 dargestellten Umgebungen liefert, können wir wie folgt deklarieren:

```
fun adjoin env env' x = env' x handle Unbound => env x
```
val adjoin : (α → β) → (α → β) → α → β

Da die Deklaration keine expliziten Hinweise darauf enthält, dass es sich bei *env* und *env′* um Umgebungen handelt, wird *adjoin* polymorph getypt. Die Adjunktion von Umgebungen erfolgt gemäß der Instanz *env* → *env* → *env*.

6.5.3 Ausführungsreihenfolge und Sequenzialisierung

Standard ML schreibt eine strikte links-nach-rechts Ausführung vor. Beispielsweise müssen die Komponenten eines Tupel-Ausdrucks $(e_1,\ldots,e_n)$ in der angegebenen Reihenfolge ausgeführt werden. Die Ausführung des Ausdrucks

```
(raise Overflow, raise Subscript)
```

muss also die Ausnahme *Overflow* werfen.

Beim Programmieren mit Ausnahmen sind manchmal Ausdrücke der Form

(⟨*Ausdruck*⟩ ; … ; ⟨*Ausdruck*⟩)

hilfreich. Solche Ausdrücke werden als **Sequenzialisierungen** bezeichnet. Eine Sequenzialisierung $(e_1;\ldots;e_n)$ wird ausgeführt, indem die Teilausdrücke $e_1,\ldots,e_n$ in der angegebenen Reihenfolge ausgeführt werden. Wenn die Ausführung aller Teilausdrücke regulär terminiert, wird der Wert des letzten Teilausdrucks e_n geliefert. Hier sind Beispiele:

```
(5 ; 7)
```
7 : int

```
(raise New ; 7)
```
! Uncaught exception: New

Ein typisches Beispiel für die Verwendung von Sequenzialisierungen beim Programmieren mit Ausnahmen ist die folgende Prozedur, die testet, ob die Multiplikation zweier Zahlen zu einem Überlauf führt:

```
fun testOverflow x y = (x*y ; false) handle Overflow => true
```
val testOverflow : int → int → bool

```
testOverflow 2 3
```
false : bool

```
testOverflow 100000 100000
```
true : bool

Bei Sequenzialisierungen handelt es sich um eine abgeleitete Form, die auf Tupel-Ausdrücke und Projektionen zurückgeführt werden kann:

$$(e_1;\ldots;e_n) \quad \rightsquigarrow \quad \#n\,(e_1,\ldots,e_n)$$

Für Let-Ausdrücke und Sequenzialisierungen gibt es eine Klammersparregel:

$$\mathit{let}\ \ldots\ \mathit{in}\ e_1;\ldots;e_n\ \mathit{end} \quad \rightsquigarrow \quad \mathit{let}\ \ldots\ \mathit{in}\ (e_1;\ldots;e_n)\ \mathit{end}$$

Aufgabe 6.12 Schreiben Sie eine Prozedur $\mathit{test} : \mathit{int} \to \mathit{bool}$, die testet, ob das Quadrat einer ganzen Zahl im darstellbaren Zahlenbereich liegt.

Aufgabe 6.13 Führen Sie zweistellige Sequenzialisierungen $(e_1; e_2)$ auf Abstraktionen und Applikationen zurück.

6.5.4 Konvention für die Spezifikation von Ausnahmen

Wenn wir den Typ einer Prozedur spezifizieren, die für bestimmte Argumente eine Ausnahme wirft, geben wir den entsprechenden Ausnahmekonstruktor oft wie folgt an:

$$\begin{array}{ll} \mathit{hd} & : \alpha\ \mathit{list} \to \alpha \quad (*\ \mathit{Empty}\ *) \\ \mathit{List.nth} & : \alpha\ \mathit{list} * \mathit{int} \to \alpha \quad (*\ \mathit{Subscript}\ *) \end{array}$$

6.5.5 Beispiel: Test auf Mehrfachauftreten

Wir wollen eine Prozedur schreiben, die testet, ob es in einer Liste ein Element gibt, das mehrfach auftritt (z.B. 2 in $[1,2,3,2,4]$). Dazu verwenden wir einen sortierenden Algorithmus, der auch bei langen Listen schnell zu einem Ergebnis führt.

Betrachten Sie die polymorphe Sortierprozedur in Abbildung 5.2 auf S. 101. Sie vergleicht die Elemente einer Liste mit einer Vergleichsprozedur *compare*. Wir machen die Beobachtung, dass es in einer Liste genau dann ein Mehrfachauftreten gibt, wenn *compare* beim Sortieren der Liste mit dieser Prozedur mindestens einmal *Equal* liefert. Wir bezeichnen diese Eigenschaft der polymorphen Sortierprozedur als **Regularität**.

Unser Test auf Mehrfachauftreten sortiert die zu testende Liste mit einer regulären Sortierprozedur und einer maskierten Vergleichsprozedur, die eine Ausnahme *Double* wirft, sobald ein Vergleich das Ergebnis *EQUAL* liefert. Also enthält eine Liste genau dann ein Mehrfachauftreten, wenn beim Sortieren die Ausnahme *Double* geworfen wird. Wir realisieren diesen Algorithmus mit der in Abbildung 6.1 gezeigten Prozedur *testDouble*.

Aufgabe 6.14 Schreiben Sie die Prozedur *testDouble* aus Abbildung 6.1 so um, dass die Ausnahme *Double* und die Hilfsprozedur *mask* mithilfe eines Let-Ausdrucks lokal deklariert werden.

Aufgabe 6.15 Die Prozedur *testDouble* testet auch sehr lange Listen schnell auf Mehrfachauftreten. Schreiben Sie einen Test auf Mehrfachauftreten, der ohne Sortieren arbeitet, und überzeugen Sie sich mit der Liste $[1,\ldots,10000]$ davon, dass *testDouble* sehr viel schneller ist. Daran ändert sich auch nichts, wenn Sie statt *List.sort* eine selbstgeschriebene Sortierprozedur verwenden, die durch Mischen sortiert.

```
exception Double

fun mask compare p = case compare p of
    EQUAL => raise Double | v => v

fun testDouble compare xs =
    (List.sort (mask compare) xs ; false)
    handle Double => true
```

*val testDouble : (α * α → order) → α list → bool*

Abbildung 6.1: *Test auf Mehrfachauftreten*

6.6 Typkonstruktoren

Betrachten Sie die **parametrisierte Typdeklaration**

```
datatype 'a mylist = Nil | Cons of 'a * 'a mylist
```

Diese deklariert einen **Typkonstruktor** *mylist*, der für jeden Typ *t* einen Typ *t mylist* liefert. Die Werte der Typen *t mylist* werden mit den polymorphen Konstruktoren

Nil : α *mylist*
Cons : $\alpha * \alpha$ *mylist* $\rightarrow \alpha$ *mylist*

gebildet. Das Gespann aus *mylist*, *Nil* und *Cons* liefert eine Datenstruktur, die der eingebauten Datenstruktur für Listen entspricht.

Beachten Sie, dass wir den Begriff Konstruktor für Werte und Typen verwenden. Zum Beispiel ist *mylist* ein Konstruktor, der Typen liefert, und *Cons* ein Konstruktor, der Werte liefert. Außerdem ist *int mylist* ein Konstruktortyp, der mit dem Typkonstruktor *mylist* gebildet wird.

Aufgabe 6.16 Schreiben Sie eine Prozedur

append : α *mylist* $\rightarrow \alpha$ *mylist* $\rightarrow \alpha$ *mylist*

die zwei gemäß des Typkonstruktors *mylist* dargestellte Listen konkateniert.

6.7 Optionen

Wir kommen jetzt zu dem vordeklarierten Typkonstruktor *option*:

```
datatype 'a option = NONE | SOME of 'a
```

Die mit *option* beschreibbaren Typen werden als **Optionstypen** bezeichnet, und ihre Werte als **Optionen**. Die mit *SOME* konstruierten Optionen werden als **eingelöst** bezeichnet und die mit *NONE* konstruierten als **uneingelöst**.

Als Beispiel für die Verwendung von Optionen betrachten wir eine Prozedur *nth*, die das n-te Element einer Liste liefert:

```
fun nth n xs = if n<0 orelse null xs then NONE
               else if n=0 then SOME (hd xs) else nth (n-1) (tl xs)
```
val nth : int → α list → α option

```
nth 2 [3,4,5]
```
SOME 5 : int option

```
nth 3 [3,4,5]
```
NONE : int option

Optionen stellen also eine Alternative zu Ausnahmen dar. Durch die Verwendung von Optionen wird der mögliche Ausnahmefall im Ergebnistyp der Prozedur als Option sichtbar.

Die vordeklarierte Prozedur

```
fun valOf (SOME x) = x
  | valOf NONE     = raise Option.Option
```
val valOf : α option → α

erlaubt den bequemen Zugriff auf eingelöste Optionen:

```
valOf (nth 2 [3,4,5])
```
5 : int

Die vordeklarierte Prozedur

```
fun isSome NONE     = false
  | isSome (SOME _) = true
```
val isSome : α option → bool

testet, ob es sich bei einer Option um eine eingelöste Option handelt.

Der vordeklarierte Bezeichner *Int.minInt* ist an eine Option gebunden, die Auskunft über die kleinste darstellbare ganze Zahl gibt:

```
Int.minInt
```
SOME ~1073741824 : int option

Wenn *Int.minInt* an die uneingelöste Option gebunden ist, bedeutet das, dass der Interpreter beliebig kleine Zahlen darstellen kann. Analog gibt es den vordeklarierten Bezeichner *Int.maxInt*, dessen Wert Auskunft über die größte darstellbare ganze Zahl gibt:

```
Int.maxInt
```
SOME 1073741823 : int option

```
valOf Int.minInt + valOf Int.maxInt
```
~1 : int

```
fun findDouble compare xs = let
    exception Double of 'a
    fun compare' (x,y) = case compare (x,y) of
        EQUAL => raise Double x | v => v
in
    (List.sort compare' xs ; NONE)
    handle Double x => SOME x
end
```

*val findDouble : (α * α → order) → α list → α option*

Abbildung 6.2: *Suche eines Mehrfachauftretens*

Wir wenden uns jetzt nochmals dem in § 6.5.5 behandelten Test auf Mehrfachauftreten zu. Wir wollen den Test so modifizieren, dass er als Ergebnis eine Option liefert, die im Erfolgsfall mit einem mehrfach auftretenden Element der Liste eingelöst ist. Dazu werfen wir bei der Entdeckung eines mehrfach auftretenden Elements eine mit einem einstelligen Ausnahmekonstruktor gebildete Ausnahme, die den gefundenen Wert enthält. Abbildung 6.2 zeigt die Realisierung einer entsprechenden Prozedur. Der Ausnahmekonstruktor *Double* wird dabei mit einer Typvariable deklariert, damit die Prozedur polymorph getypt werden kann.

Aufgabe 6.17 Schreiben Sie eine Prozedur *last* : *α list* → *α option*, die das letzte Element einer Liste liefert.

Aufgabe 6.18 Schreiben Sie eine Prozedur *find* : (*α* → *bool*) → *α list* → *α option*, die zu einer Prozedur und einer Liste das erste Element der Liste liefert, für das die Prozedur *true* liefert.

Bemerkungen

Konstruktoren und die entsprechenden Typen dienen dazu, die mit Tupeln verbundenen Ausdrucksmöglichkeiten im Rahmen einer getypten Programmiersprache verfügbar zu machen. Mit den uns jetzt zur Verfügung stehenden programmiersprachlichen Ausdrucksmitteln können wir ein reiches Repertoire an Datenstrukturen und Algorithmen realisieren. Vorerst werden wir keine weiteren programmiersprachlichen Konstrukte benötigen.

Verzeichnis

Konstruktoren: nullstellig, einstellig; Konstruktortypen, Varianten, Typkonstruktoren, (rekursive) Typdeklaration; Variantennummer, Datum; Enumerationstypen.

Lesart von Bezeichnerauftreten.

Typsynonyme.

Ausnahmen: Typ *exn*, Deklaration von Ausnahmekonstruktoren, Werfen und Fangen von Ausnahmen; Sequenzialisierungen.

Optionen: eingelöste und uneingelöste; Typkonstruktor *option*, Optionstypen, Konstruktoren *NONE* und *SOME*, Ausnahmekonstruktor *Option*, vordeklarierte Prozeduren *isSome* und *valOf*.

Darstellung arithmetischer Ausdrücke; Darstellung von Umgebungen.

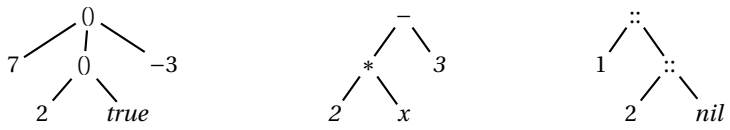
Konstruktordarstellung natürlicher und ganzer Zahlen.

Strukturelle Rekursion.

Test auf Mehrfachauftreten mit regulären Sortierprozeduren.

7 Bäume

Viele der für die Programmierung wichtigen Objekte haben einen baumartigen Aufbau. Dazu gehören geschachtelte Tupel, Ausdrücke und Listen:



Wir führen jetzt ein Standardmodell ein, mit dem wir den Begriff des baumartigen Objekts erklären und die für solche Objekte üblichen Sprechweisen definieren werden. Die Objekte des Standardmodells bezeichnen wir als Bäume.

Wir realisieren das Standardmodell durch ein Programm. Damit bekommen wir eine formale Beschreibung des Standardmodells, die die Objekte und Begriffe des Modells bis ins letzte Detail festlegt. Darüber hinaus liefert uns das Programm ein ausführbares Modell, mit dem wir nach Belieben experimentieren können.

7.1 Reine Bäume

Wir beginnen mit einer besonders einfachen Klasse von Bäumen, die wir als **reine Bäume** bezeichnen. Reine Bäume werden gemäß einer rekursiven Konstruktionsvorschrift gebildet: *Wenn* $t_1, \ldots, t_n$ *reine Bäume sind, dann ist die Liste* $[t_1, \ldots, t_n]$ *ein reiner Baum.* Reine Bäume sind also ineinander geschachtelte Listen. Der einfachste reine Baum ist die leere Liste. Ausgehend von der leeren Liste können dann komplexere reine Bäume gebildet werden. In Standard ML stellen wir reine Bäume gemäß der folgenden Typdeklaration dar:

```
datatype tree = T of tree list
```

Hier sind Beispiele für reine Bäume:

```
val t1 = T[]
val t2 = T[t1, t1, t1]
val t3 = T[T[t2], t1, t2]
```

Diese Bäume können wir wie in Abbildung 7.1 gezeigt grafisch darstellen. Gemäß unserem bisherigem Sprachgebrauch handelt es sich bei der grafischen Darstellung eines Baums um die „Baumdarstellung“ des Baums.

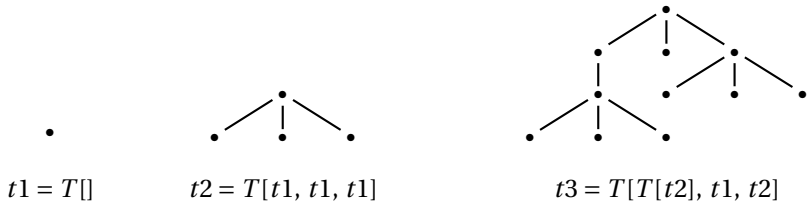
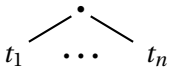


Abbildung 7.1: *Grafische Darstellung reiner Bäume*

Da wir in diesem Abschnitt nur reine Bäume betrachten, werden wir sie im Folgenden einfach als Bäume bezeichnen.

Die grafische Darstellung von Bäumen ist sehr hilfreich für das Verständnis von Bäumen. Die grafische Darstellung eines Baums $T[t_1,\ldots,t_n]$ erhält man, indem man die grafischen Darstellungen der **Unterbäume** $t_1,\ldots,t_n$ mit einem Knoten • und n als **Kanten** bezeichneten Strichen verbindet:



Der oberste Knoten der grafischen Darstellung eines Baums wird als **Wurzel** bezeichnet. Knoten, von denen keine Kante zu einem tiefer liegenden Knoten führt, werden als **Blätter** bezeichnet. Knoten, die keine Blätter sind, werden als **innere Knoten** bezeichnet.

Als Beispiel betrachten wir die grafische Darstellung des Baums $t3$ in Abbildung 7.1. Sie besteht aus 11 Knoten und 10 Kanten. Bei 7 der Knoten handelt es sich um Blätter, die restlichen 4 sind innere Knoten.

Generell gilt für die Darstellung eines Baums, dass sie mindestens einen Knoten enthält. Außerdem ist die Anzahl der Kanten immer um eins kleiner als die Anzahl der Knoten, da wir jedem Knoten, der verschieden von der Wurzel ist, die von oben auf ihn zeigende Kante zuordnen können.

Der Baum $T[]$ wird als **atomar** bezeichnet. Alle anderen Bäume werden als **zusammengesetzt** bezeichnet. Es gibt also genau einen atomaren Baum.

Wir wollen kurz erklären, wo die Bezeichnung Baum herkommt. Wenn wir die grafische Darstellung eines reinen Baums um 180 Grad drehen, bekommen wir ein Bild, das an einen sich verzweigenden natürlichen Baum erinnert. Dem Begriff des Stammbaums liegt übrigens dieselbe bildhafte Vorstellung zugrunde.

Aufgabe 7.1 Schreiben Sie eine Prozedur $compound: tree \rightarrow bool$, die testet, ob ein Baum zusammengesetzt ist.

Aufgabe 7.2 Zeichnen Sie die Darstellung des Baums $T[t2, t1, t2]$.

Aufgabe 7.3 Geben Sie Ausdrücke an, die Bäume mit den unten gezeigten grafischen Darstellungen beschreiben. Dabei können Sie die oben deklarierten Bezeichner $t1$ und $t2$ verwenden.

Aufgabe 7.4 Sei die grafische Darstellung eines Baums gegeben. Nehmen Sie an, dass die Darstellung $n \geq 1$ Kanten enthält und beantworten Sie die folgenden Fragen:

a) Wie viele Knoten enthält die Darstellung mindestens/höchstens?

b) Wie viele Blätter enthält die Darstellung mindestens/höchstens?

c) Wie viele innere Knoten enthält die Darstellung mindestens/höchstens?

7.1.1 Unterbäume

Sei $t = T[t_1, \ldots, t_n]$ ein Baum. Dann bezeichnen wir die Zahl n als die **Stelligkeit** von t und $t_1, \ldots, t_n$ als die **Unterbäume** von t. Darüber hinaus bezeichnen wir t_k als den **k-ten Unterbaum** von t (für $k \in \{1, \ldots, n\}$). Hier sind Prozeduren, die die Stelligkeit und die Unterbäume eines Baums liefern:

```
fun arity (T ts) = length ts
```
val arity : tree → int

```
arity t3
```
3 : int

```
fun dst (T ts) k = List.nth(ts, k-1)
```
val dst : tree → int → tree

```
dst t3 3
```
T[T[], T[], T[]] : tree

Die Prozedur *dst* liefert zu einem Baum t und einer positiven Zahl k den k-ten Unterbaum von t. Die Unterbäume werden dabei wie oben vereinbart mit 1 beginnend durchnummeriert. Wenn der Baum keinen k-ten Unterbaum hat, wird die Ausnahme *Subscript* geworfen. Der Bezeichner *dst* steht für direct subtree, der englischen Bezeichnung für Unterbaum.

7.1.2 Gestalt arithmetischer Ausdrücke

Die arithmetischen Ausdrücke aus § 6.4

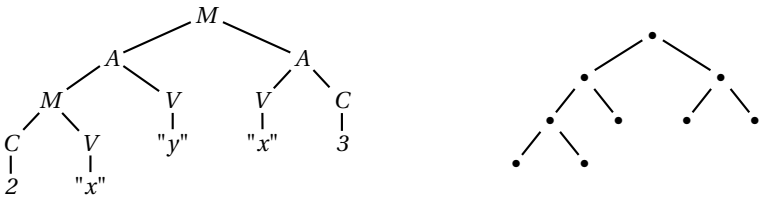
```
datatype exp = C of int | V of string | A of exp*exp | M of exp*exp
```

lassen sich als Bäume auffassen, deren Knoten mit zusätzlichen Informationen versehen sind. Dabei liefern die Konstruktoren C und V nullstellige und die Konstruktoren A und

M zweistellige Bäume. Diesen Zusammenhang zwischen arithmetischen Ausdrücken und reinen Bäumen können wir durch eine Prozedur *shape* : *exp* → *tree* präzise machen, die die **Gestalt** eines Ausdrucks liefert:

```
fun shape (C _) = T[]
  | shape (V _) = T[]
  | shape (A(e,e')) = T[shape e, shape e']
  | shape (M(e,e')) = T[shape e, shape e']
val shape : exp → tree
```

Hier ist ein Beispiel für einen Ausdruck und seine Gestalt:



Aufgabe 7.5 Geben Sie die Gestalt des Ausdrucks $(x+3)(y+7)$ an.

Aufgabe 7.6 Geben Sie einen Ausdruck mit der Gestalt $T[T[t1,t1],\ t1]$ an.

7.1.3 Lexikalische Baumordnung

Das lexikalische Ordnungsprinzip für Listen (§ 5.3) liefert bei rekursiver Anwendung eine Ordnung für reine Bäume, die als **lexikalische Baumordnung** bezeichnet wird:

```
fun compareTree (T ts, T tr) = List.collate compareTree (ts,tr)
val compareTree : tree * tree → order

compareTree(t2,t3)
LESS : order

compareTree(T[T[T[T[T[]]]]], t3)
GREATER : order
```

Aufgabe 7.7 Ordnen Sie die folgenden Bäume gemäß der lexikalischen Ordnung an: $t1$, $t2$, $t3$, $T[T[t1]]$, $T[t1, T[T[t1]]]$, $T[T[T[T[t1]]]]$.

Aufgabe 7.8 Deklarieren Sie eine zu *compareTree* äquivalente Prozedur ohne dabei die Prozedur *List.collate* zu verwenden.

7.2 Teilbäume

Da die Unterbäume eines Baums wieder mit Unterbäumen gebildet sein können, ist es sinnvoll, von den **Teilbäumen** eines Baums zu sprechen. Wir definieren Teilbäume wie folgt:

1. Wenn t ein Baum ist, dann ist t ein Teilbaum von t.
2. Wenn t' ein Unterbaum eines Baums t ist, dann ist jeder Teilbaum von t' ein Teilbaum von t.

Hier ist eine Prozedur, die zu einem Baum t und zu einem Baum t' testet, ob t ein Teilbaum von t' ist:

```
fun subtree t (T ts) = (t = T ts) orelse
                       List.exists (subtree t) ts
```
val subtree : tree → tree → bool

```
subtree (T[t2]) t3
```
true : bool

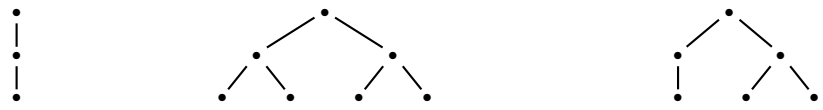
```
subtree t3 t2
```
false : bool

Wir unterscheiden zwischen Teilbäumen und ihren **Auftreten** in einem Baum. Beispielsweise tritt der Baum $t1$ dreimal als Teilbaum von $t2$ auf, und $t2$ tritt zweimal als Teilbaum von $t3$ auf (siehe Abbildung 7.1 auf S. 132). Die Auftreten der Teilbäume eines Baums entsprechen genau den Knoten der grafischen Darstellung des Baums. Damit übertragen sich die Sprechweisen für Knoten auf die Auftreten von Teilbäumen. Hier ist eine Prozedur, die zählt, wie oft ein Baum in einem Baum als Teilbaum auftritt:

```
fun count t (T ts) = if (t = T ts) then 1
                     else foldl op+ 0 (map (count t) ts)
```
val count : tree → tree → int

```
count t1 t3
7 : int
```

Ein Baum heißt **linear**, wenn jeder seiner zusammengesetzten Teilbäume einstellig ist. Ein Baum heißt **binär**, wenn jeder seiner zusammengesetzten Teilbäume zweistellig ist. Hier sind drei Beispiele:



Der linke Baum ist linear, der mittlere Baum ist binär, und der rechte Baum ist weder linear noch binär.

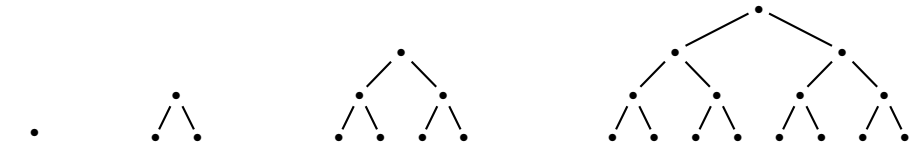
Hier ist eine Prozedur, die testet, ob ein Baum linear ist:

```
fun linear (T nil) = true
  | linear (T [t]) = linear t
  | linear _ = false
```
val linear : tree → bool

Aufgabe 7.9 Geben Sie einen Baum mit 5 Knoten an, der genau zwei Teilbäume hat. Wie viele solche Bäume gibt es?

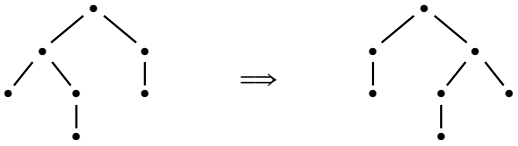
Aufgabe 7.10 Schreiben Sie eine Prozedur *binary* : *tree* → *bool*, die testet, ob ein Baum binär ist.

Aufgabe 7.11 Schreiben Sie eine Prozedur *tree* : *int* → *tree*, die für $n \geq 0$ binäre Bäume wie folgt liefert:



Achten Sie darauf, dass die identischen Unterbäume der zweistelligen Teilbäume jeweils nur einmal berechnet werden. Das sorgt dafür, dass Ihre Prozedur auch für $n = 1000$ schnell ein Ergebnis liefert. Verwenden Sie die Prozedur *iter* aus § 3.4.

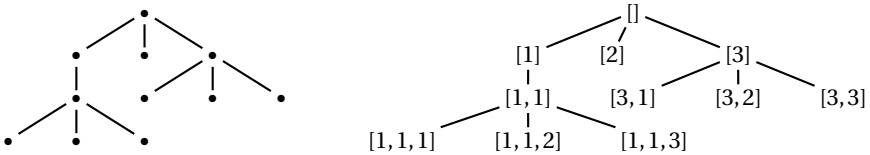
Aufgabe 7.12 (Spiegeln) Spiegeln reversiert die Ordnung der Unterbäume der Teilbäume eines Baums:



Schreiben Sie eine Prozedur *mirror* : *tree* → *tree*, die Bäume spiegelt.

7.3 Adressen

In der grafischen Darstellung eines Baums kann jeder Knoten durch eine als **Adresse** bezeichnete Liste beschrieben werden, die besagt, wie man von der Wurzel zu diesem Knoten gelangt:



Die Wurzel eines Baums hat immer die Adresse []. Die Adresse [3,2] besagt, dass man von der Wurzel beginnend zuerst der dritten und dann der zweiten nach unten gerichteten Kante folgen soll (Nummerierung von links nach rechts, beginnend mit eins).

Hier ist eine Prozedur, die zu einem Baum und zu einer Adresse den entsprechenden Teilbaum liefert:

```
fun ast t nil = t
  | ast t (n::a) = ast (dst t n) a
```
val ast : tree → int list → tree

```
ast t3 [1,1,3]
```
T[] : tree

```
ast t3 [1,1]
```
T[T[], T[], T[]] : tree

Eine Adresse a (eine Liste positiver ganzer Zahlen) heißt für einen Baum t **gültig**, wenn *ast* für t und a einen Teilbaum liefert. Wenn *ast* die Ausnahme *Subscript* wirft, sprechen wir von einer ungültigen Adresse:

```
ast t3 [1,2]
```
! Uncaught exception: Subscript

Proposition 7.1 *Zwei Bäume sind genau dann gleich, wenn sie die gleichen gültigen Adressen haben.*

Insgesamt haben wir jetzt drei Möglichkeiten für die Beschreibung eines Baums:

1. Mithilfe eines Ausdrucks des Typs *tree*.
2. Mithilfe der Menge seiner gültigen Adressen.
3. Mithilfe einer grafischen Darstellung.

Die Darstellungen von Bäumen mithilfe von Ausdrücken und gültigen Adressen eignen sich zum Programmieren. Die grafische Darstellung von Bäumen ist für Menschen sehr anschaulich, eignet sich aber nicht zum Programmieren.

Für die nächste Proposition benötigen wir einen weiteren Begriff. Eine Liste xs heißt [**echtes**] **Präfix** einer Liste ys, wenn es eine [nichtleere] Liste zs mit $xs@zs = ys$ gibt.

Proposition 7.2 *Für jeden Baum t gilt:*

1. *[] ist eine gültige Adresse für t.*
2. *Wenn a eine gültige Adresse für t ist, dann ist jedes Präfix von a eine gültige Adresse für t.*
3. *Wenn $a@[n]$ eine gültige Adresse für t ist und $1 \leq k \leq n$, dann ist $a@[k]$ eine gültige Adresse für t.*

Aufgabe 7.13 Betrachten Sie die grafische Darstellung des Baums $t3$ in Abbildung 7.1.

a) Geben Sie die Adresse der Wurzel an.

b) Geben Sie die Adressen der inneren Knoten an (es gibt genau 4).

c) Geben Sie die Adressen der Auftreten des Teilbaums $t2$ an.

Aufgabe 7.14 Die arithmetischen Ausdrücke aus §6.4 lassen sich wie in §7.1.2 besprochen als Bäume auffassen. Schreiben Sie analog zu *ast* eine Prozedur *ase*: $exp \to int\ list \to exp$, die zu einem Ausdruck und einer Adresse den entsprechenden Teilausdruck liefert. Beispielsweise soll *ase* für den Ausdruck $x(2y+3)$ und die Adresse $[2,1]$ den Teilausdruck $2y$ liefern. Für ungültige Adressen soll die Ausnahme *Subscript* geworfen werden.

Aufgabe 7.15 Schreiben Sie Prozeduren des Typs $tree \to int\ list \to bool$ wie folgt:

a) *node* testet, ob eine Adresse einen Knoten eines Baums bezeichnet.

b) *root* testet, ob eine Adresse die Wurzel eines Baums bezeichnet.

c) *inner* testet, ob eine Adresse einen inneren Knoten eines Baums bezeichnet.

d) *leaf* testet, ob eine Adresse ein Blatt eines Baums bezeichnet.

Aufgabe 7.16 Schreiben Sie eine Prozedur *prefix*: $''a\ list \to ''a\ list \to bool$, die testet, ob eine Liste ein Präfix einer Liste ist.

Aufgabe 7.17 Deklarieren Sie *ast* mithilfe von *foldl*.

7.3.1 Nachfolger und Vorgänger

Die gültigen Adressen eines Baums entsprechen genau den Knoten in der grafischen Darstellung des Baums. Das bedeutet, dass wir Knoten durch Adressen darstellen können, und dass wir Begriffe für Knoten mithilfe von Adressen definieren können. Sei ein Baum t gegeben und seien a und a' gültige Adressen für t. Wir sagen, dass

- der durch a' bezeichnete Knoten der n-**te Nachfolger** des durch a bezeichneten Knotens ist, wenn $a' = a@[n]$ gilt.
- der durch a' bezeichnete Knoten ein **Nachfolger** des durch a bezeichneten Knotens ist, wenn eine Zahl n mit $a' = a@[n]$ existiert.
- der durch a bezeichnete Knoten der **Vorgänger** des durch a' bezeichneten Knotens ist, wenn eine Zahl n mit $a@[n] = a'$ existiert.
- der durch a' bezeichnete Knoten dem durch a bezeichneten Knoten **untergeordnet** ist, wenn eine Liste ns mit $a' = a@ns$ existiert.
- der durch a bezeichnete Knoten dem durch a' bezeichneten Knoten **übergeordnet** ist, wenn eine Liste ns mit $a@ns = a'$ existiert.

Machen Sie sich die anschauliche Bedeutung der neuen Begriffe an der grafischen Darstellung von Bäumen klar. Offensichtlich ist a der Vorgänger von a' genau dann, wenn a'

ein Nachfolger von a ist, und a ist a' übergeordnet genau dann, wenn a' a untergeordnet ist.

Proposition 7.3 *Sei t ein Baum. Dann haben alle Knoten außer der Wurzel genau einen Vorgänger, die Wurzel hat keinen Vorgänger.*

Aufgabe 7.18 Zeichnen Sie einen Baum mit mindestens zwei Knoten a und a', sodass a dem Knoten a' übergeordnet ist, aber a nicht der Vorgänger von a' ist.

Aufgabe 7.19 Schreiben Sie eine Prozedur $pred : int\ list \to int\ list \to bool$, die für zwei Adressen a und a' testet, ob es einen Baum gibt, in dem a den Vorgänger des durch a' bezeichneten Knotens bezeichnet.

Aufgabe 7.20 Schreiben Sie eine Prozedur $superior : int\ list \to int\ list \to bool$, die für zwei Adressen a und a' testet, ob es einen Baum gibt, in dem a einen Knoten bezeichnet, der dem durch a' bezeichneten Knoten übergeordnet ist.

7.4 Größe und Tiefe

Die **Größe** eines Baums definieren wir als die Anzahl seiner Knoten. Beispielsweise hat der Baum $t2$ in Abbildung 7.1 auf S. 132 die Größe 4.

Wir wollen eine Prozedur $size : tree \to int$ schreiben, die die Größe eines Baums bestimmt. Dafür benötigen wir Rekursionsgleichungen. Unser Ausgangspunkt ist die Gleichung

$$size(T[t_1, \ldots, t_n]) \;=\; 1 + size\ t_1 + \cdots + size\ t_n$$

die besagt, dass die Größe eines Baums gleich eins plus die Summe der Größen seiner Unterbäume ist. Diese Gleichung eignet sich als Rekursionsgleichung, da die Unterbäume kleiner als der daraus gebildete Baum sind. Die durch die Punkt-Punkt-Notation beschriebene Rekursion über die Unterbaumliste realisieren wir mit den Prozeduren *foldl* und *map*:

```
fun size (T ts) = foldl op+ 1 (map size ts)
```
val size : tree → int

```
size t3
```
11 : int

Machen Sie sich an einem Beispiel klar, dass der Rekursionsbaum (siehe § 5.4) für einen Prozeduraufruf *size t* dieselbe Form hat wie der Argumentbaum t, da für jeden Knoten von t genau ein Aufruf von *size* erforderlich ist.

Die **Tiefe** eines Baums definieren wir als die maximale Länge der für ihn gültigen Adressen. Anschaulich gesprochen ist die Tiefe eines Baums also die maximale Anzahl von Kanten, die in seiner grafischen Darstellung auf dem Weg von der Wurzel nach unten

durchlaufen werden kann. Beispielsweise hat der Baum $t3$ in Abbildung 7.1 auf S. 132 die Tiefe 3.

Wir wollen eine Prozedur $depth : tree \to int$ angeben, die die Tiefe eines Baums bestimmt. Unser Ausgangspunkt ist die Gleichung

$$depth(T[t_1,\ldots,t_n]) \;=\; 1+\max\{depth\ t_1,\ \ldots\ ,\ depth\ t_n\} \qquad \text{für } n>0$$

die besagt, dass die Tiefe eines zusammengesetzten Baums gleich eins plus die maximale Tiefe seiner Unterbäume ist. Mit einem kleinen Trick können wir diese Gleichung auch auf den Fall $n=0$ verallgemeinern:

$$depth(T[t_1,\ldots,t_n]) \;=\; 1+\max\{-1,\ depth\ t_1,\ \ldots\ ,\ depth\ t_n\} \qquad \text{für } n\geq 0$$

Diese Gleichung eignet sich als Rekursionsgleichung, da die Unterbäume kleiner als der daraus gebildete Baum sind. Sie liefert die folgende Prozedur:

```
fun depth (T ts) = 1 + foldl Int.max ~1 (map depth ts)
val depth : tree → int

depth t3
3 : int
```

Aufgabe 7.21 Die **Breite** eines Baums ist die Anzahl seiner Blätter. Beispielsweise hat der Baum $t3$ die Breite 7. Entwickeln Sie eine Prozedur $breadth : tree \to int$, die die Breite eines Baums bestimmt.

Aufgabe 7.22 Der **Grad** eines Baums ist die maximale Stelligkeit seiner Teilbäume. Beispielsweise hat der Baum $t3$ den Grad 3. Entwickeln Sie eine Prozedur $degree : tree \to int$, die den Grad eines Baums bestimmt.

Aufgabe 7.23 Schreiben Sie die oben angegebene Prozedur *size* so um, dass sie ohne *map* auskommt. Hilfe: Erledigen Sie die rekursive Anwendung von *size* in der Verknüpfungsprozedur für *foldl*.

Aufgabe 7.24 Schreiben Sie die oben angegebene Prozedur *depth* so um, dass sie ohne *map* auskommt.

Aufgabe 7.25 Schreiben Sie für die arithmetischen Ausdrücke aus § 6.4 Prozeduren $size : exp \to int$ und $depth : exp \to int$, die die Größe und die Tiefe von Ausdrücken liefern. Beispielsweise sollen *size* und *depth* für den Ausdruck $x(2y+3)$ die Ergebnisse 7 und 3 liefern.

7.5 Faltung

Ähnlich wie für Listen kann man für Bäume eine Faltungsprozedur angeben, mit der sich viele Funktionen auf Bäumen ohne explizite Rekursion berechnen lassen:

```
fun fold f (T ts) = f (map (fold f) ts)
val fold : (α list → α) → tree → α
```

Die Idee hinter der Faltungsprozedur besteht darin, einen Baum mit einer Schrittprozedur $f : \alpha\ list \to \alpha$ zu evaluieren, die aus den Werten der Unterbäume den Wert des Baums bestimmt. Beispielsweise lässt sich mit *fold* die Größe eines Baums wie folgt bestimmen:

```
val size = fold (foldl op+ 1)
val size : tree → int
```

Aufgabe 7.26 Deklarieren Sie mithilfe von *fold* eine Prozedur

a) *depth* : *tree* → *int*, die die Tiefe eines Baums bestimmt.

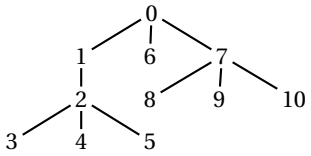
b) *breadth* : *tree* → *int*, die die Breite eines Baums bestimmt.

c) *degree* : *tree* → *int*, die den Grad eines Baums bestimmt.

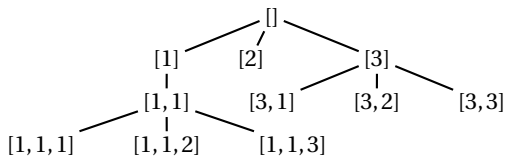
d) *mirror* : *tree* → *tree*, die einen Baum spiegelt.

7.6 Präordnung und Postordnung

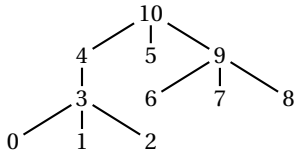
Es gibt verschiedene Möglichkeiten, die Knoten in der grafischen Darstellung eines Baums durchzunummerieren. Bei der sogenannten **Pränummerierung** beginnt man mit der Wurzel und der Zahl Null und nummeriert die Unterbäume per Rekursion von links nach rechts durch. Für den Baum $t3$ aus Abbildung 7.1 auf S. 132 ergibt sich die folgende Nummerierung:



Die durch die Pränummerierung zum Ausdruck gebrachte **Präordnung** der Knoten entspricht genau der durch die lexikalische Ordnung (§ 5.3) der Adressen gegebenen Ordnung:



Bei der sogenannten **Postnummerierung** beginnt man mit dem linkesten Blatt und der Zahl Null und arbeitet sich schrittweise von links nach rechts und von unten nach oben vor. Für unseren Beispielbaum ergibt sich damit die folgende Nummerierung:



Die durch die Postnummerierung zum Ausdruck gebrachte Ordnung der Knoten wird als **Postordnung** bezeichnet.

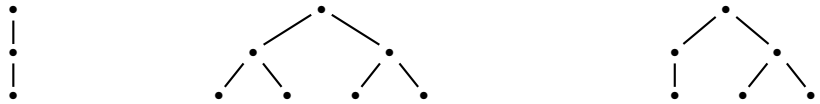
Die Prä- und Postnummerierung der Knoten der grafischen Darstellung eines Baums können bildhaft mit der sogenannten **Standardtour** erklärt werden. Die Standardtour besucht die Knoten eines Baums gemäß den folgenden Regeln:

1. Die Tour beginnt und endet an der Wurzel.
2. Die Tour folgt den Kanten des Baums. Jede Kante des Baums wird genau zweimal durchlaufen, zuerst von oben nach unten und danach von unten nach oben.
3. Die Unterbäume eines Knotens werden jeweils gemäß ihrer Ordnung besucht (also weiter links stehende Unterbäume vor weiter rechts stehenden Unterbäumen).

Ein Knoten mit n Unterbäumen wird von der Standardtour genau $n+1$-mal besucht. Die Pränummerierung ergibt sich dadurch, dass man die Knoten im Laufe der Standardtour jeweils beim ersten Besuch nummeriert, die Postnummerierung dadurch, dass man die Knoten jeweils beim letzten Besuch nummeriert.

Machen Sie sich klar, dass die Prozeduren *size* und *depth* aus § 7.4 den Argumentbaum gemäß der Standardtour durchlaufen.

Aufgabe 7.27 Geben Sie für die folgenden Bäume jeweils die Prä- und die Postnummerierung an:



Aufgabe 7.28 Die arithmetischen Ausdrücke aus § 6.4 kann man als binäre Bäume auffassen. Dabei liefern die Konstruktoren *C* und *V* atomare und die Konstruktoren *A* und *M* zusammengesetzte Bäume.

a) Schreiben Sie eine Prozedur $prelin : exp \to int\ list\ list$, die die Adressen eines Ausdrucks in Präordnung liefert.

b) Schreiben Sie eine Prozedur $postlin : exp \to int\ list\ list$, die die Adressen eines Ausdrucks in Postordnung liefert.

Aufgabe 7.29 Schreiben Sie zwei Prozeduren *prelin* und *postlin* des Typs *tree* → *int list list*, die die Adressen eines Baums in Prä- beziehungsweise Postordnung liefern.

7.6.1 Teilbaumzugriff mit Pränummern

Die durch die Pränummerierung vergebenen Nummern identifizieren die Knoten eines Baums. Wir wollen jetzt eine Prozedur entwickeln, die zu einem Baum und einer als **Pränummer** interpretierten Zahl den dadurch identifizierten Teilbaum liefert. Beispielsweise soll für den Baum $t3$ und die Zahl 2 der Baum $t2$ geliefert werden (siehe Abbildung 7.1 auf S. 132).

Um den gewünschten Teilbaumzugriff zu realisieren, schreiben wir eine allgemeinere Prozedur *prest* : *tree list* → *int* → *tree*, die zu einer Liste von Bäumen und einer Zahl den entsprechenden Teilbaum liefert. Dabei sind alle Teilbäume in der Liste gemäß der Präordnung fortlaufend durchnummeriert, sodass jeder Teilbaum in der Liste mit einer Zahl adressiert werden kann:

$$\begin{aligned} \mathit{prest}\ [t3]\ 2 &= t2 \\ \mathit{prest}\ [t2, t3]\ 0 &= t2 \\ \mathit{prest}\ [t2, t3]\ 4 &= t3 \\ \mathit{prest}\ [t2, t3]\ 6 &= t2 \end{aligned}$$

Die Prozedur *prest* lässt sich leicht mit zwei Rekursionsgleichungen beschreiben:

$$\begin{aligned} \mathit{prest}\ (t :: tr)\ 0 &= t \\ \mathit{prest}\ ((T\ ts) :: tr)\ k &= \mathit{prest}\ (ts @ tr)\ (k-1) \qquad \text{für } k > 0 \end{aligned}$$

Die Terminierung von *prest* ergibt sich aus der Tatsache, dass die Zahl, nach der gesucht wird, bei jedem Rekursionsschritt um eins verringert wird. Beachten Sie auch, dass *prest* endrekursiv ist.

Aufgabe 7.30 Schreiben Sie eine Prozedur *prest'* : *tree* → *int* → *tree*, die zu einem Baum und einer Pränummer den entsprechenden Teilbaum liefert. Wenn die angegebene Zahl keine Pränummer des Baums ist, soll die Ausnahme *Subscript* geworfen werden.

Aufgabe 7.31 Wenn wir wollen, können wir das Listenargument von *prest* als die Zusammenfassung des Primärarguments vom Typ *tree* und eines Akkus vom Typ *tree list* auffassen. Schreiben Sie eine entsprechende Prozedur *presta* : *tree list* → *tree* → *int* → *tree*, sodass *prest* [*t*] *n* = *presta nil t n* für alle Bäume *t* und alle Pränummern *n* von *t* gilt.

7.6.2 Teilbaumzugriff mit Postnummern

Sie ahnen es schon: Wir wollen jetzt eine Prozedur *post* schreiben, die zu einem Baum den durch eine **Postnummer** identifizierten Teilbaum liefert. Wie beim Teilbaumzugriff mit Pränummern arbeiten wir mit einer Liste von noch zu bearbeitenden Teilbäumen, die wir im Folgenden als Agenda bezeichnen werden. Allerdings kann ein Teilbaum t jetzt auf zwei Arten in die Agenda eingetragen werden: Entweder muss er noch komplett bearbeitet werden (Eintrag als $I\ t$), oder aber seine Unterbäume befinden sich bereits vor ihm auf der Agenda (Eintrag als $F\ t$):

$$\begin{aligned} \mathit{post}\ (F\ t :: es)\ 0 &= t \\ \mathit{post}\ (F\ t :: es)\ k &= \mathit{post}\ es\ (k-1) \qquad \text{für } k > 0 \\ \mathit{post}\ (I(T[t_1,\ldots,t_n]) :: es)\ k &= \mathit{post}\ ([I\ t_1,\ldots,I\ t_n,\ F(T[t_1,\ldots,t_n])]\ @\ es)\ k \end{aligned}$$

Die Terminierung dieser Rekursionsgleichungen ergibt sich aus der Tatsache, dass die Agenda bei jedem Rekursionsschritt kleiner wird. Unter der Größe der Agenda verstehen wir dabei die Summe der Größen der Einträge, wobei ein Eintrag $I\ t$ mit der doppelten Größe von t und ein Eintrag $F\ t$ mit der Größe 1 eingeht.

Aufgabe 7.32 Schreiben Sie eine Prozedur $\mathit{post}' : \mathit{tree} \to \mathit{int} \to \mathit{tree}$, die zu einem Baum und einer Postnummer den entsprechenden Teilbaum liefert. Realisieren Sie die Agenda von *post* mit der folgenden Typdeklaration:

```
datatype 'a entry = I of 'a | F of 'a
```

7.6.3 Linearisierungen

Bäume lassen sich durch Listen über $\mathbb{N}$ darstellen. Dabei wird jeder Knoten durch seine Stelligkeit dargestellt. Bei der **Prälinearisierung**

```
fun pre (T ts) = length ts :: List.concat (map pre ts)
val pre : tree → int list
```

werden die Stelligkeiten der Knoten gemäß der Präordnung angeordnet:

$$\begin{aligned} T[] &\rightsquigarrow [0] \\ T[T[]] &\rightsquigarrow [1,0] \\ T[T[],T[]] &\rightsquigarrow [2,0,0] \\ T[T[],\ T[T[],T[]],\ T[T[]]] &\rightsquigarrow [3,0,2,0,0,1,0] \end{aligned}$$

Bei der **Postlinearisierung**

```
fun post (T ts) = List.concat (map post ts) @ [length ts]
val post : tree → int list
```

werden die Stelligkeiten der Knoten gemäß der Postordnung angeordnet:

$$\begin{array}{rcl} T[] & \rightsquigarrow & [0] \\ T[T[]] & \rightsquigarrow & [0,1] \\ T[T[],T[]] & \rightsquigarrow & [0,0,2] \\ T[T[],\ T[T[],T[]],\ T[T[]]] & \rightsquigarrow & [0,0,0,2,0,1,3] \end{array}$$

Die Postlinearisierung wird manchmal als *polnische Notation* bezeichnet, nach ihrem Entdecker Jan Łukasiewicz.

Beide Linearisierungen haben die Eigenschaft, dass sich ein Baum aus seiner Linearisierung rekonstruieren lässt. Die Rekonstruktionsalgorithmen werden wir später im Zusammenhang mit praktischen Anwendungen kennenlernen (Aufgabe 13.7 auf S. 266 und Aufgabe 16.6 auf S. 330).

Aufgabe 7.33 Geben Sie die Prä- und die Postlinearisierung des Baums $T[T[],\ T[T[]],\ T[T[],T[]]]$ an.

Aufgabe 7.34 Gibt es Listen über $\mathbb{N}$, die gemäß der Prä- oder Postlinearisierung keine Bäume darstellen?

7.7 Balanciertheit

Wir bezeichnen einen Baum als **balanciert**, wenn die Adressen seiner Blätter alle die gleiche Länge haben. Anschaulich gesprochen bedeutet das, dass alle Blätter den gleichen Abstand von der Wurzel haben. Beispielsweise sind die Bäume $t1$ und $t2$ in Abbildung 7.1 auf S. 132 balanciert, während $t3$ nicht balanciert ist.

Um die Balanciertheit eines Baums zu testen, ist die folgende Charakterisierung nützlich.

Proposition 7.4 *Ein Baum ist genau dann balanciert, wenn seine Unterbäume alle balanciert sind und alle die gleiche Tiefe haben.*

Um die Balanciertheit eines Baums zu testen, können wir also wie folgt vorgehen: Wir berechnen die Tiefe aller Unterbäume und testen gleichzeitig, dass alle Unterbäume balanciert sind. Falls ein Unterbaum nicht balanciert ist oder eine von den anderen Unterbäumen abweichende Tiefe hat, werfen wir eine Ausnahme. Damit ergibt sich eine rekursive Prozedur, die sich an der Struktur der Prozedur *depth* orientiert.

Wir beginnen mit einer Variante der Prozedur *depth* aus § 7.4

```
fun depth' (T nil) = 0
  | depth' (T(t::tr)) = 1+foldl Int.max (depth' t) (map depth' tr)
val depth' : tree → int
```

die getrennte Regeln für atomare und zusammengesetzte Bäume verwendet. Die Regel für zusammengesetzte Bäume ändern wir so ab, dass sie zusätzlich prüft, ob alle Unterbäume dieselbe Tiefe haben:

```
exception Unbalanced

fun check (n,m) = if n=m then n else raise Unbalanced

fun depthb (T nil) = 0
  | depthb (T(t::tr)) = 1+foldl check (depthb t) (map depthb tr)
```
val depthb : tree → int

Wenn die durch *check* geprüfte Eigenschaft für alle Teilbäume eines Baums erfüllt ist, ist der Gesamtbaum balanciert. Wenn diese Eigenschaft jedoch für einen der Teilbäume verletzt ist, ist dieser Teilbaum und damit der Gesamtbaum unbalanciert. Die Prozedur *depthb* terminiert also genau dann regulär, wenn der Baum balanciert ist. In diesem Fall liefert sie die Tiefe des Baums als Ergebnis.

Durch Fangen der Ausnahme *Unbalanced* erhalten wir aus *depthb* eine Prozedur, die wie gewünscht testet, ob ein Baum balanciert ist:

```
fun balanced t = (depthb t ; true) handle Unbalanced => false
```
val balanced : tree → bool

Aufgabe 7.35 Deklarieren Sie die Prozedur *balanced* mithilfe eines Let-Ausdrucks so, dass die Ausnahme *Unbalanced* und die Prozeduren *check* und *depthb* lokal im Rumpf der Prozedur *balanced* deklariert werden.

7.8 Finitäre Mengen und gerichtete Bäume

Eine Menge (§ 5.5) heißt **rein**, wenn jedes ihrer Elemente eine reine Menge ist. Die einfachste reine Menge ist die leere Menge. Eine Menge heißt **finitär**, wenn sie endlich ist und jedes ihrer Elemente eine finitäre Menge ist. Die Menge $\{\emptyset, \{\emptyset\}\}$ ist finitär und rein. Die unendliche Menge $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \ldots\}$ ist rein, aber nicht finitär. Offensichtlich ist jede finitäre Menge eine reine Menge.

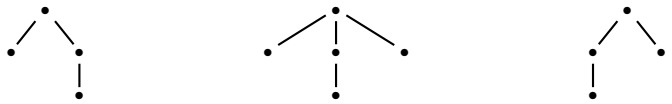
Reine Mengen sind interessanter als sie auf den ersten Blick erscheinen mögen: Alle mathematischen Objekte lassen sich durch reine Mengen darstellen. Bei reinen Mengen handelt es sich also um eine universelle Datenstruktur für die Darstellung mathematischer Objekte.

Wir wollen uns hier auf finitäre Mengen beschränken, deren rekursiver Aufbau dem reiner Bäume entspricht. Allerdings sind bei Mengen keine Listen im Spiel. Daher spielen Anordnung und Mehrfachauftreten der Elemente keine Rolle. Den Zusammenhang zwischen reinen Bäumen und finitären Mengen beschreiben wir durch die Gleichung

$$\mathrm{Set}(T[t_1, \ldots, t_n]) = \{\mathrm{Set}\, t_1, \ldots, \mathrm{Set}\, t_n\}$$

die jedem reinen Baum t per Rekursion eine finitäre Menge Set t zuordnet. Beispielsweise gilt $\mathrm{Set}(T[T[]]) = \{\emptyset\}$. Bei Set t handelt es sich um die Menge, deren Elemente genau die Mengen sind, die durch die Unterbäume von t dargestellt werden.

Machen Sie sich klar, dass jede finitäre Menge durch einen reinen Baum dargestellt werden kann. Erwartungsgemäß ist die Darstellung von finitären Mengen durch Bäume nicht eindeutig. Beispielsweise kann die Menge $\{\emptyset, \{\emptyset\}\}$ durch jeden der folgenden Bäume dargestellt werden:



Wir bekommen eine eindeutige Darstellung für finitäre Mengen, wenn wir uns auf Bäume beschränken, deren Unterbaumlisten strikt sortiert sind (gemäß der lexikalischen Baumordnung (§ 7.1.3); siehe auch § 5.5). Solche Bäume bezeichnen wir als **gerichtet**. Von den oben gezeigten Bäumen ist nur der linke gerichtet. Hier ist eine Prozedur, die testet, ob ein Baum gerichtet ist:

```
fun strict(t::t'::tr) = compareTree(t,t')=LESS andalso strict(t'::tr)
  | strict    _       = true

fun directed (T ts) = strict ts andalso List.all directed ts
```

val directed : tree → bool

Wir wollen einen Algorithmus entwickeln, der entscheidet, ob zwei reine Bäume dieselbe Menge darstellen. Wir beginnen mit den folgenden Äquivalenzen:

1. $\mathrm{Set}\, t_1 = \mathrm{Set}\, t_2$ genau dann, wenn $\mathrm{Set}\, t_1 \subseteq \mathrm{Set}\, t_2$ und $\mathrm{Set}\, t_2 \subseteq \mathrm{Set}\, t_1$.
2. $\mathrm{Set}\, t_1 \subseteq \mathrm{Set}\, t_2$ genau dann, wenn $\mathrm{Set}\, t_1' \in \mathrm{Set}\, t_2$ für jeden Unterbaum t_1' von t_1.
3. $\mathrm{Set}\, t_1 \in \mathrm{Set}\, t_2$ genau dann, wenn $\mathrm{Set}\, t_1 = \mathrm{Set}\, t_2'$ für einen Unterbaum t_2' von t_2.

Damit können wir den Test "$\mathrm{Set}\, t_1 = \mathrm{Set}\, t_2$" auf den Test "$\mathrm{Set}\, t_1 \subseteq \mathrm{Set}\, t_2$" zurückführen, und den Test "$\mathrm{Set}\, t_1 \subseteq \mathrm{Set}\, t_2$" auf den Test "$\mathrm{Set}\, t_1 \in \mathrm{Set}\, t_2$". Den Test "$\mathrm{Set}\, t_1 \in \mathrm{Set}\, t_2$" können wir schließlich wieder auf den Test "$\mathrm{Set}\, t_1 = \mathrm{Set}\, t_2$" zurückführen. Diese **verschränkte Rekursion** terminiert, da bei jedem Durchlauf mindestens einer der Argumentbäume kleiner wird und der andere nicht größer wird. Wir realisieren den Algorithmus durch drei Prozeduren

```
fun eqset x y = subset x y andalso subset y x

and subset (T xs) y = List.all (fn x => member x y) xs

and member x (T ys) = List.exists (eqset x) ys
```

die jeweils den Typ *tree → tree → bool* haben. Bei der Deklaration von *subset* und *member* verwenden wir das Schlüsselwort *and* statt des Schlüsselworts *fun*, um die ver-

schränkte Rekursion zwischen den Prozeduren *eqset*, *subset* und *member* zu ermöglichen.

Aufgabe 7.36 Verschränkte Rekursion kann stets auf einfache Rekursion zurückgeführt werden. Überzeugen Sie sich davon am Beispiel der Prozedur *eqset*, indem Sie eine semantisch äquivalente Prozedur deklarieren, die nur einfache Rekursion verwendet.

Aufgabe 7.37 Deklarieren Sie eine Prozedur $direct : tree \to tree$, die zu einem Baum einen gerichteten Baum liefert, der die gleiche Menge darstellt. Verwenden Sie die polymorphe Sortierprozedur aus Aufgabe 5.15 auf S. 107 und die Vergleichsprozedur *compareTree* aus § 7.1.3.

Aufgabe 7.38 Nehmen Sie an, dass finitäre Mengen durch gerichtete Bäume dargestellt werden.

a) Deklarieren Sie Prozeduren des Typs $tree \to tree \to tree$, die zu zwei Mengen X, Y die Vereinigung $X \cup Y$, den Schnitt $X \cap Y$ und die Differenz $X - Y$ liefern.

b) Deklarieren Sie Prozeduren des Typs $tree \to tree \to bool$, die zu zwei Mengen X, Y testen, ob $X \in Y$ beziehungsweise $X \subseteq Y$ gilt.

Aufgabe 7.39 (Mengendarstellung natürlicher Zahlen) Natürliche Zahlen lassen sich gemäß der Gleichung

$$\mathrm{Set}\, n = \text{if } n = 0 \text{ then } \emptyset \text{ else } \{\mathrm{Set}(n-1)\}$$

eindeutig durch finitäre Mengen darstellen. Beispielsweise gilt $\mathrm{Set}\, 3 = \{\{\{\emptyset\}\}\}$.

a) Schreiben Sie eine Prozedur $code : int \to tree$, die für $n \in \mathbb{N}$ den gerichteten Baum liefert, der die finitäre Menge darstellt, die n darstellt.

b) Schreiben Sie eine Prozedur $decode : tree \to int$, sodass $decode(code\ n) = n$ für alle $n \in \mathbb{N}$ gilt. Für Argumente, die nicht Darstellung einer natürlichen Zahl sind, soll *decode* die Ausnahme *Domain* werfen.

c) Deklarieren Sie zwei Prozeduren *add* und *mul* des Typs $tree \to tree \to tree$, die Darstellungen natürlicher Zahlen addieren und multiplizieren. Verwenden Sie dabei keine Operationen für *int*.

Aufgabe 7.40 (Mengendarstellung von Paaren) Paare lassen sich gemäß der Gleichung $\mathrm{Set}(x, y) = \{\{x\}, \{x, y\}\}$ eindeutig durch Mengen darstellen. Machen Sie sich mit dieser Tatsache vertraut, indem Sie zwei Prozeduren $code : tree * tree \to tree$ und $decode : tree \to tree * tree$ schreiben, für die gilt:

1. Für alle gerichteten Bäume t_1, t_2 gilt: $decode(code(t_1, t_2)) = (t_1, t_2)$.
2. *code* und *decode* liefern zu gerichteten Bäumen gerichtete Bäume.

Wenn *decode* erkennt, dass sein Argument nicht die Codierung eines Paars sein kann, soll es die Ausnahme *Domain* werfen.

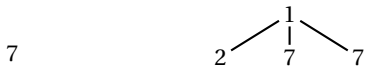
7.9 Markierte Bäume

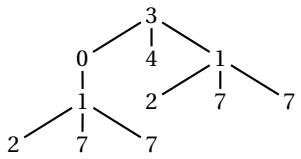
In der Praxis sind die Knoten baumartiger Objekte meist mit Informationen versehen. Beispielsweise tragen die Knoten arithmetischer Ausdrücke Information darüber, ob es sich um eine Addition oder Multiplikation oder eine bestimmte Konstante oder Variable handelt. Wir wollen dieses Phänomen jetzt genauer betrachten. Dazu arbeiten wir mit sogenannten **markierten Bäumen**, bei denen jeder Knoten mit einem Wert aus einem **Grundtyp** versehen ist:

```
datatype 'a ltr = L of 'a * 'a ltr list
```

Ein markierter Baum über einem Typ t besteht also aus einem Wert von t und einer Liste von markierten Bäumen über t. Hier sind Beispiele für Bäume über *int*:

```
val t1 = L(7,[])
val t2 = L(1, [L(2,[]), t1, t1])
val t3 = L(3, [L(0,[t2]), L(4,[]), t2])
```





$t1 = L(7,[])$ $t2 = L(1,[L(2,[]),t1,t1])$ $t3 = L(3,[L(0,[t2]),L(4,[]),t2])$

Bei einem markierten Baum über einem Typ t ist jeder Knoten mit einem Wert aus t versehen, den man als die **Marke** des Knotens bezeichnet. Die Marke der Wurzel eines Baums bezeichnen wir als den **Kopf** des Baums:

```
fun head (L(x,_)) = x
val head : α ltr → α
```

Gemäß Definition ist ein markierter Baum durch seinen Kopf und seine Unterbaumliste bestimmt. Unter der **Gestalt** eines markierten Baums verstehen wir den reinen Baum, den man durch Löschen der Marken erhält:

```
fun shape (L(_,ts)) = T(map shape ts)
val shape : α ltr → tree
```

```
shape t2
T [T [], T [], T []] : tree
```

Über die Gestalt übertragen sich alle Begriffe, die wir für reine Bäume definiert haben, auf markierte Bäume. Hier ist eine Prozedur, die testet, ob zwei markierte Bäume die gleiche Gestalt haben:

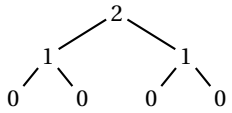
```
fun sameshape t t' = shape t = shape t'
val sameshape : α ltr → β ltr → bool
```

```
sameshape t2 (L(1, [L(2,[]), L(7,[]), L(7,[])]))
```
true : bool

Aufgabe 7.41 Schreiben Sie Prozeduren, die die Größe und die Tiefe von markierten Bäumen bestimmen.

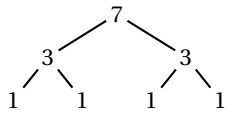
Aufgabe 7.42 Schreiben Sie Prozeduren *leftmost*, *rightmost* : α *ltr* $\to \alpha$, die die Marke des linkesten, beziehungsweise des rechtesten Blatts eines Baums liefern.

Aufgabe 7.43 Schreiben Sie eine Prozedur *ltrd* : *int* $\to$ *int ltr*, die zu $n \geq 0$ einen balancierten Binärbaum der Tiefe n liefert, dessen Teilbäume mit ihrer Tiefe markiert sind. Für $n = 2$ soll *ltrd* den Baum



liefern. Verwenden Sie die Prozedur *iterup*.

Aufgabe 7.44 Schreiben Sie eine Prozedur *ltrs* : *int* $\to$ *int ltr*, die zu $n \geq 0$ einen balancierten Binärbaum der Tiefe n liefert, dessen Teilbäume mit ihrer Größe markiert sind. Für $n = 2$ soll *ltrs* den Baum



liefern. Verwenden Sie die Prozedur *iter*.

Aufgabe 7.45 Schreiben Sie eine Prozedur *sum* : *int ltr* $\to$ *int*, die die Summe der Marken eines Baums liefert. Wenn eine Marke mehrfach auftritt, soll sie auch mehrfach in die Summe eingehen.

Aufgabe 7.46 Schreiben Sie eine Prozedur *lmap* : $(\alpha \to \beta) \to \alpha$ *ltr* $\to \beta$ *ltr*, die eine Prozedur auf alle Marken eines Baums anwendet. Verwenden Sie die Prozedur *map* für Listen.

Aufgabe 7.47 Schreiben Sie eine Prozedur *forall* : $(\alpha \to$ *bool*$) \to \alpha$ *ltr* $\to$ *bool*, die testet, ob eine Prozedur für alle Marken eines Baums *true* liefert.

Aufgabe 7.48 Schreiben Sie eine Prozedur *prestl* : α *ltr* $\to$ *int* $\to \alpha$, die zu einem Baum und einer Pränummer die Marke des durch die Nummer identifizierten Knotens liefert. Orientieren Sie sich an der Prozedur *prest* in § 7.6.1.

Aufgabe 7.49 Schreiben Sie eine Prozedur *find* : $(\alpha \to$ *bool*$) \to \alpha$ *ltr* $\to \alpha$ *option*, die zu einer Prozedur und einem Baum die gemäß der Präordnung erste Marke des Baums liefert, für die die Prozedur *true* liefert. Orientieren Sie sich an der Prozedur *prest* in § 7.6.1.

Aufgabe 7.50 (Spiegeln) Spiegeln reversiert die Ordnung der Unterbäume der Teilbäume eines Baums:

Schreiben Sie eine Prozedur *mirror* : α *ltr* $\rightarrow$ α *ltr*, die markierte Bäume spiegelt.

Aufgabe 7.51 (Lexikalische Ordnung) Deklarieren Sie eine Vergleichsprozedur *compareLtr* : $(\alpha * \alpha \rightarrow order) \rightarrow \alpha\ ltr * \alpha\ ltr \rightarrow order$ für markierte Bäume, die eine Ordnung für die Markierungen mit der lexikalischen Baumordnung kombiniert. Beispielsweise soll *compareLtr Int.compare* (*t1*, *t2*) den Wert *GREATER* liefern ($t1$ und $t2$ sind die oben als Beispiele angegebenen Bäume).

7.10 Projektionen

Unter der **Präprojektion** *prep t* eines markierten Baums t verstehen wir die gemäß der Präordnung geordnete Liste seiner Marken (mit Mehrfachauftreten):

$$prep(L(x,\ [t_1,\ldots,t_n])) = [x]\ @\ prep\ t_1\ @\ \ldots\ @\ prep\ t_n$$

Beispielsweise gilt $prep\ t3 = [3,0,1,2,7,7,4,1,2,7,7]$. Entsprechend verstehen wir unter der **Postprojektion** eines markierten Baums die gemäß der Postordnung geordnete Liste seiner Marken (mit Mehrfachauftreten):

$$pop(L(x,\ [t_1,\ldots,t_n])) = pop\ t_1\ @\ \ldots\ @\ pop\ t_n\ @\ [x]$$

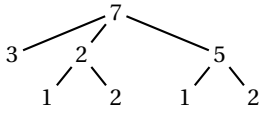
Beispielsweise gilt $pop\ t3 = [2,7,7,1,0,4,2,7,7,1,3]$.

Aufgabe 7.52 Schreiben Sie eine Prozedur *prep* : α *ltr* $\rightarrow$ α *list*, die die Präprojektion eines markierten Baums liefert.

Aufgabe 7.53 Schreiben Sie eine Prozedur *pop* : α *ltr* $\rightarrow$ α *list*, die die Postprojektion eines markierten Baums liefert.

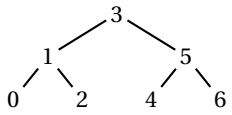
Aufgabe 7.54 Die **Grenze** eines markierten Baums ist die Liste der Marken seiner Blätter, in der Ordnung ihres Auftretens von links nach rechts und mit Mehrfachauftreten. Die Grenze des Baums *t3* ist $[2,7,7,4,2,7,7]$. Schreiben Sie eine Prozedur *frontier* : α *ltr* $\rightarrow$ α *list*, die die Grenze eines Baums liefert.

Aufgabe 7.55 Die **n-te Ebene** eines markierten Baums ist die Liste der Marken der Knoten, die von der Wurzel den Abstand n haben, angeordnet entsprechend der graphischen Anordnung der Knoten. Als Beispiel betrachten wir den Baum



Die nullte Ebene ist [7], die erste Ebene ist [3,2,5], die zweite Ebene ist [1,2,1,2], und die dritte und alle nachfolgenden Ebenen sind []. Schreiben Sie eine Prozedur *level* : α *ltr* $\rightarrow$ *int* $\rightarrow$ α *list*, die die n-te Ebene eines Baums liefert.

Aufgabe 7.56 Bei der **Inprojektion** eines binären Baums erscheinen die Marken der inneren Knoten jeweils nach den Marken des linken und vor den Marken des rechten Unterbaums: $inpro(L(x,[t_1,t_2])) = inpro\ t_1 @ [x] @ inpro\ t_2$. Beispielsweise hat der Baum



die Inprojektion [0,1,2,3,4,5,6]. Schreiben Sie eine polymorphe Prozedur *inpro* : α *ltr* $\rightarrow$ α *list*, die die Inprojektion eines binären Baums liefert. Wenn die Prozedur auf einen Baum angewendet wird, der nicht binär ist, soll die Ausnahme *Domain* geworfen werden.

Bemerkungen

In diesem Kapitel haben wir den Begriff des baumartigen Objektes mithilfe ausführbarer Standardmodelle für reine und markierte Bäume präzise formuliert. Standardmodelle spielen in der Informatik eine wichtige Rolle, da man mit ihnen in allgemeiner Form Begriffe und Algorithmen erarbeitet kann, die sich auf verschiedene Anwendungen übertragen lassen.

Verzeichnis

Reine Bäume: Atomare, zusammengesetzte; Unterbäume und Stelligkeit; lexikalische Baumordnung; Teilbäume, Auftreten und Adressen; lineare, binäre, balancierte; Größe, Tiefe, Grad.

Knoten und Kanten: Wurzel, Blätter, innere Knoten; Nachfolger und Vorgänger, n-ter Nachfolger; übergeordnet und untergeordnet.

Falten und Spiegeln von Bäumen.

Prä- und Postordnung, Prä- und Postnummerierung, Prä- und Postlinearisierung, Standardtour.

Teilbaumzugriff mit Adressen sowie Prä- und Postnummern.

Reine und finitäre Mengen; Darstellung finitärer Mengen durch gerichtete Bäume; Mengendarstellung von natürlichen Zahlen und Paaren.

Verschränkte Rekursion.

Markierte Bäume: Bäume über t; Marken, Kopf, Gestalt; Präprojektion, Postprojektion, Inprojektion; Grenze und Ebenen.

8 Mengenlehre

Eine wichtige Aufgabe der Informatik besteht in der Konstruktion präziser gedanklicher Modelle. Dabei kommen mathematische Datenstrukturen zum Einsatz, wie wir sie im Folgenden behandeln werden: Mengen, Tupel, Graphen, Relationen und Funktionen.

Beginnend mit diesem Kapitel werden wir uns verstärkt der Sprache der Mathematik bedienen. Die mathematische Sprache ist eine der großen Kulturleistungen der Menschheit und gehört zu den Grundlagen der Informatik. Ihre Genauigkeit und ihre Ausdrucksfähigkeit sind für die fachliche Kommunikation unverzichtbar.

8.1 Mengen

Die grundlegende mathematische Datenstruktur sind Mengen. Mengen dienen als universelle Datenstruktur, auf die alle anderen mathematischen Datenstrukturen zurückgeführt werden. Jedes mathematische Objekt lässt sich als Menge darstellen.

Mengen sind gedankliche Objekte. Üblicherweise beschreibt man Mengen als Zusammenfassungen von mathematischen Objekten (§ 5.5). Da sich jedes mathematische Objekt als Menge darstellen lässt, genügt es, nur solche Mengen zu betrachten, deren Elemente wieder Mengen sind (also reine Mengen, § 7.8). Hier sind Beispiele für Mengen, die wir ausgehend von der leeren Menge bilden können:

$$\emptyset,\quad \{\emptyset\},\quad \{\{\emptyset\}\},\quad \{\{\{\emptyset\}\}\},\quad \{\emptyset,\{\emptyset\}\},\quad \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}$$

Die Menge $\{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}$ hat 3 Elemente: die leere Menge $\emptyset$, die einelementige Menge $\{\emptyset\}$, und die zweielementige Menge $\{\emptyset,\{\emptyset\}\}$.

Man kann sich Mengen als reine Bäume vorstellen, wobei die Elemente als Unterbäume erscheinen. Es gibt jedoch drei wesentliche Änderungen: die Elemente einer Menge sind nicht angeordnet, sie können nicht mehrfach auftreten und eine Menge kann unendlich viele Elemente haben.

Eine Menge heißt **finitär**, wenn sie nur endlich viele Elemente hat und jedes ihrer Elemente wieder eine finitäre Menge ist. Finitäre Mengen können durch gerichtete Bäume eindeutig dargestellt werden (§ 7.8).

Die Axiome in Abbildung 8.1 versuchen, Mengen mit möglichst wenigen Begriffen definierend zu beschreiben. Jedes Axiom formuliert eine grundlegende Annahme über Mengen. Die ersten drei Axiome formulieren uns bereits vertraute Eigenschaften von Mengen. Das vierte und das fünfte Axiom verdienen eine genauere Betrachtung.

1. Eine Menge ist eine Zusammenfassung von Mengen. Die von einer Menge x zusammengefassten Mengen bezeichnen wir als die **Elemente von** x. Wir schreiben $x \in y$, um zu sagen, dass die Menge x ein Element der Menge y ist.
2. Es gibt genau eine Menge, die keine Elemente hat. Diese Menge heißt **leere Menge** und wird mit $\emptyset$ bezeichnet.
3. Jede endliche oder unendliche Sammlung von Mengen kann zu einer Menge zusammengefasst werden. Die Menge, die die Mengen $x_1, \ldots, x_n$ zusammenfasst, bezeichnen wir mit $\{x_1, \ldots, x_n\}$.
4. **Gleichheit:** Zwei Mengen x und y sind genau dann gleich ($x = y$), wenn jedes Element von x ein Element von y ist und jedes Element von y ein Element von x ist.
5. **Wohlfundiertheit:** Es gibt keine unendliche Folge x_1, x_2, x_3, $\ldots$ von Mengen, sodass $\ldots \in x_3 \in x_2 \in x_1$ gilt.

Abbildung 8.1: *Axiomatische Beschreibung von Mengen*

Das **Gleichheitsaxiom** (4) besagt, dass eine Menge vollständig durch ihre Elemente festgelegt ist. Es gibt also keine zwei verschiedenen Mengen, die genau die gleichen Elemente haben. Das bedeutet, dass eine Menge im Gegensatz zu einem Tupel keine Ordnung für ihre Elemente festlegt und dass ein Element nicht mehrfach in einer Menge vorkommen kann. Folglich stellen die Beschreibungen

$$\{\emptyset, \{\emptyset\}\} \qquad \{\{\emptyset\}, \emptyset\} \qquad \{\emptyset, \{\emptyset\}, \emptyset\}$$

alle dieselbe Menge dar.

Das **Wohlfundierungsaxiom** (5) besagt, dass man in der Baumdarstellung einer Menge nicht unendlich oft absteigen kann. Das bedeutet, dass man bei einer Menge nach endlich vielen Abstiegen zu einem Element stets die leere Menge erreicht, man also terminiert.

Durch das Wohlfundierungsaxiom wird die durch das Axiom (3) zugesicherte Bildungsmöglichkeit von Mengen eingeschränkt. In einem gewissen Sinn besagt das Wohlfundierungsaxiom, dass eine Menge erst gebildet werden kann, nachdem alle ihre Elemente gebildet sind. Dabei ist das „nachdem“ auf einer unendlichen Zeitskala zu verstehen, da ja auch Mengen mit unendlich vielen Elementen bildbar sein sollen.

Eine Menge heißt **endlich**, wenn sie nur endlich viele Elemente hat. Das einfachste Beispiel für eine **unendliche Menge** ist $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \ldots\}$.

Sei X eine endliche Menge. Die Anzahl der Elemente von X heißt **Kardinalität von** X und wird mit $|X|$ bezeichnet. Beispielsweise gilt $|\emptyset| = 0$ und $|\{0,1\}| = 2$.

Wir geben noch einige gebräuchliche Sprechweisen für Mengen an. Man sagt, dass eine Menge aus ihren Elementen **besteht** oder dass eine Menge ihre Elemente **enthält**. Statt „x ist ein Element von X“ sagt man auch kürzer „x **ist in** X“. Unter einer **einelementigen Menge** versteht man eine Menge, die genau ein Element enthält. Allgemeiner versteht

man unter einer ***n*-elementigen Menge** eine endliche Menge, die genau n verschiedene Elemente hat.

Eine Menge X heißt **Teilmenge** einer Menge Y, wenn jedes Element von X ein Element von Y ist. Wir schreiben $X \subseteq Y$, um zu sagen, dass X eine Teilmenge von Y ist. Statt $X \subseteq Y$ schreiben wir auch $Y \supseteq X$.

Unter einer **echten Teilmenge** einer Menge X verstehen wir eine Teilmenge von X, die verschiedenen von X ist. Wir schreiben $X \subset Y$, um zu sagen, dass X eine echte Teilmenge von Y ist. Statt $X \subset Y$ schreiben wir auch $Y \supset X$.

Seien X und Y Mengen. Dann heißt X **(echte) Obermenge** von Y, wenn Y eine (echte) Teilmenge von X ist.

Zwei Mengen X und Y heißen **disjunkt**, wenn es keine Menge gibt, die sowohl ein Element von X als auch ein Element von Y ist.

Wir schreiben $x \notin X$, um zu sagen, dass x kein Element der Menge X ist, $X \neq Y$ um zu sagen, dass X und Y verschiedene Mengen sind und $X \not\subseteq Y$, um zu sagen, dass X keine Teilmenge von Y ist.

Wir haben bereits gesehen, dass sich die natürlichen Zahlen durch die finitären Mengen $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$, … darstellen lassen (Aufgabe 7.39 auf S. 148). Wir können die Symbole $0, 1, 2, \ldots$ also als Bezeichnungen für die entsprechenden Mengen auffassen. Folglich könnten wir $0 \in 1 \in 2 \in \cdots$ schreiben. Wir wollen von diesem Hintergrundwissen aber keinen Gebrauch machen und die Symbole für Zahlen nur typgerecht verwenden.

Auch die reellen Zahlen können als Mengen dargestellt werden. Dazu muss man allerdings mehr Aufwand betreiben als bei den natürlichen Zahlen. Zunächst muss man die rationalen Zahlen darstellen. Danach kann man die reellen Zahlen durch sogenannte Dedekindsche Schnitte darstellen. Wir vereinbaren die folgenden Bezeichnungen für oft vorkommende Zahlenmengen:

$\mathbb{B}$	$:= \{0, 1\}$	**Boolesche Werte**
$\mathbb{N}$	$:= \{0, 1, 2, \ldots\}$	**Natürliche Zahlen**
$\mathbb{N}_+$	$:= \{1, 2, 3, \ldots\}$	**Positive natürliche Zahlen**
$\mathbb{Z}$	$:= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$	**Ganze Zahlen**
$\mathbb{R}$	:= Menge der reellen Zahlen	**Reelle Zahlen**

Wir gehen davon aus, dass die Darstellungen der Zahlen so gewählt sind, dass $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$ gilt.

Mit der Notation $\{x \mid A\}$ bezeichnen wir die Menge aller Mengen x, die die Eigenschaft A haben. Mithilfe dieser Notation können wir den Schnitt, die Vereinigung und die Differenz zweier Mengen X, Y wie folgt definieren:

$X \cap Y := \{z \mid z \in X \text{ und } z \in Y\}$	**Schnitt**
$X \cup Y := \{z \mid z \in X \text{ oder } z \in Y\}$	**Vereinigung**
$X - Y := \{z \mid z \in X \text{ und } z \notin Y\}$	**Differenz**

Erfinden bedeutet Auswählen

Mit der Klasse der Mengen haben wir einen fixen Vorrat mathematischer Objekte, der alle jemals zu betrachtenden mathematischen Objekte bereits enthält. Es geht also nicht darum, völlig neue mathematische Objekte zu erfinden. Stattdessen genügt es, aus der Klasse der Mengen interessante Teilklassen auszuwählen. Eine ähnliche Situation ergibt sich bei der sogenannten *Binärcodierung*, deren Existenz Computer ihre multimedialen Fähigkeiten verdanken. Mithilfe der Binärcodierung kann jeder Roman, jedes Bild und jedes Musikstück durch eine Folge von Nullen und Einsen dargestellt werden. Das bedeutet, dass es alle Romane, Bilder und Musikstücke, die es jemals geben wird, bereits gibt. Autoren neuer Kunstwerke wählen also lediglich bisher unbekannte Werke aus dem bereits bestehenden Vorrat aller Binärcodierungen aus.

Hier sind Beispiele:

$$\{1,2,3,4\} \cap \{3,4,5\} = \{3,4\}$$
$$\{1,2,3,4\} \cup \{3,4,5\} = \{1,2,3,4,5\}$$
$$\{1,2,3,4\} - \{3,4,5\} = \{1,2\}$$

Die **Potenzmenge** einer Menge X ist die Menge aller Teilmengen von X:

$$\mathscr{P}(X) := \{ Y \mid Y \subseteq X \}$$

Hier ist ein Beispiel für eine Potenzmenge:

$$\mathscr{P}(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$$

Aufgabe 8.1 Welches Axiom besagt, dass es keine Menge x gibt, die sich selbst enthält (d.h. $x \in x$)? Warum folgt daraus, dass es keine Menge gibt, die alle Mengen enthält?

Aufgabe 8.2 Geben Sie eine Menge an, die genau 7 echte Teilmengen hat.

Aufgabe 8.3 Sei X eine Menge. Geben Sie eine echte Obermenge von X an.

Aufgabe 8.4 In den Aufgaben 7.39 und 7.40 auf S. 148 haben wir Mengendarstellungen für die natürlichen Zahlen und für Paare angegeben. Geben Sie gemäß dieser Darstellungen die Menge an, die das Tupel $\langle 2,1 \rangle$ darstellt.

8.2 Aussagen

Unter einer **Aussage** verstehen wir einen Ausdruck, der eine Eigenschaft mathematischer Objekte formuliert. Eine Aussage hat entweder den Wert 0 oder 1. Wenn eine Aussage den Wert 1 hat, sagen wir, dass die Aussage **gültig** oder **wahr** ist. Wenn eine Aussage

den Wert 0 hat, sagen wir, dass die Aussage **ungültig** oder **falsch** ist. Mit den **Booleschen Operatoren** können Aussagen negiert oder verknüpft werden:

$\neg A$	*nicht A*	**Negation**
$A \wedge B$	*A und B*	**Konjunktion**
$A \vee B$	*A oder B*	**Disjunktion**
$A \Rightarrow B$	*wenn A, dann B*	**Implikation**
$A \Leftrightarrow B$	*A genau dann, wenn B*	**Äquivalenz**

Die Semantik der Booleschen Operatoren ist wie folgt definiert:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Beachten Sie die Semantik von Disjunktion und Implikation. In der Umgangssprache werden „oder“ und „wenn dann“ manchmal mit anderer Bedeutung verwendet. Dagegen verwenden die mathematische Sprache „oder“ und „wenn dann“ nur mit der oben definierten Bedeutung.

Da Aussagen Ausdrücke sind, kann man Gleichungen zwischen Aussagen betrachten. Beispielsweise gilt für alle Aussagen A, B:

$$A \Rightarrow B \;=\; \neg A \vee B \;=\; \neg B \Rightarrow \neg A$$

Da Äquivalenz auf Gleichheit testet, ist die Gültigkeit dieser Gleichungskette gleichbedeutend mit der Gültigkeit der Aussage

$$A \Rightarrow B \iff \neg A \vee B \iff \neg B \Rightarrow \neg A$$

Aussagen können auch durch Quantifizierung gebildet werden:

$\forall x \in X\colon A$	*für alle $x \in X$ gilt A*	**Universelle Quantifizierung**
$\exists x \in X\colon A$	*es existiert $x \in X$, sodass A gilt*	**Existentielle Quantifizierung**

Dabei spielt x die Rolle einer quantifizierten Variable. Die Semantik der Quantifizierungen ist wie folgt definiert:

- $(\forall x \in X\colon A) = 1$ genau dann, wenn $A = 1$ für alle $x \in X$ gilt.
- $(\exists x \in X\colon A) = 1$ genau dann, wenn $A = 1$ für mindestens ein $x \in X$ gilt.

Für Quantifizierungen gelten unter anderem die folgenden Gleichungen:

$$\begin{aligned}
\neg(\forall x \in X\colon A) &= (\exists x \in X\colon \neg A)\\
\neg(\exists x \in X\colon A) &= (\forall x \in X\colon \neg A)\\
(\forall x \in \emptyset\colon A) &= 1\\
(\exists x \in \emptyset\colon A) &= 0
\end{aligned}$$

Mithilfe der eingeführten Konstrukte für Aussagen kann das Gleichheitsaxiom für Mengen wie folgt formuliert werden:

$$\text{Für alle Mengen } X, Y\colon \quad X = Y \iff (\forall x \in X\colon x \in Y) \wedge (\forall y \in Y\colon y \in X)$$

Hier ist ein Beispiel für eine gültige Aussage über die natürlichen Zahlen:

$$\forall n \in \mathbb{N}\colon\ n = 0 \ \vee\ \exists m \in \mathbb{N}\colon\ m < n$$

Neue Notationen, die sich mit einer Gleichung definieren lassen, führen wir, wie bereits mehrfach geschehen, mithilfe des Symbols := ein. Wenn es sich um Notationen für Aussagen handelt, werden wir stattdessen das Symbol :$\iff$ verwenden.

8.3 Tupel

Wir haben bereits gezeigt, wie man natürliche Zahlen und Paare als Mengen darstellen kann (Aufgaben 7.39 und 7.40 auf S. 148). Darauf aufbauend ist es einfach, Tupel beliebiger Länge als Mengen darzustellen:

$$\langle x_1, \ldots, x_n\rangle := \{(1, x_1), \ldots, (n, x_n)\} \qquad \text{für } n \geq 0$$

Hier sind Beispiele:

$$\begin{aligned}
\langle\rangle &= \emptyset && \textbf{leeres Tupel}\\
\langle x\rangle &= \{(1, x)\}\\
\langle x, y\rangle &= \{(1, x), (2, y)\}\\
\langle x, y, z\rangle &= \{(1, x), (2, y), (3, z)\}
\end{aligned}$$

Die Länge, die Komponenten und die Positionen von Tupeln definieren wir wie folgt:

$$\begin{aligned}
|\langle x_1, \ldots, x_n\rangle| &:= n && \textbf{Länge}\\
\mathit{Com}\,\langle x_1, \ldots, x_n\rangle &:= \{x_1, \ldots, x_n\} && \textbf{Komponenten}\\
\mathit{Pos}\,\langle x_1, \ldots, x_n\rangle &:= \{1, \ldots, n\} && \textbf{Positionen}
\end{aligned}$$

Tupel der Länge n bezeichnen wir als n-**stellige Tupel**. Wenn t ein Tupel mit $(i, x) \in t$ ist, bezeichnen wir x als die i-**te Komponente** von t.

Unter einem **Paar** wollen wir im Folgenden ein Tupel mit 2 Positionen verstehen, unter einem **Tripel** ein Tupel mit 3 Positionen. Die für die Darstellung von Tupeln verwendete Darstellung von Paaren dient also nur als Hilfskonstruktion. Tupel $\langle x_1, \ldots, x_n \rangle$ mit mindestens zwei Positionen schreiben wir auch mit runden Klammern: $(x_1, \ldots, x_n)$.

Sei X eine Menge. Die Menge X^* aller **Tupel über** X ist wie folgt definiert:

$$X^* := \{ t \mid t \text{ Tupel mit } \mathit{Com}\, t \subseteq X \}$$

Es gilt $\emptyset^* = \{\langle\rangle\}$.

Das **Produkt** von $n \geq 0$ Mengen $X_1, \ldots, X_n$ definieren wir wie folgt:

$$\times\langle X_1, \ldots, X_n \rangle := \{ \langle x_1, \ldots, x_n \rangle \mid x_1 \in X_1, \ldots, x_n \in X_n \} \qquad \text{für } n \geq 0$$

Die Elemente des Produkts sind n-stellige Tupel, deren i-te Komponente ein Element des **i-ten Faktors** X_i ist (für $i = 1, \ldots, n$). Für $n \geq 2$ verwenden wir die Notation

$$X_1 \times \cdots \times X_n := \times\langle X_1, \ldots, X_n \rangle \qquad \text{für } n \geq 2$$

Ein Produkt aus $n \geq 0$ gleichen Faktoren X bezeichnen wir mit X^n:

$$X^n := \{ t \in X^* \mid |t| = n \} \qquad \text{für } n \geq 0$$

Hier sind Beispiele:

$$\{1,2\} \times \{3,4\} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 2,4\rangle\}$$
$$\mathbb{B}^2 = \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$$
$$\mathbb{N}^0 = \{\langle\rangle\}$$

Beachten Sie, dass es sich bei $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$ und $\mathbb{N}\times(\mathbb{N}\times\mathbb{N})$ um verschiedene Mengen handelt: Während die erste Menge nur Tripel enthält, enthält die zweite Menge nur Paare.

Mit Tupeln können wir Listen und Bäume darstellen. Sei X eine Menge. Wir folgen den rekursiven Darstellungen aus § 4.1, § 7.1 und § 7.9:

$$\mathscr{L}(X) := \{\langle\rangle\} \cup (X \times \mathscr{L}(X)) \qquad \textbf{Listen über } X$$
$$\mathscr{T} := \mathscr{L}(\mathscr{T}) \qquad \textbf{Reine Bäume}$$
$$\mathscr{T}(X) := X \times \mathscr{L}(\mathscr{T}(X)) \qquad \textbf{Bäume über } X$$

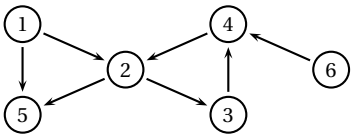
Die in § 4.1 eingeführten Notationen *nil*, $x :: xs$ und $[x_1, \ldots, x_n]$ werden wir auch für mit Mengen dargestellte Listen verwenden.

Konstruktortypen in Standard ML entsprechen Mengen, die als sogenannte Summen gebildet werden. Die **Summe** von $n \geq 0$ Mengen $X_1, \ldots, X_n$ definieren wir wie folgt:

$$+\langle X_1, \ldots, X_n \rangle := \{ \langle i, x \rangle \mid i \in \{1, \ldots, n\} \text{ und } x \in X_i \} \qquad \text{für } n \geq 0$$
$$X_1 + \cdots + X_n := +\langle X_1, \ldots, X_n \rangle \qquad \text{für } n \geq 2$$

Die Elemente der Summe sind Paare $\langle i, x \rangle$, die aus einer **Variantennummer** $i \in \mathbb{N}$ und einem Element x des **i-ten Summanden** X_i bestehen. Hier ist ein Beispiel:

$$\{3,4\} + \{5,6\} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,5\rangle, \langle 2,6\rangle\}$$



$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1,2), (1,5), (2,3), (2,5), (3,4), (4,2), (6,4)\}$

Abbildung 8.2: *Ein Graph (V,E) mit seiner grafischen Darstellung*

Aufgabe 8.5 Geben Sie die Elemente der Menge $\mathbb{N}^* \cap \mathbb{B}^2 \cap (\mathbb{B} + \mathbb{B})$ an.

Aufgabe 8.6 Geben Sie die Menge an, die die Liste [1] darstellt.

8.4 Gerichtete Graphen

Die mathematische Datenstruktur der gerichteten Graphen hat in der Informatik besonders viele Anwendungen. Gerichtete Graphen haben die nützliche Eigenschaft, dass sie sich sehr anschaulich grafisch darstellen lassen. Wir beginnen mit der mathematischen Definition gerichteter Graphen.

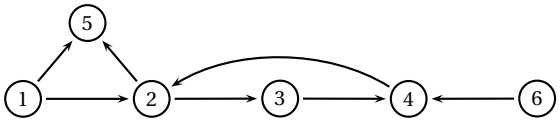
Ein **gerichteter Graph** ist ein Paar (V,E), das aus einer Menge V und einer Menge $E \subseteq V \times V$ besteht. Die Elemente von V werden als die **Knoten** und die Elemente von E als die **Kanten** des Graphen bezeichnet. Wir sagen, dass eine Kante (v,w) **von** v **nach** w **führt**.

Da wir nur gerichtete Graphen betrachten, bezeichnen wir sie der Kürze halber einfach als Graphen. Abbildung 8.2 zeigt ein Beispiel für einen Graphen und seine grafische Darstellung. Die grafische Darstellung eines Graphen besteht aus der grafischen Darstellung seiner Knoten durch kleine Kreise und seiner Kanten durch Pfeile. Dabei erscheint eine Kante (v,v') als ein Pfeil, der vom Knoten v zum Knoten v' führt.

Die Wahl der Buchstaben V und E für die Mengen der Knoten und Kanten erklärt sich aus der Tatsache, dass Knoten im Englischen als *vertices* (singular *vertex*) und Kanten als *edges* bezeichnet werden.

Graphen tauchen bei der Analyse vieler praktischer Situationen auf. Ein typisches Beispiel sind Webseiten im Internet: Die Seiten können als Knoten modelliert werden und die Verweise von einer Seite auf eine andere als Kanten. So gesehen besagt der Graph in Abbildung 8.2 unter anderem, dass die Seite 1 auf die Seiten 2 und 5 verweist und dass keine der dargestellten Seiten auf die Seite 1 verweist.

Bei der grafischen Darstellung von Graphen kann die Lage der Knoten frei gewählt werden. Man spricht vom **Layout eines Graphen**. Hier ist ein alternatives Layout des Graphen in Abbildung 8.2:



Im Rest dieses Abschnitts führen wir die wichtigsten Sprechweisen für Graphen ein. Machen Sie sich für jede Sprechweise klar, wie sie mathematisch definiert ist und was sie anschaulich bedeutet. Es ist dieses für die Informatik typische Wechselspiel zwischen mathematischer Definition und anschaulicher Bedeutung, auf das es im Folgenden ankommt.

Sei $G = (V, E)$ ein Graph.

- Ein Knoten w heißt **Nachfolger** eines Knotens v, wenn $(v, w) \in E$.
- Ein Knoten v heißt **Vorgänger** eines Knotens w, wenn $(v, w) \in E$.
- Zwei Knoten v und w heißen **benachbart** oder **adjazent**, wenn $(v, w) \in E$ oder $(w, v) \in E$.
- Ein **Pfad** ist ein nichtleeres Tupel $\langle v_1, \ldots, v_n \rangle$, sodass für alle $i \in \{1, \ldots, n-1\}$ gilt: $(v_i, v_{i+1}) \in E$. Dabei wird $n-1$ als die **Länge**, v_1 als der **Ausgangspunkt** und v_n als der **Endpunkt** des Pfades bezeichnet. Wir sagen auch, dass der Pfad **von** v_1 **nach** v_n **führt**.
- Ein Pfad heißt **einfach**, wenn er keinen Knoten mehrfach enthält.
- Ein Pfad $\langle v_1, \ldots, v_n \rangle$ heißt **Zyklus**, wenn $n \geq 2$, $v_1 = v_n$ und $\langle v_1, \ldots, v_{n-1} \rangle$ einfach ist.
- Ein Knoten w ist von einem Knoten v aus **erreichbar**, wenn ein Pfad von v nach w existiert.
- Ein Knoten heißt **Wurzel**, wenn von ihm aus alle Knoten erreichbar sind.
- Ein Knoten heißt **Quelle** oder **initial**, wenn er keinen Vorgänger hat.
- Ein Knoten heißt **Senke** oder **terminal**, wenn er keinen Nachfolger hat.
- Ein Knoten heißt **isoliert**, wenn er initial und terminal ist.

Für den Graphen in Abbildung 8.2 gilt:

- Vom Knoten 2 aus sind die Knoten 2, 5, 3 und 4 erreichbar.
- Vom Knoten 1 aus sind alle Knoten bis auf 6 erreichbar.
- Der Pfad $\langle 2, 3, 4, 2 \rangle$ ist ein Zyklus.
- Der Knoten 5 ist terminal und die Knoten 1 und 6 sind initial.
- Es gibt keine Wurzel und keinen isolierten Knoten.

Ein Graph heißt

- **endlich**, wenn er nur endlich viele Knoten hat.
- **symmetrisch**, wenn er für jede Kante (v, w) auch die Kante (w, v) hat.
- **gewurzelt**, wenn er eine Wurzel hat.
- **zyklisch**, wenn er einen Zyklus enthält.

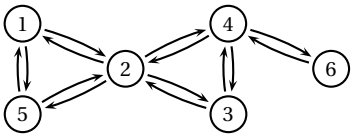
- **azyklisch**, wenn er keinen Zyklus enthält.

Der Graph in Abbildung 8.2 ist zyklisch und endlich und weder gewurzelt noch symmetrisch.

Die **Tiefe** eines endlichen Graphen mit mindestens einem Knoten ist die maximale Länge seiner einfachen Pfade. Der Graph in Abbildung 8.2 hat die Tiefe 3.

Man kann sich einen Graphen als ein Spielfeld vorstellen, auf dem Spielfiguren entlang der Kanten bewegt werden können (nur in Richtung der Kanten; denken Sie an das Spiel „Mensch Ärgere Dich nicht“). Pfade stellen dann mögliche Zugfolgen für Figuren dar. Aus der **Spielsicht** betrachtet sind endliche Graphen ohne Zyklen eher langweilig, da eine Figur nach einigen Schritten immer auf einem terminalen Knoten landet, den sie nicht mehr verlassen kann. Dagegen erlauben Graphen mit Zyklen interessante Spiele, da sie den Figuren eine Chance geben, terminale Knoten zu vermeiden.

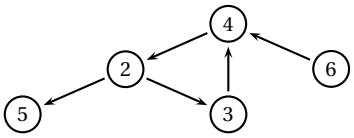
Den **symmetrischen Abschluss** eines Graphen erhält man, indem man für jede Kante (v, v') die **inverse Kante** (v', v) hinzufügt. Hier ist der symmetrische Abschluss des Graphen in Abbildung 8.2:



Der symmetrische Abschluss eines Graphen ist immer symmetrisch.

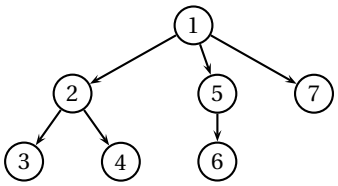
Ein Graph heißt **stark zusammenhängend**, wenn jeder seiner Knoten von jedem seiner Knoten aus erreichbar ist. Ein Graph heißt **zusammenhängend**, wenn sein symmetrischer Abschluss stark zusammenhängend ist. Der Graph in Abbildung 8.2 ist zusammenhängend, aber nicht stark zusammenhängend.

Ein Graph $G = (V, E)$ heißt **Teilgraph** eines Graphen $G' = (V', E')$, wenn $V \subseteq V'$ und $E \subseteq E'$ gilt. Sei $G = (V, E)$ ein Graph und $v \in V$. Der **von v aus erreichbare Teilgraph** von G besteht aus allen Knoten, die von v aus erreichbar sind, und aus allen Kanten zwischen diesen Knoten. Für den Graphen in Abbildung 8.2 sieht der vom Knoten 6 aus erreichbare Teilgraph wie folgt aus:



Der von einem Knoten v aus erreichbare Teilgraph ist zusammenhängend und hat v als Wurzel.

Ein Graph heißt **baumartig**, wenn er eine Wurzel ohne Vorgänger hat und wenn seine anderen Knoten alle genau einen Vorgänger haben. Hier ist ein Beispiel für einen baumartigen Graphen:



Ein baumartiger Graph hat immer genau eine Quelle, die gleichzeitig auch die eindeutig bestimmte Wurzel des Graphen ist. Außerdem haben baumartige Graphen die charakteristische Eigenschaft, dass es von der Wurzel zu einem Knoten immer genau einen Pfad gibt.

Ein wichtiger Unterschied zwischen baumartigen Graphen und Bäumen gemäß § 8.3 besteht darin, dass baumartige Graphen keine Ordnung für die Unterbäume vorgeben. In der obigen Darstellung eines baumartigen Graphen ist die Reihenfolge der Unterbäume also frei gewählt.

Aufgabe 8.7 Sei ein Graph $G = (V, E)$ wie folgt gegeben:

$V = \{1,2,3,4,5,6\}$

$E = \{(1,5),(2,1),(2,3),(2,4),(3,5),(6,2),(6,3)\}$

a) Zeichnen Sie den Graphen ohne überkreuzende Kanten.
b) Welche Tiefe hat der Graph?
c) Welche Quellen, Senken und Wurzeln hat der Graph?
d) Ist der Graph zyklisch? Wenn ja, geben Sie einen Zyklus an.
e) Geben Sie einen einfachen Pfad maximaler Länge an.
f) Ist der Graph zusammenhängend? Stark zusammenhängend?
g) Zeichnen Sie den vom Knoten 2 aus erreichbaren Teilgraphen.

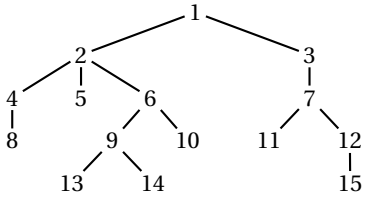
Aufgabe 8.8 Zeichnen Sie zusammenhängende Graphen wie folgt:

a) Einen Graphen mit 3 Knoten, sodass jeder Knoten eine Wurzel ist.
b) Einen Graphen mit 3 Knoten, der eine Quelle und zwei Senken hat.
c) Einen Graphen mit 4 Knoten, der zwei Quellen und zwei Senken hat.

Aufgabe 8.9

a) Gibt es einen stark zusammenhängenden Graphen mit einer Quelle?
b) Gibt es einen stark zusammenhängenden Graphen mit zwei Quellen?

Aufgabe 8.10 Sei der folgende baumartige Graph gegeben:



Dabei seien alle Kanten von oben nach unten gerichtet.

a) Geben Sie die Knotenmenge V und die Kantenmenge E des Graphen an.
b) Geben Sie die Tiefe des Graphen an.
c) Geben Sie die Wurzel des Graphen an.
d) Geben Sie die terminalen Knoten des Graphen an.
e) Geben Sie für den Knoten 14 einen Pfad an, der von der Wurzel zu diesem Knoten führt. Gibt es mehrere solcher Pfade?
f) Zeichnen Sie alle Teilgraphen, die den Knoten 7 als Wurzel haben.
g) Geben Sie einen Teilgraphen an, der nicht baumartig ist.

8.5 Binäre Relationen

Wir kommen jetzt zum nächsten Klassiker unter den mathematischen Datenstrukturen, den sogenannten binären Relationen. Wie sich gleich herausstellen wird, handelt es sich dabei eigentlich um spezielle Graphen. Wir beginnen mit der mathematischen Definition von binären Relationen, die noch einfacher ist als die von Graphen.

Eine **binäre Relation** ist eine Menge R, sodass jedes Element von R ein Paar ist.

Da wir nur binäre Relationen betrachten, bezeichnen wir sie der Kürze halber einfach als Relationen. Für eine Relation R definieren wir die folgenden Mengen:

$$\begin{aligned} \mathit{Dom}\, R &:= \{x \mid \exists y\colon (x,y) \in R\} && \textbf{Definitionsbereich} \\ \mathit{Ran}\, R &:= \{y \mid \exists x\colon (x,y) \in R\} && \textbf{Wertebereich} \\ \mathit{Ver}\, R &:= \mathit{Dom}\, R \cup \mathit{Ran}\, R && \textbf{Knotenmenge} \end{aligned}$$

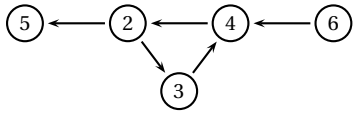
Die Kürzel *Dom*, *Ran* und *Ver* sind aus den englischen Bezeichnungen *domain*, *range* und *vertex* abgeleitet. Die Elemente von $\mathit{Ver}\, R$ bezeichnen wir als die **Knoten von** R. Wir sagen, dass eine Relation R

- für x **definiert** ist, wenn $x \in \mathit{Dom}\, R$.
- x den Wert y **zuordnet**, wenn $(x,y) \in R$.

Abbildung 8.3 zeigt ein Beispiel für eine Relation und ihre grafische Darstellung. Machen Sie sich an diesem Beispiel die Bedeutung der Begriffe Definitionsbereich, Wertebereich und Knotenmenge klar.

Sei R eine Relation. Dann bezeichnen wir $(\mathit{Ver}\, R, R)$ als den **zu** R **gehörigen Graphen**. Die grafische Darstellung einer Relation ergibt sich gerade als die grafische Darstellung des zu R gehörigen Graphen. Machen Sie sich klar, dass die zu Relationen gehörigen Graphen genau die Graphen sind, die keine isolierten Knoten haben.

Da wir Relationen als Graphen auffassen können, lassen sich die Begriffe für Graphen auf Relationen übertragen. Wir sprechen von der **Graphsicht** einer Relation.



$R = \{(2,3), (2,5), (3,4), (4,2), (6,4)\}$

$Dom\,R = \{2, 3, 4, 6\}$ $Ran\,R = \{2, 3, 4, 5\}$ $Ver\,R = \{2, 3, 4, 5, 6\}$

Abbildung 8.3: *Eine Relation R mit ihrer grafischen Darstellung*

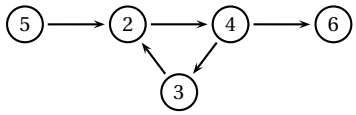
Unter einer **Relation auf einer Menge** X verstehen wir eine Relation R mit $Ver\,R \subseteq X$. Die Relationen auf einer Menge X sind genau die Teilmengen von $X \times X$. Beachten Sie, dass die leere Menge eine Relation auf jeder Menge X ist.

8.5.1 Umkehrrelationen

Die **Umkehrrelation** R^{-1} einer Relation R erhält man, indem man alle Paare von R umkehrt:

$$R^{-1} := \{(x, y) \mid (y, x) \in R\}$$

Hier ist die Graphdarstellung der Umkehrrelation der Relation aus Abbildung 8.3:



Statt Umkehrrelation sagt man auch **inverse Relation**. Offensichtlich gilt $(R^{-1})^{-1} = R$, $Dom\,(R^{-1}) = Ran\,R$ und $Ver\,(R^{-1}) = Ver\,R$.

8.5.2 Funktionale und injektive Relationen

Eine Relation R heißt **funktional**, wenn zu jedem $x \in Dom\,R$ genau ein $y \in Ran\,R$ existiert mit $(x, y) \in R$, und **injektiv**, wenn zu jedem $y \in Ran\,R$ genau ein $x \in Dom\,R$ existiert mit $(x, y) \in R$. Funktionalität und Injektivität sind symmetrische Begriffe: Eine Relation ist genau dann funktional, wenn ihre Umkehrrelation injektiv ist.

Aus der Graphsicht ist eine Relation genau dann funktional, wenn von keinem Knoten mehr als eine Kante ausgeht, und injektiv genau dann, wenn auf keinen Knoten mehr als eine Kante zeigt. Mit diesem Wissen kann man sofort sehen, dass die Relation in Abbildung 8.3 weder funktional noch injektiv ist.

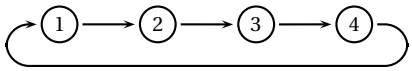
Die Ein-Ausgabe-Relation einer Prozedur ist genau dann funktional, wenn es zu einem Argument höchstens ein Ergebnis gibt, und injektiv genau dann, wenn es für ein Ergebnis höchstens ein Argument gibt. Bisher haben wir nur Prozeduren mit funktionalen Ein-Ausgabe-Relationen betrachtet.

8.5.3 Totale und surjektive Relationen

Sei R eine Relation und X eine Menge. Dann heißt R **total auf** X, wenn $X \subseteq \mathit{Dom}\, R$, und **surjektiv auf** X, wenn $X \subseteq \mathit{Ran}\, R$.

Aus der Graphsicht ist eine Relation genau dann total auf X, wenn von jedem Element von X eine Kante ausgeht, und surjektiv auf X genau dann, wenn auf jedes Element von X eine Kante zeigt. Totalität und Surjektivität sind symmetrische Begriffe: Eine Relation ist genau dann surjektiv auf X, wenn ihre Umkehrrelation total auf X ist. Jede Relation R ist total auf $\mathit{Dom}\, R$ und surjektiv auf $\mathit{Ran}\, R$.

Hier ist die Graphdarstellung einer funktionalen und injektiven Relation, die total und surjektiv auf $\{1,2,3,4\}$ ist:



Aufgabe 8.11 Sei die Relation $R = \{(1,2),\ (1,3),\ (2,4),\ (4,1),\ (5,2)\}$ gegeben.

a) Zeichnen Sie die Graphdarstellungen von R und R^{-1}.

b) Geben Sie die Mengen $\mathit{Dom}\, R$, $\mathit{Ran}\, R$ und $\mathit{Ver}\, R$ an.

c) Ist R funktional? Injektiv? Total auf $\{1,2,4,5\}$? Surjektiv auf $\mathit{Ver}\, R$?

d) Welches Paar müssen Sie entfernen, damit R funktional und injektiv wird?

Aufgabe 8.12 Zeichnen Sie alle Relationen R mit $\mathit{Ver}\, R = \{1,2\}$,

a) die funktional und injektiv sind.

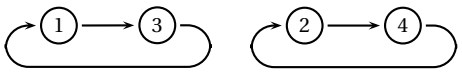
b) die total und surjektiv auf $\mathit{Ver}\, R$ sind.

8.5.4 Komposition von Relationen

Die **Komposition** $R \circ R'$ zweier Relationen R und R' ist die wie folgt definierte Relation:

$$R \circ R' := \{(x,z) \mid \exists (x,y) \in R\colon\ (y,z) \in R'\}$$

Die Komposition enthält ein Paar (x,z) also genau dann, wenn es einen gemeinsamen Knoten y von R und R' gibt, sodass (x,y) ein Paar von R und (y,z) ein Paar von R' ist. Die Komposition der oben dargestellten Relation mit sich selbst ergibt die folgende Relation:



Diese Relation ist funktional und injektiv sowie total und surjektiv auf $\{1,2,3,4\}$.

Proposition 8.1 *Seien R, R' und R'' Relationen. Dann gilt:*

1. $R \circ (R' \circ R'') = (R \circ R') \circ R''$.
2. $(R \circ R')^{-1} = R'^{-1} \circ R^{-1}$.
3. *Wenn R und R' funktional sind, dann ist $R \circ R'$ funktional.*
4. *Wenn R und R' injektiv sind, dann ist $R \circ R'$ injektiv.*

Die Komposition von Relationen ist also assoziativ (1), verträglich mit der Inversion (2) und erhält Funktionalität (3) und Injektivität (4).

Sei X eine Menge. Die **Identität auf** X ist die wie folgt definierte Relation:

$$Id(X) := \{(x,x) \mid x \in X\}$$

Offensichtlich gilt $Id(X) = Id(X)^{-1} = Id(X) \circ Id(X)$. Außerdem ist $Id(X)$ funktional, injektiv sowie total und surjektiv auf X.

Proposition 8.2 *Sei R eine Relation. Dann gilt:*

1. *R ist genau dann funktional, wenn $R^{-1} \circ R = Id(Ran\, R)$.*
2. *R ist genau dann injektiv, wenn $R \circ R^{-1} = Id(Dom\, R)$.*

Aufgabe 8.13 Sei die Relation $R = \{(1,2), (1,3), (2,4), (4,1), (5,2)\}$ gegeben. Zeichnen Sie die Komposition $R \circ R$.

8.5.5 Reflexivität, Transitivität und Ordnungen

Eine Relation R heißt

- **reflexiv**, wenn $Id(Ver\, R) \subseteq R$. Reflexivität bedeutet, dass R für jeden Knoten x die Kante (x,x) enthält.
- **transitiv**, wenn $R \circ R \subseteq R$. Transitivität bedeutet, dass R zu zwei Kanten (x,y) und (y,z) stets auch die komponierte Kante (x,z) enthält.
- **antisymmetrisch**, wenn $R \cap R^{-1} \subseteq Id(Ver\, R)$. Antisymmetrie bedeutet, dass für $x \neq y$ höchstens eines der Paare (x,y) und (y,x) in R ist.
- **linear**, wenn für alle $x, y \in Ver\, R$ mindestens eine der folgenden drei Eigenschaften erfüllt ist: $x = y$ oder $(x,y) \in R$ oder $(y,x) \in R$.

Unter einer **Ordnung** versteht man in mathematischer Sprechweise eine Relation, die reflexiv, transitiv und antisymmetrisch ist. Unter einer **Ordnung für eine Menge** X versteht man eine Ordnung R mit $Ver\, R = X$.

Proposition 8.3 (Natürliche Ordnung) *Die Menge $\{(x,y) \in \mathbb{N}^2 \mid x \leq y\}$ ist eine lineare Ordnung für $\mathbb{N}$.*

Proposition 8.4 (Inklusionsordnung) *Sei X eine Menge. Dann ist die Menge $\{(x,y) \in \mathscr{P}(X)^2 \mid x \subseteq y\}$ eine Ordnung für $\mathscr{P}(X)$.*

Aufgabe 8.14 Warum ist die Inklusionsordnung für $\mathscr{P}(\{1,2\})$ nicht linear?

8.6 Funktionen

Für die praktische mathematische Arbeit sind Funktionen genauso wichtig wie Mengen. Wir beginnen mit der mathematischen Definition von Funktionen.

Eine **Funktion** ist eine funktionale Relation.

Die charakteristische Eigenschaft einer Funktion f besteht darin, dass zu jedem $x \in \mathit{Dom}\ f$ genau ein $y \in \mathit{Ran}\ f$ mit $(x,y) \in f$ existiert. Dieses y bezeichnen wir mit $f\,x$ (sprich „f von x“). Oft wird $f\,x$ mit Klammern als $f(x)$ geschrieben. Das Objekt $f\,x$ wird als der **Wert von** f **für** x bezeichnet. Man sagt, dass f für x den Wert $f\,x$ **liefert** und dass f den Wert x auf $f\,x$ **abbildet**. Die Notation „$f\,x$“ ist von immenser praktischer Bedeutung, da sie es erlaubt, mit Funktionen Ausdrücke und Gleichungen zu bilden, wie zum Beispiel $f(x+g\,y) = g(f\,x+y)$.

Hier ist ein Beispiel für eine endliche und injektive Funktion:

$$f = \{(1,2),\ (3,4),\ (5,6),\ (7,8)\}$$

Diese Funktion hat die Graphdarstellung

Wir sagen, dass eine Funktion g **kleiner als** eine Funktion h ist, wenn $g \subseteq h$. Machen Sie sich klar, dass es sich bei der obigen Funktion f um die kleinste Funktion handelt, die die folgenden Gleichungen erfüllt: $f\,1 = 2,\ f\,3 = 4,\ f\,5 = 6,\ f\,7 = 8$.

Proposition 8.5 *Für alle Funktionen f, g und alle $x \in \mathit{Dom}\,(f \circ g)$ gilt:*
$(f \circ g)\,x = g(f\,x)$.

Sie werden vielleicht überrascht sein, dass bei einer Komposition $f \circ g$ zuerst f und dann g angewendet wird. Das kommt daher, dass wir Komposition mit dieser Orientierung für Relationen definiert haben (§ 8.5.4). Da Funktionen spezielle Relationen sind, gilt diese Definition auch für Funktionen. Die meisten Mathematikbücher verwenden für Funktionen allerdings die umgekehrte Orientierung. Auch Standard ML verwendet für die Komposition von Prozeduren die umgekehrte Orientierung (§ 3.12).

Die Umkehrrelation einer Funktion f ist genau dann eine Funktion, wenn f eine injektive Relation ist. Wenn die Umkehrrelation einer Funktion eine Funktion ist, bezeichnen wir sie als **Umkehrfunktion** oder **inverse Funktion**.

Proposition 8.6 *Für jede injektive Funktion f gilt:*

1. f^{-1} *ist eine injektive Funktion.*
2. $\forall x, y \in \mathit{Dom}\,f\colon\ f\,x = f\,y \iff x = y.$
3. $\forall x \in \mathit{Dom}\,f\colon\ f^{-1}(f\,x) = x.$
4. $\forall x \in \mathit{Ran}\,f\colon\ f(f^{-1}x) = x.$

Aufgabe 8.15 Sei g die injektive Funktion $\{(2,1),\ (4,3),\ (6,5),\ (8,7)\}$.

a) Zeichnen Sie die Graphdarstellung von g.

b) Geben Sie die Umkehrfunktion von g an.

c) Geben Sie Gleichungen an, sodass g die kleinste Funktion ist, die diese Gleichungen erfüllt.

8.6.1 Lambda-Notation

Oft ist es am einfachsten, eine Funktion durch die Angabe einer Vorschrift zu beschreiben, die zu jedem Argument das gewünschte Ergebnis liefert. Das kann mithilfe der sogenannte **Lambda-Notation** erfolgen, die den Abstraktionen von Standard ML entspricht. Beispielsweise kann die Funktion $\{(n,n) \mid n \in \mathbb{N}\}$ mithilfe der Lambda-Notation eleganter durch $\lambda\, n \in \mathbb{N}.\, n$ beschrieben werden. Hier sind zwei weitere Beispiele:

$$\lambda\, n \in \mathbb{N}.\, 2n + n^3 \;=\; \{(n,\ 2n+n^3) \mid n \in \mathbb{N}\}$$

$$\lambda\, (x,y) \in \mathbb{Z}^2.\, x + y \;=\; \{((x,y),\ x+y) \mid x \in \mathbb{Z},\ y \in \mathbb{Z}\}$$

Wie man an den Beispielen sehen kann, handelt es sich bei der Lambda-Notation um eine an die Beschreibung von Funktionen angepasste Variante der Mengennotation $\{x \mid A\}$.

Aufgabe 8.16 Geben Sie die folgenden Mengen an:

a) $Dom\,(\lambda\, x \in \mathbb{N}.\, x^2)$

b) $Ran\,(\lambda\, x \in \mathbb{N}.\, x^2)$

8.6.2 Funktionsmengen

Seien X und Y Mengen. Wir verwenden die folgenden Notationen:

$$X \rightharpoonup Y \;:=\; \{f \mid f \text{ Funktion mit } Dom\, f \subseteq X \text{ und } Ran\, f \subseteq Y\}$$
Funktionen von X nach Y

$$X \to Y \;:=\; \{f \mid f \text{ Funktion mit } Dom\, f = X \text{ und } Ran\, f \subseteq Y\}$$
Totale Funktionen von X nach Y

$$X \xrightarrow{\text{fin}} Y \;:=\; \{f \mid f \text{ endliche Funktion mit } Dom\, f \subseteq X \text{ und } Ran\, f \subseteq Y\}$$
Endliche Funktionen von X nach Y

Unter einer **Funktion** $X \to Y$ verstehen wir eine Funktion $f \in (X \to Y)$. Statt $f \in (X \to Y)$ schreiben wir kürzer $f \in X \to Y$. Wenn wir einer Funktion einen Namen geben wollen, zum Beispiel $f = \lambda\, x \in \mathbb{R}.\, 3x$, verwenden wir statt der Lambda-Notation oft die folgende Notation:

$$f \in \mathbb{R} \to \mathbb{R}$$
$$f\, x = 3x$$

Hier sind konkrete Beispiele für Funktionsmengen:

$$\mathbb{B} \to \mathbb{B} \;=\; \{\,\{(0,0),(1,0)\},\ \{(0,1),(1,1)\},\ \{(0,1),(1,0)\},\ \{(0,0),(1,1)\}\,\}$$

$$\mathbb{B} \rightharpoonup \mathbb{B} \;=\; (\mathbb{B} \to \mathbb{B}) \cup \{\,\emptyset,\ \{(0,0)\},\ \{(0,1)\},\ \{(1,0)\},\ \{(1,1)\}\}$$

$$\mathbb{B} \overset{\text{fin}}{\rightharpoonup} \mathbb{B} \;=\; \mathbb{B} \rightharpoonup \mathbb{B}$$

Proposition 8.7 *Seien X, Y, Z Mengen und f, g Funktionen. Dann gilt:*

1. $f \in X \rightharpoonup Y \;\wedge\; g \in Y \rightharpoonup Z \implies f \circ g \in X \rightharpoonup Z$
2. $f \in X \to Y \;\wedge\; g \in Y \to Z \implies f \circ g \in X \to Z$
3. $f \in X \overset{\text{fin}}{\rightharpoonup} Y \;\wedge\; g \in Y \overset{\text{fin}}{\rightharpoonup} Z \implies f \circ g \in X \overset{\text{fin}}{\rightharpoonup} Z$

Aufgabe 8.17 Beschreiben Sie mithilfe der Lambda-Notation:

a) Eine Funktion $\mathbb{N} \to \mathbb{N}$, die nicht injektiv ist.

b) Eine injektive Funktion $\mathbb{N} \to \mathbb{N}$, die surjektiv auf $\mathbb{N}$ ist.

c) Eine injektive Funktion $\mathbb{N} \to \mathbb{N}$, die nicht surjektiv auf $\mathbb{N}$ ist.

8.6.3 Klammersparregeln

$$\begin{aligned} X \times Y \to Z &\quad\rightsquigarrow\quad (X \times Y) \to Z \\ X \to Y \to Z &\quad\rightsquigarrow\quad X \to (Y \to Z) \\ f\,x\,y &\quad\rightsquigarrow\quad (f\,x)\,y \end{aligned}$$

8.6.4 Adjunktion

In § 2.7 sind wir der Adjunktion von Umgebungen begegnet. Da wir Umgebungen später als Funktionen formalisieren werden, wollen wir den Begriff der Adjunktion für allgemeine Funktionen definieren.

Die **Adjunktion** zweier Funktionen f, g ist die wie folgt definierte Funktion:

$$f + g \;:=\; \lambda\, x \in \mathit{Dom}\, f \cup \mathit{Dom}\, g.\ \text{if } x \in \mathit{Dom}\, g \text{ then } g\,x \text{ else } f\,x$$

Die Funktion $f + g$ verhält sich für alle Werte, auf denen g definiert ist, wie g. Für alle anderen Werte verhält sich $f + g$ wie f. Für jede Funktion f gilt $f + f = f$. Für einen häufig vorkommenden Spezialfall der Adjunktion definieren wir eine eigene Notation:

$$f[x{:=}y] \;:=\; f + \{(x,y)\} \qquad (\text{lies „}f \text{ mit } x \text{ nach } y\text{“})$$

Hier sind Beispiele für konkrete Adjunktionen:

$$\{(1,5),\ (2,6)\} + \{(2,7),\ (3,8)\} \;=\; \{(1,5),\ (2,7),\ (3,8)\}$$

$$\{(1,5),\ (2,7),\ (3,8)\}[2{:=}0] \;=\; \{(1,5),\ (2,0),\ (3,8)\}$$

$$(\lambda\, x \in \mathbb{N}.\ x+3)[7{:=}5] \;=\; \lambda\, x \in \mathbb{N}.\ \text{if } x = 7 \text{ then } 5 \text{ else } x+3$$

Beachten Sie, dass die Adjunktion von Funktionen im Gegensatz zur Addition von Zahlen keine kommutative Operation ist.

8.6.5 Bijektionen und Darstellungen

Unter einer **Bijektion** $X \to Y$ verstehen wir eine injektive Funktion $X \to Y$, die surjektiv auf Y ist. Eine Bijektion $f \in X \to Y$ stellt einen symmetrischen Zusammenhang zwischen X und Y her, da ihre Umkehrfunktion eine Bijektion $Y \to X$ ist:

$$X \underset{f^{-1}}{\overset{f}{\rightleftarrows}} Y$$

Hier ist ein Beispiel für eine Bijektion, die Listen über X mit Tupeln über X verknüpft:

$$B \in \mathscr{L}(X) \to X^*$$
$$B[x_1, \ldots, x_n] = \langle x_1, \ldots, x_n \rangle$$

Unter einer **Darstellung** für eine Menge X verstehen wir eine Funktion f mit $Ran\, f = X$. Dabei fassen wir die Elemente von $Dom\, f$ als Darstellungen für die Elemente von X auf. Wenn f eine Darstellung für X ist, existiert für jedes $x \in X$ mindestens ein $d \in Dom\, f$, das x gemäß $f\,d = x$ darstellt. Wenn f eine injektive Darstellung für X ist, existiert für jedes $x \in X$ genau ein $d \in Dom\, f$, das x gemäß $f\,d = x$ darstellt. Injektive Darstellungen bezeichnen wir auch als **eindeutige Darstellungen**. Ein Beispiel für eine eindeutige Darstellung ist die obige Funktion B, die Tupel über X durch Listen über X darstellt. Ein Beispiel für eine nicht eindeutige Darstellung ist die Funktion

$$\mathrm{Set} \in \mathscr{L}(X) \to \mathscr{P}(X)$$
$$\mathrm{Set}[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$$

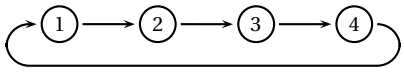
die endliche Mengen über X durch Listen über X darstellt. Beispielsweise gilt $\{2,3\} = \mathrm{Set}[2,3] = \mathrm{Set}[3,2]$. Machen Sie sich klar, dass der hier definierte Darstellungsbegriff genau dem informellen Darstellungsbegriff aus § 5.5 und § 7.8 entspricht.

Schließlich halten wir noch fest, dass eine eindeutige Darstellung für X eine Bijektion $Dom\, f \to X$ ist und dass eine Bijektion $X \to Y$ eine eindeutige Darstellung für Y ist.

8.7 Terminierende Relationen

Unter einer terminierenden Relation verstehen wir eine Relation, die gemäß der Graphsicht ein Spielfeld beschreibt, dass keine unendliche Zugfolge zulässt. Wir interessieren uns für terminierende Relationen, da sich die Terminierungseigenschaften von rekursiven Prozeduren als die Terminierungseigenschaften von Relationen analysieren lassen.

Zunächst wollen wir präzise definieren, was wir unter einer terminierenden Relation verstehen wollen. Eine Relation R heißt **fortschreitend**, wenn sie nichtleer ist und für jeden Knoten $x \in \mathit{Ver}\, R$ eine Kante $(x, y) \in R$ existiert, die von x ausgeht. Hier ist ein Beispiel für eine fortschreitende Relation:



Eine Relation heißt **terminierend**, wenn sie keine fortschreitende Relation enthält. Wir sagen, dass eine Relation R für ein Objekt x **terminiert**, wenn es keine fortschreitende Relation $R' \subseteq R$ mit $x \in \mathit{Ver}\, R'$ gibt. Die obige Relation terminiert für keinen ihrer Knoten. Statt zu sagen, dass eine Relation terminierend ist, sagen wir auch, dass sie **terminiert**.

Proposition 8.8 *Eine fortschreitende Relation terminiert für keinen ihrer Knoten.*

Proposition 8.9 *Eine Relation terminiert genau dann, wenn sie für jeden ihrer Knoten terminiert.*

Proposition 8.10 *Jede Teilmenge einer terminierenden Relation ist eine terminierende Relation.*

Proposition 8.11 *Eine endliche Relation terminiert genau dann, wenn sie azyklisch ist.*

Das Paradebeispiel für eine unendliche und terminierende Relation ist

$$\mathit{Ter} := \{(x, y) \in \mathbb{N}^2 \mid x > y\}$$

Für die nachfolgenden Betrachtungen benötigen wir den Begriff der Terminierungsfunktion. Sei R eine Relation. Eine Funktion $f \in \mathit{Ver}\, R \to \mathbb{N}$ heißt **natürliche Terminierungsfunktion** für R, wenn für jede Kante $(x, y) \in R$ gilt: $f\, x > f\, y$.

Eine natürliche Terminierungsfunktion ordnet jedem Knoten einer Relation eine Größe zu, sodass die Größe beim Durchlauf durch eine Kante echt kleiner wird. Da die Größen natürliche Zahlen sind, muss die Relation terminieren. Ansonsten gäbe es Knoten, deren Größe beliebig oft verkleinert werden könnte.

Proposition 8.12 *Jede Relation, für die es eine natürliche Terminierungsfunktion gibt, ist terminierend.*

Beweis Durch Widerspruch. Sei R eine nicht terminierende Relation und f eine natürliche Terminierungsfunktion für R. Da R nicht terminierend ist, gibt es eine fortschreitende Relation $R' \subseteq R$. Also ist $\{(f\, x, f\, y) \mid (x, y) \in R'\}$ eine fortschreitende Relation, die in *Ter* enthalten ist. Widerspruch. ∎

Als Beispiel betrachten wir die Relation

$$R = \{((x,y),(x',y')) \in (\mathbb{N}^2)^2 \mid x + y > x' + y'\}$$

Die Funktion $\lambda(x,y) \in \mathbb{N}^2.\ x+y$ ist eine natürliche Terminierungsfunktion für R. Also ist R gemäß Proposition 8.12 terminierend.

Aufgabe 8.18 Geben Sie eine unendliche Relation an, die für keinen ihrer Knoten terminiert.

Aufgabe 8.19 Geben Sie eine unendliche, funktionale und terminierende Relation an.

Aufgabe 8.20 Beweisen Sie die Terminierung der folgenden Relationen, indem Sie natürliche Terminierungsfunktionen angeben.

a) $\{((x,y),(x',y')) \in (\mathbb{N} \times \mathbb{R})^2 \mid x > x' \wedge\ y < y'\}$

b) $\{((x,y),(x',y')) \in (\mathbb{Z} \times \mathbb{Z})^2 \mid x + y > x' + y' \geq -150\}$

8.8 Strukturelle Terminierungsfunktionen

Wir haben die Terminierung der Relation *Ter* als gegeben angenommen, da es sich um eine grundlegende Eigenschaft der natürlichen Zahlen handelt. Da wir die natürlichen Zahlen als Mengen darstellen, haben wir jedoch die Möglichkeit, die Terminierung von *Ter* durch Rückgriff auf das Wohlfundierungsaxiom (§ 8.1) zu beweisen. In der Tat handelt es sich beim Wohlfundierungsaxiom um eine grundlegende Terminierungsannahme für Mengen. Zunächst benötigen wir den Begriff einer Konstituenten.

Die **Konstituenten** einer Menge X sind rekursiv wie folgt definiert:

1. Jedes Element von X ist eine Konstituente von X.
2. Jede Konstituente jedes Elements von X ist eine Konstituente von X.

Die Menge $\{\{\{\emptyset\}\}\}$ hat genau 3 Konstituenten: $\{\{\emptyset\}\}$, $\{\emptyset\}$ und $\emptyset$. Die Menge $\{\emptyset,\ \{\emptyset\},\ \{\emptyset,\{\emptyset\}\}\}$ hat die Besonderheit, dass jede ihrer Konstituenten auch eines ihrer Elemente ist.

Proposition 8.13 *Seien x, y Mengen. Dann ist x genau dann eine Konstituente von y, wenn $n \geq 2$ Mengen $x_1, \ldots, x_n$ existieren, sodass $x = x_1,\ x_1 \in x_2, \ldots,\ x_{n-1} \in x_n$ und $x_n = y$ gilt.*

Proposition 8.14 *Eine Menge ist genau dann finitär, wenn sie nur endlich viele Konstituenten hat.*

Der Zusammenhang zwischen Terminierung und dem Wohlfundierungsaxiom lässt sich durch strukturelle Relationen beschreiben. Eine Relation R heißt **strukturell**, wenn für jede Kante $(x,y) \in R$ gilt, dass y eine Konstituente von x ist.

Proposition 8.15 *Jede strukturelle Relation ist terminierend.*

Beweis Folgt mit Widerspruch aus dem Wohlfundierungsaxiom. ∎

Gemäß unserer Darstellung der natürlichen Zahlen (Aufgabe 7.39, S. 148) ist *Ter* eine strukturelle Relation. Damit folgt die Terminierung von *Ter* mit der obigen Proposition.

Sei R eine Relation. Eine Funktion f heißt **strukturelle Terminierungsfunktion** für R, wenn $Dom\, f = Ver\, R$, und wenn für jede Kante $(x, y) \in R$ gilt, dass $f\, y$ eine Konstituente von $f\, x$ ist.

Proposition 8.16 *Jede Relation, für die es eine strukturelle Terminierungsfunktion gibt, ist terminierend.*

Beweis Durch Widerspruch. Sei R eine nicht terminierende Relation, für die es eine strukturelle Terminierungsfunktion gibt. Dann können wir eine strukturelle Relation R' konstruieren, die fortschreitend ist. Das widerspricht Proposition 8.15. ∎

Aufgabe 8.21 Geben Sie eine zweielementige Menge wie folgt an:

1. Jede Konstituente der Menge ist ein Element der Menge.
2. Jede nichtleere Konstituente der Menge ist eine einelementige Menge.

Können Sie auch eine drei- und eine vierelementige Menge mit diesen Eigenschaften angeben?

Bemerkungen

Es ist heute üblich, alle mathematischen Objekte durch Mengen darzustellen. Die Darstellung einer Objektklasse durch Mengen und eine darauf beruhende Begriffsdefinition bezeichnet man als **Formalisierung**. Durch die Formalisierung einer Objektklasse erreicht man, dass die Objekte der Klasse und die für ihre Untersuchung erforderlichen Begriffe präzise und vollständig beschrieben sind. Formalisierung ist die Voraussetzung für die präzise Formulierung und den verlässlichen Beweis von Eigenschaften.

Als Begründer der Mengenlehre gilt Georg Cantor (1845-1918). Sein Interesse galt vor allem der systematischen Erforschung unendlicher Mengen. Die Idee, alle mathematischen Objekte als Mengen darzustellen, entwickelte sich nach 1900. Kazimierz Kuratowski stellte 1921 geordnete Paare durch Mengen und Funktionen durch Mengen geordneter Paare dar. Das Wohlfundierungsaxiom wurde 1925 von John von Neumann formuliert. Es schränkt die Bildung von Mengen so ein, dass die Russellsche Antinomie gegenstandslos wird. Der Begriff der Funktion wird René Descartes (1596–1650) zugeschrieben. Graphen wurden erstmals von Leonhard Euler (1707–1783) betrachtet (Königsberger Brückenproblem).

Wenn Sie mit den Begriffen und der mathematischen Sprechweise dieses Kapitels Schwierigkeiten haben, kann Ihnen eine der sanften Einführungen in die diskrete Mathematik helfen, die in den letzten Jahren speziell in Hinblick auf die Bedürfnisse der

Informatik entwickelt wurden, etwa das didaktisch hervorragende und zum Selbststudium gut geeignete Buch von Rosen [6].

Verzeichnis

Mengen: Elemente, Konstituenten; $x \in X$, $x \notin X$; leere Menge $\emptyset$; n-elementige Mengen $\{x_1, \ldots, x_n\}$, Kardinalität $|X|$; endliche und finitäre Mengen; Gleichheits- und Wohlfundierungsaxiom; (echte) Teilmengen, (echte) Obermengen, $X \subset Y$, $X \subseteq Y$; Disjunktheit; Notation $\{x \mid A\}$; Schnitt $X \cap Y$, Vereinigung $X \cup Y$, Differenz $X - Y$, Potenzmenge $\mathscr{P}(X)$.

Ausgezeichnete Mengen: $\mathbb{B}$, $\mathbb{N}$, $\mathbb{N}_+$, $\mathbb{Z}$, $\mathbb{R}$.

Aussagen: Boolesche Operatoren und Quantifizierung; Negation $\neg A$, Konjunktion $A \wedge B$, Disjunktion $A \vee B$, Implikation $A \Rightarrow B$, Äquivalenz $A \Leftrightarrow B$; universelle Quantifizierung $\forall x \in X \colon A$, existentielle Quantifizierung $\exists x \in X \colon A$; Definition von Notationen mit $:=$ und $:\Longleftrightarrow$.

Tupel: $\langle x_1, \ldots, x_n \rangle$; Länge, Komponenten, Positionen; Paare, Tripel; Tupel über X, X^*; Listen $\mathscr{L}(X)$ und Bäume $\mathscr{T}(X)$; Produkte $X \times Y$ und Summen $X + Y$.

Gerichtete Graphen: $G = (V, E)$; Knoten und Kanten; Layout und Spielsicht von Graphen; Nachfolger und Vorgänger; initiale (Quellen), terminale (Senken), isolierte und benachbarte Knoten; Pfade (Ausgangs- und Endpunkt, Länge, einfach); Zyklen; Wurzeln und Erreichbarkeit; endliche, symmetrische, gewurzelte, zyklische, und azyklische Graphen; Tiefe von endlichen Graphen; inverse Kanten und symmetrischer Abschluss; (stark) zusammenhängende Graphen; (erreichbare) Teilgraphen; baumartige Graphen.

Binäre Relationen: Definitionsbereich $Dom\, R$, Wertebereich $Ran\, R$, Knotenmenge $Ver\, R$, Knoten von R; zugehöriger Graph, Graphsicht; Relation auf X; Ein-Ausgabe-Relation einer Prozedur; Umkehrrelation R^{-1}, inverse Relation; funktionale, injektive, totale und surjektive Relationen; Komposition $R \circ R'$; Identität $Id(X)$; Reflexivität, Transitivität, Antisymmetrie, Linearität; Ordnungen; Natürliche Ordnung, Inklusionsordnung;

Funktionen: $f\,x$; Umkehrfunktion, inverse Funktion; Lambda-Notation $\lambda\, x \in X.\, A$; Funktionsmengen $X \rightharpoonup Y$, $X \to Y$ und $X \xrightarrow{\text{fin}} Y$; Funktionen $X \to Y$; Adjunktion $f + g$ und $f[x{:=}y]$.

Bijektionen und (eindeutige) Darstellungen.

Terminierende Relationen: fortschreitende Relationen; strukturelle Relationen; natürliche und strukturelle Terminierungsfunktionen.

Formalisierung.

9 Mathematische Prozeduren

In diesem Kapitel betrachten wir mathematische Prozeduren, die unabhängig von einer konkreten Programmiersprache existieren. Unser Hauptinteresse gilt dem Beweis von Korrektheitseigenschaften. Dazu zählt neben der Terminierung rekursiver Prozeduren die Eigenschaft, dass eine Prozedur eine gegebene Funktion berechnet.

9.1 Beschreibung

Unter einer **mathematischen Prozedur** verstehen wir eine Berechnungsvorschrift, die bei der **Anwendung** auf ein **Argument** ein **Ergebnis** liefert. Mathematische Prozeduren werden durch eine oder mehrere Gleichungen beschrieben, die wir als die **definierenden Gleichungen** der Prozedur bezeichnen. Zu einer mathematischen Prozedur müssen zwei Mengen angegeben werden, die als **Argumentbereich** und **Ergebnisbereich** der Prozedur bezeichnet werden. Die Argumente der Prozedur müssen Elemente des Argumentbereichs sein und die Ergebnisse Elemente des Ergebnisbereichs. Hier ist ein Beispiel für eine mathematische Prozedur, die den Absolutbetrag einer ganzen Zahl bestimmt:

$$abs : \mathbb{Z} \to \mathbb{N}$$
$$abs\ x = \text{if } x < 0 \text{ then } -x \text{ else } x$$

Der **Name** der Prozedur ist *abs*, der Argumentbereich $\mathbb{Z}$, der Ergebnisbereich $\mathbb{N}$, und die definierende Gleichung der Prozedur ist $abs\ x = \text{if } x < 0 \text{ then } -x \text{ else } x$.

Mathematische Prozeduren sind mathematische Objekte, die unabhängig von einer Programmiersprache existieren. Sie liefern uns ein elegantes Modell, mit dem wir Korrektheit und Laufzeit von Prozeduren analysieren können. Mathematische Prozeduren ersparen uns die Behandlung von Details, die erst im Zusammenhang mit einer konkreten Programmiersprache relevant werden.

Im Folgenden werden wir mathematische Prozeduren einfach als Prozeduren bezeichnen. Aus dem Zusammenhang wird immer klar sein, ob wir über eine mathematische Prozedur oder eine Prozedur in Standard ML sprechen.

Abbildung 9.1 zeigt einige rekursive Prozeduren, die uns als Beispiele dienen werden. Die Prozeduren *fac* und fac' berechnen die Fakultäten (Aufgabe 1.26 auf S. 22), die Prozedur *fib* die sogenannten Fibonacci-Zahlen, und die Prozeduren *euclid* und *gcd* berechnen den größten gemeinsamen Teiler zweier positiver Zahlen (§ 9.8). Die Prozedur *euclid*

$fac : \mathbb{N} \to \mathbb{N}$
$fac\ 0 = 1$
$fac\ n = n \cdot fac(n-1) \quad \text{für } n > 0$

$fac' : \mathbb{Z} \to \mathbb{Z}$
$fac'\ x = \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot fac'(x-1)$

$fib : \mathbb{N} \to \mathbb{N}$
$fib\ n = \text{if } n < 2 \text{ then } n \text{ else } fib(n-1) + fib(n-2)$

$euclid : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$euclid(x, y) = \text{if } y = 0 \text{ then } x \text{ else } euclid(y,\ x \bmod y)$

$gcd : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+$
$gcd(x, x) = x$
$gcd(x, y) = gcd(x - y,\ y) \quad \text{für } x > y$
$gcd(x, y) = gcd(x,\ y - x) \quad \text{für } x < y$

Abbildung 9.1: *Beispiele für rekursive Prozeduren*

formuliert einen Algorithmus, der in der Informatik gerne als *Euklidscher Algorithmus* bezeichnet wird. Belegt ist, dass Euklid etwa 300 vor Christus eine geometrische Version des durch die Prozedur *gcd* beschriebenen Algorithmus betrachtet hat.

Unter einer **Prozedur** $X \to Y$ verstehen wir eine mathematische Prozedur mit dem Argumentbereich X und dem Ergebnisbereich Y.

Aus Gründen der Einfachheit werden wir immer nur eine Prozedur betrachten. Mathematische Prozeduren können also keine Hilfsprozeduren verwenden. Dafür können mathematische Prozeduren beliebige Funktionen verwenden.

Wir werden nur Prozeduren betrachten, deren definierende Gleichungen die folgenden **Wohlgeformtheitsbedingungen** erfüllen:

- Rekursive Anwendungen der Prozedur erfolgen nur auf Elemente des Argumentbereichs der Prozedur.
- Funktionen werden nur auf Elemente ihres Definitionsbereichs angewendet.
- Es werden nur Ergebnisse im Ergebnisbereich der Prozedur geliefert.
- Die definierenden Gleichungen sind **disjunkt** und **erschöpfend** (§ 4.6).

Mathematische Prozeduren können im Rahmen der Mengenlehre formalisiert werden. Allerdings ist dafür mehr Aufwand erforderlich als für Funktionen, da ein Mindestmaß an Syntax behandelt werden muss. Daher verzichten wir auf die Formalisierung mathematischer Prozeduren und betrachten stattdessen abgeleitete Objekte, deren Formalisierung wir bereits kennen.

Aufgabe 9.1 Geben Sie den Argumentbereich und den Ergebnisbereich der Prozedur *gcd* an.

Aufgabe 9.2 Bleiben die Wohlgeformtheitsbedingungen für die definierenden Gleichungen gültig, wenn man bei der Prozedur

a) *fac* den Ergebnisbereich zu $\mathbb{Z}$ verändert?

b) *fac* den Ergebnisbereich zu $\mathbb{N}_+$ verändert?

c) *fac* den Argumentbereich und den Ergebnisbereich zu $\mathbb{Z}$ verändert?

d) *fac'* den Argumentbereich zu $\mathbb{N}$ verändert?

e) *euclid* den Ergebnisbereich zu $\mathbb{N}_+$ verändert?

f) *gcd* den Argumentbereich zu $\mathbb{N} \times \mathbb{N}$ und den Ergebnisbereich zu $\mathbb{N}$ verändert?

9.2 Ausführung

Eine Prozedur stellt eine Berechnungsvorschrift dar, deren Ausführung für ein gegebenes Argument entweder mit einem Ergebnis terminiert oder aber unendlich fortschreitet. Die Ausführung von Prozeduren werden wir mithilfe von Ausführungsprotokollen beschreiben.

Die definierenden Gleichungen einer Prozedur liefern für jedes x im Argumentbereich der Prozedur eine **Anwendungsgleichung**. Diese erhält man, indem man den Wert x in die zuständige definierende Gleichung einsetzt und die rechte Seite der so erhaltenen Gleichung soweit wie möglich auswertet. Hier sind Anwendungsgleichungen für unsere Beispielprozeduren:

$$\begin{aligned} \mathit{fac}\,5 &= 5 \cdot \mathit{fac}\,4 \\ \mathit{fac}'(-3) &= -3 \cdot \mathit{fac}'(-4) \\ \mathit{fib}\,4 &= \mathit{fib}\,3 + \mathit{fib}\,2 \\ \mathit{euclid}(12,4) &= \mathit{euclid}(4,0) \\ \mathit{gcd}(12,4) &= \mathit{gcd}(8,4) \\ \mathit{gcd}(7,7) &= 7 \end{aligned}$$

Beachten Sie, dass für jede Prozedur und für jedes Argument im Argumentbereich der Prozedur genau eine Anwendungsgleichung existiert.

Für mathematische Prozeduren betrachten wir **Ausführungsprotokolle**, die Prozeduranwendungen gemäß den Anwendungsgleichungen der Prozedur ersetzen. Hier ist

ein Beispiel:

$$\begin{aligned} \mathit{fib}\,3 &= \mathit{fib}\,2 + \mathit{fib}\,1 \\ &= (\mathit{fib}\,1 + \mathit{fib}\,0) + \mathit{fib}\,1 \\ &= (1 + \mathit{fib}\,0) + \mathit{fib}\,1 \\ &= (1+0) + \mathit{fib}\,1 \\ &= 1 + \mathit{fib}\,1 \\ &= 1+1 \\ &= 2 \end{aligned}$$

Neben Prozeduranwendungen werden in Ausführungsprotokollen auch auswertbare Teilausdrücke (z.B. $1+0$) durch ihre Werte ersetzt. Wenn es bei der Fortsetzung eines Ausführungsprotokolls mehr als eine Ersetzungsmöglichkeit gibt, hat die am weitesten links stehende Vorrang. Ein Ausführungsprotokoll heißt **terminierend**, wenn es so wie das obige Protokoll mit einem Wert endet.

Bis auf Konditionale dürfen bei der Bildung von Anwendungsgleichungen nur Teilausdrücke ausgewertet werden, die keine Prozeduranwendungen enthalten. Diese Vorgabe sorgt dafür, dass man bei der Prozedur

$$\begin{aligned} &p : \mathbb{N} \to \mathbb{N} \\ &pn = n \cdot pn \end{aligned}$$

für das Argument 0 die Anwendungsgleichung $p0 = 0 \cdot p0$ erhält (statt $p0 = 0$). Folglich gibt es für $p0$ kein terminierendes Ausführungsprotokoll.

Sei p eine Prozedur $X \to Y$ und $x \in X$. Wir sagen, dass die Anwendung[1]

- px **mit dem Ergebnis** y **terminiert**, wenn es ein Ausführungsprotokoll gibt, das mit px beginnt und mit y endet.
- px **divergiert**, wenn es kein terminierendes Ausführungsprotokoll gibt, das mit px beginnt.

Wenn px mit dem Ergebnis y terminiert, sagen wir auch, dass p **für** x **mit** y **terminiert** oder dass p **für** x **das Ergebnis** y **liefert**. Wenn px divergiert, sagen wir auch, dass p **für** x **divergiert**. Weiter sagen wir, dass px **terminiert** oder dass p **für** x **terminiert** oder dass x **ein terminierendes Argument von** p **ist**, wenn es ein terminierendes Ausführungsprotokoll gibt, das mit px beginnt. Schließlich nennen wir eine Prozedur **terminierend**, oder sagen, dass eine Prozedur **terminiert**, wenn die Prozedur für alle Elemente ihres Argumentbereichs terminiert.

[1] Im Kontext von Standard ML haben wir streng zwischen Prozeduranwendungen (Syntax) und Prozeduraufrufen (Semantik) unterschieden. Da Syntax bei mathematischen Prozeduren keine wesentliche Rolle spielt, werden wir bei mathematischen Prozeduren auf diese Unterscheidung verzichten und nur von Prozeduranwendungen sprechen.

Die Prozeduren *fac*, *fib*, *euclid* und *gcd* sind terminierend. Die Prozedur fac' terminiert für alle natürlichen Zahlen und divergiert für alle negativen ganzen Zahlen. Da wir keine Hilfsprozeduren zulassen, kann die Anwendung einer Prozedur nur dann divergieren, wenn die Prozedur rekursiv ist.

Der **Definitionsbereich** einer Prozedur $p : X \to Y$ ist die Menge aller terminierenden Argumente:

$$Dom\ p := \{x \in X \mid p \text{ terminiert für } x\}$$

Wenn die Anwendung einer Prozedur auf ein Argument terminiert, liefert sie ein eindeutig bestimmtes Ergebnis, bei dem es sich um ein Element des Ergebnisbereichs der Prozedur handelt.

Die Ausführung einer Prozeduranwendung findet gedanklich statt, nicht auf einem realen Interpreter. Es gibt also keine Speicherplatzbeschränkung und keine Größenbeschränkung für Zahlen. Zusammen mit den Wohlgeformtheitsbedingungen für Prozeduren sorgt dies dafür, dass keine Laufzeitfehler auftreten können.

Eine Prozedur p **erweitert** eine Prozedur q, wenn jede Anwendungsgleichung für q bis auf den Prozedurnamen eine Anwendungsgleichung für p ist. Beispielsweise erweitert die Prozedur fac' die Prozedur *fac*. Als weiteres Beispiel betrachten wir die Prozedur

$$\begin{aligned}
&fib' : \mathbb{N} \to \mathbb{N}\\
&fib'\ n = n \quad \text{für } n < 2\\
&fib'\ n = fib'(n-1) + fib'(n-2) \quad \text{für } n \geq 2
\end{aligned}$$

Offensichtlich erweitert fib' die Prozedur *fib*. Umgekehrt erweitert aber auch *fib* die Prozedur fib'. Wir sagen, dass sich die Prozeduren *fib* und fib' **wechselseitig erweitern**.

Satz 9.1 (Erweiterung) *Wenn eine Prozedur p eine Prozedur q erweitert, gilt:*

1. *Der Argumentbereich von q ist eine Teilmenge des Argumentbereichs von p.*
2. *Jedes Ausführungsprotokoll für q ist ein Ausführungsprotokoll für p (bis auf den Prozedurnamen).*
3. *Terminiert q für x mit dem Ergebnis y, so gilt dies auch für p.*

Beweis Die erste Behauptung folgt aus der Tatsache, dass für jedes Argument eine Anwendungsgleichung existiert. Die zweite Behauptung folgt aus der Tatsache, dass die Ausführungsprotokolle einer Prozedur mit den Anwendungsgleichungen der Prozedur gebildet werden. Die dritte Behauptung folgt aus der zweiten Behauptung. ∎

Aufgabe 9.3 Geben Sie die Anwendungsgleichungen für die folgenden Anwendungen der Beispielprozeduren an:

a) $fib\,7$

b) $euclid(63, 35)$

c) $gcd(35, 21)$

Aufgabe 9.4 Geben Sie eine Anwendungsgleichung für fac' an, die nicht in entsprechender Form für *fac* existiert.

Aufgabe 9.5 Geben Sie eine Prozedur $euclid' : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ an, die *euclid* erweitert und deren definierenden Gleichungen ohne Konditionale formuliert sind.

9.3 Rekursionsfunktionen

Sei eine Prozedur gegeben. Dann können wir für ein Argument x die **Folgeargumente** $x_1, \ldots, x_n$ betrachten, für die rekursive Anwendungen der Prozedur erforderlich sind, um das Ergebnis der Anwendung der Prozedur auf x zu bestimmen. Hier sind Beispiele:

- Bei *fac* hat 4 das Folgeargument 3.
- Bei *fib* hat 4 die Folgeargumente 3 und 2.

Die sogenannte **Rekursionsfunktion** einer Prozedur bildet jedes Argument auf das Tupel seiner Folgeargumente ab. Hier sind die Rekursionsfunktionen einiger unserer Beispielprozeduren:

$$
\begin{array}{rcl}
abs & : & \lambda\, n \in \mathbb{Z}.\, \langle\rangle \\
fac & : & \lambda\, n \in \mathbb{N}.\ \text{if } n = 0 \text{ then } \langle\rangle \text{ else } \langle n-1 \rangle \\
fib & : & \lambda\, n \in \mathbb{N}.\ \text{if } n < 2 \text{ then } \langle\rangle \text{ else } \langle n-1,\ n-2 \rangle \\
gcd & : & \lambda\, (x,y) \in \mathbb{N}_+ \times \mathbb{N}_+.\ \text{if } x = y \text{ then } \langle\rangle \text{ else} \\
 & & \qquad\qquad \text{if } x > y \text{ then } \langle (x-y, y) \rangle \text{ else } \langle (x, y-x) \rangle
\end{array}
$$

Wenn es wie bei *fib* mehr als ein Folgeargument gibt, werden die Folgeargumente gemäß der von links nach rechts schreitenden Ausführungsreihenfolge angeordnet.

Sei p eine Prozedur $X \to Y$. Dann ist die Rekursionsfunktion von p eine totale Funktion $r \in X \to X^*$ (auch dann, wenn die Prozedur für alle Argumente divergiert). Mithilfe der Rekursionsfunktion r von p können wir die folgenden, bereits bekannten Begriffe präzise definieren:

- p heißt **rekursiv**, wenn es mindestens ein Argument $x \in X$ gibt, für das $r\,x$ nicht leer ist.
- p heißt **linear-rekursiv**, wenn p rekursiv ist und $r\,x$ für alle Argumente $x \in X$ höchstens eine Position hat.
- p heißt **baumrekursiv**, wenn es mindestens ein Argument $x \in X$ gibt, für das $r\,x$ zwei oder mehr Positionen hat.

Die Prozeduren *fac* und *gcd* sind linear-rekursiv und die Prozedur *fib* ist baumrekursiv. Ein Blick auf die Rekursionsfunktion von *abs* zeigt, dass *abs* nicht rekursiv ist.

Die Beschreibung der Rekursionsfunktion lässt sich mechanisch aus der Prozedur ableiten. Solange die rekursiven Anwendungen der Prozedur nicht ineinander verschachtelt sind (z.B. $p(p\ x)$), ergibt sich eine nicht-rekursive Beschreibung der Rekursionsfunktion.

Aufgabe 9.6 Geben Sie die Rekursionsfunktion der Prozedur *fac'* an.

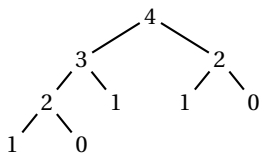
Aufgabe 9.7 Geben Sie eine terminierende und baumrekursive Prozedur $\mathbb{N} \to \mathbb{N}$ an, die für jedes Argument das Ergebnis 0 liefert.

Aufgabe 9.8 Geben Sie eine Prozedur mit der folgenden Rekursionsfunktion an: $\lambda\, x \in \mathbb{Z}.$ if $x < 1$ then $\langle\rangle$ else $\langle x-1,\ x-1\rangle$.

9.4 Rekursionsbäume

Für jedes terminierende Argument einer Prozedur kann die Abfolge der rekursiven Anwendungen durch einen sogenannten **Rekursionsbaum** beschrieben werden. Rekursionsbäume sind uns bereits im Zusammenhang mit der baumrekursiven Prozedur *msort* begegnet (§ 5.4).

Sei p eine Prozedur $X \to Y$. Dann sind die Rekursionsbäume von p Bäume über X. Als Beispiel geben wir den Rekursionsbaum der Prozedur *fib* für das Argument 4 an:



Die Rekursionsbäume einer Prozedur können rekursiv mithilfe der Rekursionsfunktion der Prozedur bestimmt werden. Der Rekursionsbaum für ein Argument x hat den Kopf x und die Rekursionsbäume der Folgeargumente als Unterbäume.

Für ein Argument x existiert genau dann ein Rekursionsbaum, wenn die Prozedur für x terminiert. Wir können die Rekursionsfunktion einer Prozedur als eine Vorschrift ansehen, gemäß der die Rekursionsbäume der Prozedur gebildet werden können.

Die Rekursionsbäume linear-rekursiver Prozeduren sind lineare Bäume. Wir bezeichnen sie daher auch als **Rekursionsfolgen** (§ 1.9) und zeichnen sie von links nach rechts statt von oben nach unten. Hier ist die Rekursionsfolge für die Prozedur *euclid* und das Argument (91,35):

$$(91,35) \to (35,21) \to (21,14) \to (14,7) \to (7,0)$$

Die Rekursionsfolge für *gcd* und (91,35) sieht wie folgt aus:

$$(91,35) \to (56,35) \to (21,35) \to (21,14) \to (7,14) \to (7,7)$$

Die **Rekursionstiefe** eines terminierenden Arguments x einer Prozedur ist die Tiefe des Rekursionsbaums für x. Beispielsweise hat das Argument 4 der Prozedur *fib* die Rekursionstiefe 3. Das Argument $(91,35)$ der Prozedur *gcd* hat die Rekursionstiefe 5.

Zu einer Prozedur $p : X \to Y$ kann man eine Prozedur $X \to \mathscr{T}(X)$ angeben, die für terminierende Argumente x von p den Rekursionsbaum für x und p liefert. Hier ist eine solche Prozedur für *fib*:

$$\begin{array}{l} \mathit{fibT} : \mathbb{N} \to \mathscr{T}(\mathbb{N}) \\ \mathit{fibT}\ n\ =\ \text{if } n < 2 \text{ then } (n, []) \text{ else } (n, [\mathit{fibT}(n-1), \mathit{fibT}(n-2)]) \end{array}$$

Aufgabe 9.9 Realisieren Sie die Prozedur *fibT* in Standard ML.

Aufgabe 9.10 Zu einer linear-rekursiven Prozedur $p : X \to Y$ kann man eine Prozedur $X \to \mathscr{L}(X)$ angeben, die für terminierende Argumente x von p die Rekursionsfolge für x und p liefert. Schreiben Sie eine solche Prozedur für *euclid*. Realisieren Sie die Prozedur in Standard ML.

Aufgabe 9.11 Zu einer Prozedur $p : X \to Y$ kann man Prozeduren $X \to \mathbb{N}$ angeben, die für terminierende Argumente x von p die Größe und die Tiefe des Rekursionsbaums für x und p liefern. Schreiben Sie solche Prozeduren für *gcd*. Realisieren Sie die Prozeduren in Standard ML.

Aufgabe 9.12 Sei eine Prozedur mit der folgenden Rekursionsfunktion gegeben:

$$\lambda\, x \in \mathbb{Z}.\, \text{if } x < 0 \text{ then } \langle x-5 \rangle \text{ else if } x < 4 \text{ then } \langle\rangle \text{ else } \langle x-3,\, x-2 \rangle$$

a) Geben Sie den Argumentbereich der Prozedur an.
b) Geben Sie den Rekursionsbaum für das Argument 8 an.
c) Geben Sie den Definitionsbereich der Prozedur an.

9.5 Rekursionsrelationen

Unter einem **Rekursionsschritt** einer Prozedur verstehen wir ein Paar (x, x'), das aus einem Argument x und einem Folgeargument x' von x besteht. Die Menge aller Rekursionsschritte einer Prozedur bezeichnen wir als die **Rekursionsrelation** der Prozedur. Die Rekursionsrelation R einer Prozedur p kann mithilfe der Rekursionsfunktion r von p beschrieben werden:

$$R = \{(x, x') \mid x \in \mathit{Dom}\, r \ \wedge\ x' \in \mathit{Com}\,(r\,x)\}$$

Hier sind die Rekursionsrelationen für einige unserer Beispielprozeduren:

$$\begin{aligned}
\mathit{abs} &: \emptyset \\
\mathit{fac} &: \{(n, n-1) \mid n \in \mathbb{N},\ n > 0\} \\
\mathit{fib} &: \{(n, n') \in \mathbb{N}^2 \mid 0 \le n-2 \le n' \le n-1\} \\
\mathit{gcd} &: \{((x,y),\ (x,\ y-x)) \mid (x,y) \in \mathbb{N}^2,\ 0 < x < y\} \cup \\
&\quad\ \{((x,y),\ (x-y,\ y)) \mid (x,y) \in \mathbb{N}^2,\ x > y > 0\}
\end{aligned}$$

Mithilfe der Rekursionsrelation können die Terminierungseigenschaften einer Prozedur analysiert werden:

Satz 9.2 (Terminierung) *Sei p eine Prozedur $X \to Y$ mit der Rekursionsrelation R. Dann gilt:*

1. *p terminiert für ein Argument x genau dann, wenn R für x terminiert.*
2. *p terminiert genau dann, wenn R terminiert.*
3. *$\mathit{Dom}\ p = \{x \in X \mid R \text{ terminiert für } x\}$*

Beweis Behauptung (2) und (3) folgen sofort aus (1). Behauptung (1) ist zumindest für linear-rekursive Prozeduren offensichtlich. Für baumrekursive Prozeduren folgt (1) mit Königs Lemma (ein grundlegendes Resultat der Mengenlehre). ∎

Proposition 9.3 *Eine Prozedur terminiert für ein Argument x genau dann, wenn sie für alle Folgeargumente von x terminiert.*

Sei $p : X \to Y$ eine Prozedur. Dann heißt eine Funktion $f \in X \to \mathbb{N}$ **natürliche Terminierungsfunktion** für p, wenn für jeden Rekursionsschritt (x, x') von p gilt: $f\,x > f\,x'$. Durch die Angabe einer Terminierungsfunktion können wir beweisen, dass eine Prozedur für alle Argumente terminiert:

Proposition 9.4 *Wenn für eine Prozedur eine natürliche Terminierungsfunktion existiert, dann terminiert die Prozedur für alle Argumente.*

Beweis Sei p eine Prozedur mit der Rekursionsrelation R. Aus der natürlichen Terminierungsfunktion für p ergibt sich eine natürliche Terminierungsfunktion für R. Also terminiert R gemäß Proposition 8.12 auf S. 174, und p gemäß dem Terminierungssatz 9.2. ∎

Hier sind natürliche Terminierungsfunktionen für unsere Beispielprozeduren:

$$\begin{aligned}
\mathit{fac} &: \lambda\, n \in \mathbb{N}.\ n \\
\mathit{fib} &: \lambda\, n \in \mathbb{N}.\ n \\
\mathit{euclid} &: \lambda\,(x,y) \in \mathbb{N}^2.\ y \\
\mathit{gcd} &: \lambda\,(x,y) \in \mathbb{N}_+^2.\ x + y
\end{aligned}$$

Für die Prozedur *fac'* gibt es keine Terminierungsfunktion, da sie nicht für alle Argumente terminiert. Informell würde man die Terminierung der obigen Prozeduren im Einklang mit den angegebenen Terminierungsfunktionen wie folgt begründen:

- *fac* und *fib* terminieren, weil ihre Argumente bei jedem Rekursionsschritt kleiner werden.
- *euclid* terminiert, weil das zweite Argument bei jedem Rekursionsschritt kleiner wird.
- *gcd* terminiert, weil die Summe der Argumente bei jedem Rekursionsschritt kleiner wird.

Aufgabe 9.13 Geben Sie die Rekursionsrelation der Prozedur *fac'* an.

Aufgabe 9.14 Sei die folgende Prozedur gegeben:

$p : \mathbb{Z}^2 \to \mathbb{Z}$
$p(x, y) =$ if $x < y$ then $p(x, y-1)$ else
if $x > y$ then $p(x-1, y)$ else x

a) Geben Sie die Rekursionsfolge und die Rekursionstiefe für p und $(-2, 1)$ an.
b) Geben Sie die Rekursionsfunktion und die Rekursionsrelation von p an.
c) Geben Sie eine natürliche Terminierungsfunktion für p an.

9.6 Ergebnisfunktionen

Prozeduren stellen Algorithmen für die Berechnung von Funktionen dar. Gegeben eine Prozedur p, bezeichnen wir die Funktion

$$\{(x, y) \mid x \in \mathit{Dom}\ p \text{ und die Anwendung von } p \text{ auf } x \text{ liefert } y\}$$

als die **Ergebnisfunktion** von p. Außerdem sagen wir, dass eine Prozedur p **eine Funktion** f **berechnet**, wenn $\mathit{Dom}\ f \subseteq \mathit{Dom}\ p$ und p für jedes Argument $x \in \mathit{Dom}\ f$ das Ergebnis $f\,x$ liefert.

Eine Prozedur berechnet eine Funktion f also genau dann, wenn f eine Teilmenge der Ergebnisfunktion der Prozedur ist. Umgekehrt können wir die Ergebnisfunktion einer Prozedur als die größte Funktion charakterisieren, die von einer Prozedur berechnet wird.

Proposition 9.5 *Sei p eine Prozedur, die eine Funktion f berechnet. Dann ist f die Ergebnisfunktion von p genau dann, wenn $\mathit{Dom}\ f = \mathit{Dom}\ p$.*

Proposition 9.6 *Wenn eine Prozedur p eine Prozedur q erweitert, dann berechnet p alle Funktionen, die q berechnet.*

Beweis Folgt aus dem Erweiterungssatz 9.1 (3). ∎

Wir erklären anhand von Beispielen, was es bedeutet, dass **eine Funktion eine Gleichung erfüllt**. Als Funktion wählen wir $f = \lambda n{\in}\mathbb{N}.\, n$. Offensichtlich erfüllt f die Gleichungen $f0 = 0$ und $f3 = f2 + f1$. Dagegen erfüllt f die Gleichungen $f2 = f1 + f3$ und $f(-1) = f(-1)$ nicht. Etwas allgemeiner können wir sagen, dass f die Anwendungsgleichungen von *fac* für $x \in \{1,2\}$ erfüllt. Außerdem erfüllt f die Anwendungsgleichungen von *fib* für $x \in \{0,1\}$. Eine Funktion f kann eine Gleichung nur dann für ein x erfüllen, wenn sowohl die Funktion als auch die Gleichung für x definiert sind.

Proposition 9.7 *Sei p eine Prozedur und f eine Funktion. Dann erfüllt f die definierende Gleichung von p für x genau dann, wenn f die Anwendungsgleichung von p für x erfüllt.*

Satz 9.8 (Ergebnis) *Sei p eine Prozedur $X \to Y$. Dann ist die Ergebnisfunktion von p eine Funktion Dom $p \to Y$, die die definierenden Gleichungen von p für alle $x \in$ Dom p erfüllt.*

Beweis Es genügt zu zeigen, dass die Ergebnisfunktion jede Gleichung jedes Ausführungsprotokolls für eine Anwendung px mit $x \in Dom\ p$ erfüllt. Das gelingt durch Induktion über die Länge der Ausführungsprotokolle (induktive Beweise behandeln wir in § 10.1). Wir verzichten auf eine detaillierte Darstellung des Beweises. ∎

Als Beispiel für die Anwendung des Ergebnissatzes betrachten wir die Prozedur *fac*. Da *fac* für alle Argumente terminiert, sagt uns der Satz, dass die Ergebnisfunktion von *fac* eine Funktion $\mathbb{N} \to \mathbb{N}$ ist. Der Einfachheit halber bezeichnen wir die Ergebnisfunktion auch mit *fac*. Gemäß dem Ergebnissatz und Proposition 9.7 gelten die Anwendungsgleichungen der Prozedur auch für die Ergebnisfunktion. Damit können wir die Gültigkeit der Gleichung $fac\,4 = 24$ wie folgt beweisen:

$fac\,4 = 4 \cdot fac\,3$	Anwendungsgleichung für $fac\,4$
$= 4 \cdot 3 \cdot fac\,2$	Anwendungsgleichung für $fac\,3$
$= 12 \cdot 2 \cdot fac\,1$	Anwendungsgleichung für $fac\,2$
$= 24 \cdot 1 \cdot fac\,0$	Anwendungsgleichung für $fac\,1$
$= 24 \cdot 1$	Anwendungsgleichung für $fac\,0$
$= 24$	

Da es bequem ist, werden wir die Ergebnisfunktion einer Prozedur meistens mit demselben Namen wie die Prozedur bezeichnen, so wie wir es gerade für *fac* getan haben. Es muss dann aus dem Kontext ermittelt werden, ob die Prozedur oder die Funktion gemeint ist.

Manchmal wird eine Funktion als die Ergebnisfunktion einer rekursiven Prozedur definiert. Man sagt dann, dass die Funktion **rekursiv definiert** ist. Typische Beispiele sind die Ergebnisfunktionen der Prozeduren *fac* und *fib* (S. 180), die als **Fakultätsfunktion** und **Fibonacci-Funktion** bezeichnet werden.[2]

[2] Die Fibonacci-Funktion kann mithilfe der Binetschen Formel auch ohne Rekursion beschrieben werden. Allerdings sind dafür Wurzeln und reelle Zahlen erforderlich.

Zwei Prozeduren heißen **semantisch äquivalent**, wenn sie denselben Argumentbereich und dieselbe Ergebnisfunktion haben. Die Ergebnisbereiche semantisch äquivalenter Prozeduren dürfen verschieden sein.

Proposition 9.9 *Wenn zwei Prozeduren sich wechselseitig erweitern, dann sind sie semantisch äquivalent.*

Beweis Folgt aus dem Erweiterungssatz 9.1. ∎

Die Umkehrung der Proposition gilt nicht. In der Tat gibt es viele semantisch äquivalente Prozeduren, die sich nicht wechselseitig erweitern. Betrachten Sie die folgenden Prozeduren:

$p : \mathbb{N} \to \mathbb{N}$
$pn = 0$

$q : \mathbb{N} \to \mathbb{N}$
$qn = \text{if } n = 0 \text{ then } 0 \text{ else } q(n-1)$

Aufgabe 9.15 Beweisen Sie, dass die Ergebnisfunktion f der Prozedur *fib* die Gleichung $2 \cdot f(n+1) = f(n+3) - fn$ für alle $n \in \mathbb{N}$ erfüllt.

Aufgabe 9.16 Machen Sie sich klar, dass die Prozeduren *fac* und *fac'* dieselbe Ergebnisfunktion haben.

Aufgabe 9.17 Machen Sie sich mithilfe der folgenden Beispiele klar, dass aus der Ergebnisfunktion einer Prozedur nicht ermittelt werden kann, welchen Argument- und Ergebnisbereich die Prozedur hat und ob sie rekursiv ist.

a) Geben Sie eine Prozedur $\mathbb{N} \to \mathbb{N}$ an, die die Ergebnisfunktion $\emptyset$ hat.

b) Geben Sie eine rekursive Prozedur $\mathbb{N} \to \mathbb{N}$ an, die die Funktion $\lambda\, n \in \mathbb{N}.0$ berechnet.

9.7 Der Korrektheitssatz

Mit dem Korrektheitssatz können wir beweisen, dass eine Prozedur eine gegebene Funktion berechnet.

Satz 9.10 (Korrektheit) *Eine Prozedur p berechnet eine Funktion f, wenn die folgenden Bedingungen erfüllt sind:*

1. *Dom $f \subseteq$ Dom p.*
2. *f erfüllt die definierenden Gleichungen von p für alle $x \in$ Dom f.*

Beweis Seien p und f wie verlangt gegeben und sei $x \in$ *Dom* f. Da *Dom* $f \subseteq$ *Dom* p, terminiert p für x mit einem Wert y. Es genügt zu zeigen, dass $fx = y$. Da p für x terminiert,

existiert ein Ausführungsprotokoll für px, das das Ergebnis y liefert. Da f die definierenden Gleichungen von p für alle $z \in Dom\, f$ erfüllt, erfüllt f auch die Anwendungsgleichungen von p für alle $z \in Dom\, f$. Also bekommen wir einen Beweis für $fx = y$, wenn wir in dem Ausführungsprotokoll p durch f ersetzen. ■

Der Korrektheitssatz hat viele Anwendungen. Wir beginnen mit zwei Korrektheitsbeweisen für endrekursive Hilfsprozeduren.

9.7.1 Endrekursive Bestimmung von Potenzen

Wir betrachten die endrekursive Prozedur

$$p : \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$$
$$p(a, x, 0) = a$$
$$p(a, x, n) = p(a \cdot x, x, n-1) \quad \text{für } n > 0$$

und die Funktion

$$f \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$$
$$f(a, x, n) = a \cdot x^n$$

Wir wollen zeigen, dass p die Funktion f berechnet.

Offensichtlich ist $\lambda(a, x, n) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{N}.\, n$ eine natürliche Terminierungsfunktion für p. Also gilt $Dom\, p = Dom\, f$. Außerdem erfüllt f die definierenden Gleichungen von p für alle $(a, x, n) \in Dom\, f$:

$$f(a, x, 0) = a \cdot x^0 = a$$
$$f(a, x, n) = a \cdot x^n = a \cdot x \cdot x^{n-1} = f(a \cdot x, x, n-1) \quad \text{für } n > 0$$

Also folgt mit dem Korrektheitssatz, dass p die Funktion f berechnet. Mit Proposition 9.5 auf S. 188 folgt zudem, dass f die Ergebnisfunktion von p ist.

Aufgabe 9.18 Schreiben Sie in Standard ML eine Prozedur $power : int * int \to int$, die mithilfe einer endrekursiven Hilfsprozedur für x und $n \geq 0$ die Potenz x^n bestimmt.

9.7.2 Endrekursive Bestimmung von Fakultäten

Wir betrachten die endrekursive Prozedur

$$p : \mathbb{N}^2 \to \mathbb{N}$$
$$p(a, 0) = a$$
$$p(a, n) = p(a \cdot n, n-1) \quad \text{für } n > 0$$

und die Funktion

$$f \in \mathbb{N}^2 \to \mathbb{N}$$
$$f(a,n) = a \cdot fac\, n$$

Dabei bezeichnet *fac* die Ergebnisfunktion der Prozedur *fac*. Wir wollen zeigen, dass p die Funktion f berechnet.

Offensichtlich ist $\lambda(a,n) \in \mathbb{N}^2.\, n$ eine natürliche Terminierungsfunktion für p. Also gilt $Dom\, p = Dom\, f$. Um zu beweisen, dass p die Funktion f berechnet, müssen wir gemäß des Korrektheitssatzes nur noch zeigen, dass f die definierenden Gleichungen von p für $(a,n) \in \mathbb{N}^2$ erfüllt:

$$f(a,0) = a$$
$$f(a,n) = f(a \cdot n, n-1) \quad \text{für } n > 0$$

Gemäß der Definition von f genügt es zu zeigen, dass die Ergebnisfunktion der Prozedur *fac* die folgenden Gleichungen für $a, n \in \mathbb{N}$ erfüllt:

$$a \cdot fac\, 0 = a$$
$$a \cdot fac\, n = a \cdot n \cdot fac(n-1) \quad \text{für } n > 0$$

Gemäß dem Ergebnissatz 9.8 erfüllt die Ergebnisfunktion von *fac* die definierenden Gleichungen der Prozedur *fac*. Mit den definierenden Gleichungen der Prozedur *fac* lassen sich die obigen Gleichungen für die Ergebnisfunktion sofort zeigen:

$a \cdot fac\, 0 = a \cdot 1 = a$	erste definierende Gleichung
$a \cdot fac\, n = a \cdot n \cdot fac(n-1)$ für $n > 0$	zweite definierende Gleichung

Aufgabe 9.19 Schreiben Sie in Standard ML eine Prozedur $fac : int \to int$, die mithilfe einer endrekursiven Hilfsprozedur für $n \geq 0$ die n-te Fakultät bestimmt.

Aufgabe 9.20 Zeigen Sie, dass die Prozedur

$$p : \mathbb{N} \to \mathbb{N}$$
$$pn = \text{if } n < 1 \text{ then } 1 \text{ else } p(n-1) + 2n + 1$$

die Funktion $\lambda\, n \in \mathbb{N}.\, (n+1)^2$ berechnet.

Aufgabe 9.21 Beweisen Sie, dass die Prozedur

$$p : \mathbb{Z}^2 \to \mathbb{Z}$$
$$p(x,y) = \text{ if } x < y \text{ then } p(x, y-1) \text{ else}$$
$$\text{if } x > y \text{ then } p(x-1, y) \text{ else } x$$

die Funktion $\lambda(x,y) \in \mathbb{Z}^2.\, \min\{x,y\}$ berechnet.

Aufgabe 9.22 Beweisen Sie, dass die Prozedur

$p : \mathbb{Z}^2 \to \mathbb{Z}$
$p(x,y) =$ if $x < y$ then $p(x+1, y)$ else
if $x > y$ then $p(x, y+1)$ else x

die Funktion $\lambda\,(x,y) \in \mathbb{Z}^2.\ \max\{x,y\}$ berechnet.

Aufgabe 9.23 Beweisen Sie, dass die Prozedur

$p : \mathbb{N}^2 \to \mathbb{N}$
$p(n,x) =$ if $n^2 \leq x$ then $p(n+1, x)$ else $n-1$

die Funktion $\lambda\,(n,x) \in \mathbb{N}^2.\ \max\{n-1, \lfloor\sqrt{x}\rfloor\}$ berechnet.

Aufgabe 9.24 Beweisen Sie, dass die Prozedur

$p : \mathbb{N}^2 \to \mathbb{N}$
$p(x,y) =$ if $x = 0$ then y else
if $y = 0$ then x else
if $x \leq y$ then $p(x-1, y+1)$ else $p(x+1, y-1)$

die Funktion $\lambda\,(x,y) \in \mathbb{N}^2.\ x + y$ berechnet.

9.8 Größte gemeinsame Teiler

Wir wollen jetzt beweisen, dass die Prozedur *gcd* aus Abbildung 9.1 auf S. 180 den größten gemeinsamen Teiler zweier Zahlen berechnet. Dazu definieren wir zunächst eine Funktion, die den größten gemeinsamen Teiler zweier Zahlen liefert und beweisen dann mit dem Korrektheitssatz, dass die Prozedur *gcd* die Funktion berechnet.

Die **Teiler einer natürlichen Zahl** $x \in \mathbb{N}$ definieren wir wie folgt:

$$T(x) := \{k \in \mathbb{N}_+ \mid \exists n \in \mathbb{N}:\ x = k \cdot n\}$$

Für $x > 0$ gilt $\{1, x\} \subseteq T(x) \subseteq \{1, \ldots, x\}$. Die Funktion, die den **größten gemeinsamen Teiler** zweier positiver ganzer Zahlen liefert, definieren wir wie folgt:

$ggt \in \mathbb{N}_+^2 \to \mathbb{N}_+$
$ggt(x,y) = \max(T(x) \cap T(y))$

Wir zeigen jetzt mithilfe des Korrektheitssatzes, dass die Prozedur *gcd* die Funktion *ggt* berechnet. Zunächst stellen wir mithilfe der natürlichen Terminierungsfunktion $\lambda\,(x,y) \in \mathbb{N}_+^2.\ x + y$ fest, dass *gcd* terminiert. Damit bleibt noch zu zeigen, dass *ggt* die definierenden Gleichungen von *gcd* für alle $x, y \in \mathbb{N}_+$ erfüllt:

$ggt(x,x) = x$
$ggt(x,y) = ggt(x-y, y)$ für $x > y$
$ggt(x,y) = ggt(x, y-x)$ für $x < y$

Die erste Gleichung ergibt sich gemäß der Definition von *ggt* wie folgt:

$$ggt(x,x) = \max(T(x) \cap T(x)) = \max(T(x)) = x$$

Die Gültigkeit der zweiten und dritten Gleichung ergibt sich gemäß der Definition von *ggt* aus der Gültigkeit der Gleichung

$$T(x) \cap T(y) = T(x-y) \cap T(y) \quad \text{für } x > y$$

Die Gültigkeit dieser Gleichung beweisen wir, indem wir zeigen, dass jedes Element der linken Menge ein Element der rechten Menge ist und dass umgekehrt jedes Element der rechten Menge ein Element der linken Menge ist. Sei also $x > y$.

Sei $k \in T(x) \cap T(y)$. Dann existieren $n, n' \in \mathbb{N}$ mit $x = kn$ und $y = kn'$. Also $x - y = k(n - n')$. Da $x - y > 0$ und $k > 0$ folgt $n - n' > 0$. Also $k \in T(x-y)$. Also $k \in T(x-y) \cap T(y)$.

Sei $k \in T(x-y) \cap T(y)$. Dann existieren $n, n' \in \mathbb{N}$ mit $x - y = kn$ und $y = kn'$. Also $x = x - y + y = kn + kn' = k(n+n')$. Also $k \in T(x)$. Also $k \in T(x) \cap T(y)$.

Aufgabe 9.25 Zeigen Sie, dass die Prozedur *euclid* aus Abbildung 9.1 auf S. 180 die folgende Funktion berechnet:

$ggt' \in \mathbb{N}^2 \to \mathbb{N}$
$ggt'(x,y) =$ if $x = 0$ then y else if $y = 0$ then x else $ggt(x,y)$

Beweisen Sie zunächst, dass $T(x) \cap T(y) = T(y) \cap T(x \bmod y)$ für $x \in \mathbb{N}$ und $y \in \mathbb{N}_+$ gilt.

9.9 Gaußsche Formel

Die **Gaußsche Formel**

$$0 + 1 \cdots + n = \frac{n}{2}(n+1) \quad \text{für } n \in \mathbb{N}$$

besagt, dass die Summe der natürlichen Zahlen von 0 bis n statt durch wiederholte Addition auch durch Auswerten des Ausdrucks $\frac{n}{2}(n+1)$ berechnet werden kann. Für größere n erspart die Gaußsche Formel viel Rechenarbeit.

Wir wollen zeigen, dass man die Gültigkeit der Gaußschen Formel mithilfe des Korrektheitssatzes beweisen kann. Zunächst stellen wir fest, dass die linke Seite der Gaußschen Formel auch mithilfe der Prozedur

$sum : \mathbb{N} \to \mathbb{N}$
$sum\ 0 = 0$
$sum\ n = sum(n-1) + n \quad$ für $n > 0$

beschrieben werden kann, die für n die Summe der natürlichen Zahlen von 0 bis n berechnet. Beispielsweise gilt

$$\begin{aligned} \mathit{sum}\,3 &= \mathit{sum}\,2+3 \\ &= \mathit{sum}\,1+2+3 \\ &= \mathit{sum}\,0+1+2+3 \\ &= 0+1+2+3 \end{aligned}$$

Mithilfe der Ergebnisfunktion der Prozedur *sum* können wir die Gaußsche Formel wie folgt formulieren:

$$\mathit{sum}\,n = \frac{n}{2}(n+1) \quad \text{für } n \in \mathbb{N}$$

Die Gaußsche Formel besagt also gerade, dass die Prozedur *sum* die Funktion

$$\begin{aligned} &f \in \mathbb{N} \to \mathbb{N} \\ &f\,n = \frac{n}{2}(n+1) \end{aligned}$$

berechnet. Folglich können wir die Gültigkeit der Gaußschen Formel mithilfe des Korrektheitssatzes beweisen. Dazu müssen wir zwei Dinge zeigen:

1. Die Prozedur *sum* terminiert.
2. Die Funktion f erfüllt die folgenden Gleichungen:

$$\begin{aligned} &f\,0 = 0 \\ &f\,n = f\,(n-1)+n \quad \text{für } n > 0 \end{aligned}$$

Die Terminierung von *sum* ergibt sich mit der natürlichen Terminierungsfunktion $\lambda\, n \in \mathbb{N}.\, n$. Die Gültigkeit der Gleichungen folgt mit der definierenden Gleichung der Funktion f:

$$\begin{aligned} &\frac{0}{2}(0+1) = 0 \\ &\frac{n}{2}(n+1) = (\frac{n-1}{2}+1)\,n = \frac{n-1}{2}(n-1+1)+n \quad \text{für } n > 0 \end{aligned}$$

Aufgabe 9.26 Geben Sie eine rekursive Prozedur $p : \mathbb{N}_+ \to \mathbb{N}$ an, die für $n \in \mathbb{N}_+$ die Summe $1+3+\cdots+(2n-1)$ der ungeraden Zahlen von 1 bis $2n-1$ berechnet. Beweisen Sie, dass Ihre Prozedur die Funktion $\lambda\, n \in \mathbb{N}_+.\, n^2$ berechnet.

Aufgabe 9.27 Geben Sie eine rekursive Prozedur $p : \mathbb{N} \to \mathbb{N}$ an, die für $n \in \mathbb{N}$ die Summe $0^2+1^2+\cdots+n^2$ berechnet. Beweisen Sie, dass Ihre Prozedur für alle $n \in \mathbb{N}$ das Ergebnis $\frac{n}{6}(2n^2+3n+1)$ liefert.

Aufgabe 9.28 (Geometrische Summe) Geben Sie eine rekursive Prozedur $p : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$ an, die für x und n die Summe $x^0 + x^1 + \cdots + x^n$ berechnet. Beweisen Sie, dass für $x \in \mathbb{R}$ mit $x \neq 1$ und $n \in \mathbb{N}$ gilt: $p(x,n) = (1 - x^{n+1})/(1-x)$.

Aufgabe 9.29 Sie sollen zeigen, dass die Prozeduren

$$p : \mathbb{Z} \to \mathbb{Z}$$
$$p\ x = \text{if } x < 1 \text{ then } 0 \text{ else } p(x-1) + x$$

$$q : \mathbb{Z} \to \mathbb{Z}$$
$$q\ x = \text{if } x < 1 \text{ then } 0 \text{ else } \frac{x}{2}(x+1)$$

semantisch äquivalent sind. Gehen Sie dabei wie folgt vor:

a) Geben Sie natürliche Terminierungsfunktionen für p und q an.

b) Geben Sie die Ergebnisfunktion von q an.

c) Zeigen Sie, dass die Ergebnisfunktion von q die definierende Gleichung von p für alle $x \in \mathbb{Z}$ erfüllt.

9.10 Geschachtelte Rekursion

Wir haben bisher nur Prozeduren betrachtet, deren Terminierung einfach zu zeigen war. Damit Sie sehen, dass dies nicht immer so sein muss, betrachten wir die sogenannte **Ackermann-Prozedur**:

$$\begin{aligned} &am : \mathbb{N}^2 \to \mathbb{N}_+ \\ &am(0,y) = y+1 \\ &am(x,0) = am(x-1,1) && \text{für } x > 0 \\ &am(x,y) = am(x-1,\ am(x,y-1)) && \text{für } x,y > 0 \end{aligned}$$

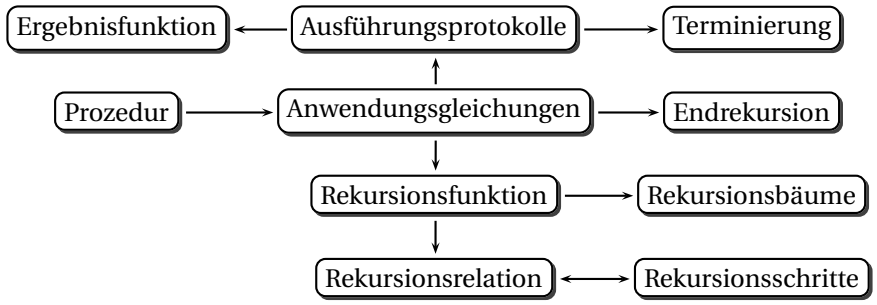
Die letzte definierende Gleichung von *am* schachtelt eine Selbstanwendung in eine Selbstanwendung, ein Phänomen, das als **geschachtelte Rekursion** bezeichnet wird. Unter anderem erhalten wir die folgende Anwendungsgleichung:

$$am(73,21) = am(72,\ am(73,20))$$

Damit ist klar, dass *am* einen Rekursionsschritt von $(73,21)$ nach $(73,20)$ hat. Falls die Anwendung $am(73,20)$ divergiert, gibt es keinen weiteren Rekursionsschritt, der von $(73,21)$ ausgeht. Falls die Anwendung $am(73,20)$ aber mit dem Ergebnis n terminiert, gibt es zusätzlich einen Rekursionsschritt von $(73,21)$ nach $(72,n)$. Wir sind also mit der Situation konfrontiert, dass wir die Rekursionsrelation von *am* nur durch Rückgriff auf die Ergebnisfunktion von *am* angeben können. Für einen Terminierungsbeweis ist das wenig hilfreich. Glücklicherweise ist aber klar, dass die Rekursionsrelation von *am* eine Teilmenge der folgenden Relation ist:

$$Ter_2 := \{((x,y),(x',y')) \in (\mathbb{N}^2)^2 \mid x > x' \vee (x = x' \wedge y > y')\}$$

Definitorische Abhängigkeiten



Bei Ter_2 handelt es sich um eine lexikalische Variante (siehe § 5.3) der terminierenden Relation *Ter* aus § 8.7. Man kann recht einfach zeigen, dass Ter_2 terminiert. Da Ter_2 die Rekursionsrelation von *am* enthält, folgt mit Propositionen 8.10 auf S. 174 und dem Terminierungssatz 9.2 auf S. 187, dass die Prozedur *am* terminiert.

Proposition 9.11 *Die Relation Ter_2 terminiert.*

Beweis Durch Widerspruch. Sei Ter_2 nicht terminierend. Dann existiert eine fortschreitende Relation $R \subseteq Ter_2$. Wir wählen das kleinste $n \in \mathbb{N}$ für das ein $m \in \mathbb{N}$ mit $(n,m) \in Ver\,R \subseteq \mathbb{N}^2$ existiert und betrachten die fortschreitende Relation $R' = \{(y,y') \mid ((n,y),(n,y')) \in R\}$. Das ist ein Widerspruch, da $R' \subseteq Ter$. ∎

Aufgabe 9.30

a) Geben Sie den Rekursionsbaum für *am* und $(2,2)$ an.

b) Geben Sie eine Prozedur an, die für $(x,y) \in \mathbb{N}^2$ die Größe des Rekursionsbaums für *am* und (x,y) liefert.

Aufgabe 9.31 Geben Sie in der Relation Ter_2 einen Pfad der Länge 5 an, der vom Knoten $(1,0)$ zum Knoten $(0,0)$ geht.

Aufgabe 9.32 Machen Sie sich die folgenden Sachverhalte klar:

a) In der Relation *Ter* gilt für jeden Knoten n, dass die von ihm ausgehenden Pfade höchstens die Länge n haben.

b) In der Relation Ter_2 gilt für jeden Knoten (x,y) mit $x > 0$, dass die Länge der von ihm ausgehenden Pfade nach oben unbeschränkt ist.

Bemerkungen

In diesem Kapitel haben wir eine Klasse mathematischer Prozeduren betrachtet, die unabhängig von einer konkreten Programmiersprache existieren. Dabei galt unser Haupt-

interesse dem Beweis von Korrektheitseigenschaften. Die dafür wichtigsten Werkzeuge haben wir mit 4 Sätzen formuliert: Erweiterungs-, Terminierungs-, Ergebnis- und Korrektheitssatz.

Unsere Konzeption mathematischer Prozeduren ist bewusst einfach gehalten. Einerseits soll die Formulierung interessanter Algorithmen und der Beweis von Korrektheitseigenschaften möglich sein. Andererseits sollen programmiersprachliche Aspekte soweit wie möglich ausgeblendet werden.

Die betrachteten mathematischen Prozeduren dienen in erster Linie dazu, Berechnungsverfahren für Funktionen zu formulieren. Darüber hinaus können bestimmte Funktionen am natürlichsten durch rekursive Prozeduren beschrieben werden. Dazu zählen die Fakultäts- und die Fibonacci-Funktion.

Die Ergebnisfunktion einer Prozedur liefert eine Black-Box-Sicht der Prozedur. Sie sagt, für welche Argumente die Prozedur terminiert und welche Ergebnisse sie jeweils liefert, aber sie verschweigt, wie die Prozedur die Ergebnisse berechnet.

Verzeichnis

Mathematische Prozeduren: Name, Argumentbereich, Ergebnisbereich, definierende Gleichungen; Argumente, Ergebnisse, Anwendungen; Prozedur $X \to Y$; Wohlgeformtheitsbedingungen, Disjunktheit und Erschöpfung.

Ausführung mathematischer Prozeduren: Anwendungsgleichungen, (terminierende) Ausführungsprotokolle; Terminierung, Divergenz, Definitionsbereich; (wechselseitige) Erweiterung.

Folgeargumente, Rekursionsschritte, Rekursionsfunktion, Rekursionsrelation.

Rekursionsbäume, Rekursionsfolgen, Rekursionstiefe.

Rekursive, linear-rekursive und baumrekursive Prozeduren.

Natürliche Terminierungsfunktionen.

Ergebnisfunktion, p berechnet f, semantische Äquivalenz.

Rekursiv definierte Funktionen, Fakultätsfunktion, Fibonacci-Funktion.

Erweiterungs-, Terminierungs-, Ergebnis- und Korrektheitssatz.

Größte gemeinsame Teiler, Euklidscher Algorithmus.

Gaußsche Formel.

Geschachtelte Rekursion, Ackermann-Prozedur.

10 Induktive Korrektheitsbeweise

Rekursion kann auch bei der Formulierung von Beweisen verwendet werden. Rekursive Beweise bezeichnet man als induktive Beweise, und statt von Rekursion spricht man bei Beweisen von Induktion.

In diesem Kapitel betrachten wir induktive Korrektheitsbeweise für rekursive Prozeduren. Neben Prozeduren für bestimmte und unbestimmte Iteration betrachten wir auch Prozeduren für Listen und Bäume.

10.1 Induktion

Bei der Konstruktion einer Prozedur für eine Funktion f können wir bei der Bestimmung von $f\,x$ davon ausgehen, dass wir f für Argumente, die kleiner als x sind, bereits bestimmen können. Wenn wir von dieser Möglichkeit Gebrauch machen, erhalten wir eine rekursive Prozedur. Bei der Konstruktion von Beweisen für allquantifizierte Aussagen $\forall\, x \in X : A(x)$ ist ein analoges Vorgehen zulässig, das **induktive Beweise** liefert: Bei der Konstruktion eines Beweises für $A(x)$ können wir davon ausgehen, dass wir $A(y)$ für Argumente y, die kleiner als x sind, bereits beweisen können. Wir betrachten ein Beispiel:

Behauptung $\forall\, n \in \mathbb{N}\colon\ n < 2^n$.

Beweis Durch Induktion über $n \in \mathbb{N}$. Wir unterscheiden drei Fälle.

Sei $n = 0$. Dann $n = 0 < 1 = 2^0 = 2^n$.

Sei $n = 1$. Dann $n = 1 < 2 = 2^1 = 2^n$.

Sei $n \geq 2$. Durch Induktion haben wir einen Beweis für $n - 1 < 2^{n-1}$. Also gilt:

$$
\begin{aligned}
2^n &= 2 \cdot 2^{n-1} \\
&> 2(n-1) && \text{Induktion für } n-1 \\
&= n + (n-2) \\
&\geq n && n \geq 2
\end{aligned}
$$

■

Das Beispiel zeigt, wie wir induktive Beweise aufschreiben werden. Gleich zu Anfang sagen wir, dass die Behauptung induktiv bewiesen werden soll. Außerdem annotieren wir alle Beweisschritte, die von der Annahme Gebrauch machen, dass die Behauptung für kleinere Argumente bewiesen werden kann, mit dem Wort Induktion.

Als zweites Beispiel betrachten wir einen Induktionsbeweis, der zeigt, dass für natürliche Zahlen Primzerlegungen existieren.

Behauptung Jede natürliche Zahl $n \geq 2$ kann als Produkt von Primzahlen dargestellt werden.

Beweis Durch Induktion über $n \geq 2$. Wir unterscheiden zwei Fälle.

n ist eine Primzahl. Dann ist die Behauptung trivial.

n ist keine Primzahl. Dann gibt es $a, b \in \mathbb{N}$ mit $n = a \cdot b$ und $2 \leq a, b < n$. Mit Induktion für a und b folgt, dass a und b als Produkte von Primzahlen darstellbar sind. Also ist $n = ab$ als Produkt von Primzahlen darstellbar. ∎

Mit Induktion kann man auch beweisen, dass eine n-elementige Menge 2^n Teilmengen hat.

Behauptung $\forall n \in \mathbb{N}\colon\ |\mathscr{P}(\{1,\ldots,n\})| = 2^n$.

Beweis Durch Induktion über $n \in \mathbb{N}$. Fallunterscheidung.

Sei $n = 0$. Dann $|\mathscr{P}(\{1,\ldots,n\})| = |\mathscr{P}(\emptyset)| = |\{\emptyset\}| = 1 = 2^0 = 2^n$.

Sei $n > 0$. Dann können wir $\mathscr{P}(\{1,\ldots,n\})$ als disjunkte Vereinigung darstellen:

$$\mathscr{P}(\{1,\ldots,n\}) = \mathscr{P}(\{1,\ldots,n-1\}) \cup \{\, X \cup \{n\} \mid X \in \mathscr{P}(\{1,\ldots,n-1\}) \,\}$$

Also $|\mathscr{P}(\{1,\ldots,n\})| = |\mathscr{P}(\{1,\ldots,n-1\})| + |\mathscr{P}(\{1,\ldots,n-1\})|$. Mit Induktion für $n-1$ folgt $|\mathscr{P}(\{1,\ldots,n-1\})| = 2^{n-1}$. Also $|\mathscr{P}(\{1,\ldots,n\})| = 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n$. ∎

Induktive Beweise beziehen sich auf die Größe der Argumente der zu beweisenden Aussage. Für jeden Induktionsbeweis muss eine terminierende Relation existieren, sodass die Größen der Argumente Knoten der Relation sind. Diese Relation bezeichnen wir als die **Induktionsrelation** des Beweises. Wenn es sich bei der Induktionsrelation so wie in den obigen Beweisen um die Relation *Ter* handelt (§ 8.7), sprechen wir von **natürlicher Induktion**. Ist die Induktionsrelation strukturell (§ 8.8), so sprechen wir von **struktureller Induktion**.

Aufgabe 10.1 Beweisen Sie die folgenden Aussagen durch Induktion:

a) $\forall n \in \mathbb{N}\colon\ 2n \leq 2^n$

b) $\forall n \in \mathbb{N}\colon\ 3n \leq 3^n$

c) $\forall n \in \mathbb{N}\ \exists k \in \mathbb{N}\colon\ n^3 - n = 3k$

Aufgabe 10.2 Dieter Schlau findet Induktionsbeweise toll. Zum Spaß versucht er zu zeigen, dass $2^n = 1$ für alle $n \in \mathbb{N}$ gilt. Und siehe da, nach einiger Zeit hat er einen Induktionsbeweis für diese Behauptung. Können Sie Dieter sagen, wo der Fehler in seinem Beweis steckt?

Behauptung: $\forall n \in \mathbb{N}\colon\ 2^n = 1$.
Beweis: Durch Induktion über $n \in \mathbb{N}$. Wir unterscheiden zwei Fälle.
Sei $n = 0$. Dann $2^n = 2^0 = 1$.
Sei $n > 0$. Mit Induktion für $n-1$ und $n-2$ folgt $2^{n-1} = 1$ und $2^{n-2} = 1$. Also $2^n = \frac{2^{n-1} \cdot 2^{n-1}}{2^{n-2}} = \frac{1 \cdot 1}{1} = 1$.

10.2 Bestimmte Iteration

Wir haben jetzt das nötige Rüstzeug, um induktive Korrektheitsbeweise für rekursive Prozeduren zu führen. Als Beispiel betrachten wir eine endrekursive Prozedur *iter*, die das Rekursionsschema der bestimmten Iteration formuliert (§ 3.4). Mit bestimmter Iteration lassen sich beispielsweise die Funktionen für Potenzen, Fakultäten und Fibonacci-Zahlen endrekursiv berechnen.

Zunächst formulieren wir *iter* als mathematische Prozedur. Die Polymorphie von *iter* modellieren wir, indem wir für jede Menge X eine eigene Prozedur *iter* definieren:

$$
\begin{aligned}
&\mathit{iter}\colon \mathbb{N} \times X \times (X \to X) \to X \\
&\mathit{iter}(0, x, f) = x \\
&\mathit{iter}(n, x, f) = \mathit{iter}(n-1,\, f\,x,\, f) \quad \text{für } n > 0
\end{aligned}
$$

Das Ausführungsprotokoll

$$
\begin{aligned}
\mathit{iter}(3,\, x,\, f) &= \mathit{iter}(2,\, f\,x,\, f) \\
&= \mathit{iter}(1,\, f(f\,x),\, f) \\
&= \mathit{iter}(0,\, f(f(f\,x)),\, f) \\
&= f(f(f\,x))
\end{aligned}
$$

zeigt die für endrekursive Prozeduren typische Form. Da *iter* sein erstes Argument bei jedem Rekursionsschritt verkleinert, terminiert *iter* für alle Argumente.

Es gibt diverse Unterschiede zwischen der mathematischen Prozedur *iter* und der konkreten Prozedur *iter* aus § 3.4. Der schwerwiegendste Unterschied besteht darin, dass die konkrete Prozedur eine Prozedur als Argument erhält, während die mathematische Prozedur eine Funktion erhält. Wenn die konkrete Prozedur eine divergierende Prozedur als Argument erhält, kann es durchaus sein, dass sie divergiert. Dagegen terminiert die mathematische Prozedur *iter* für alle Argumente.[1]

Wir formulieren jetzt eine Vertauschungseigenschaft der Prozedur *iter*, die wir in einigen der nachfolgenden Beweisen verwenden werden. Für den Beweis der Vertauschungseigenschaft benötigen wir Induktion. Damit haben wir ein erstes Beispiel für einen induktiven Korrektheitsbeweis.

[1] Beachten Sie, dass eine Funktion $X \to X$ immer für jedes Argument $x \in X$ definiert ist.

Proposition 10.1 (Vertauschung) *Sei X eine Menge. Dann gilt:*
$\forall n \in \mathbb{N}\ \forall x \in X\ \forall f \in X \to X\colon\ \mathit{iter}(n+1, x, f) = f(\mathit{iter}(n, x, f))$.

Beweis Durch Induktion über $n \in \mathbb{N}$. Fallunterscheidung.

Sei $n = 0$. Dann folgt mit der Definition von *iter*: $\mathit{iter}(n+1, x, f) = \mathit{iter}(1, x, f) = \mathit{iter}(0, fx, f) = fx = f(\mathit{iter}(0, x, f)) = f(\mathit{iter}(n, x, f))$.

Sei $n > 0$. Dann gilt:

$$\begin{aligned} \mathit{iter}(n+1, x, f) &= \mathit{iter}(n, fx, f) && \text{Definition } \mathit{iter} \\ &= f(\mathit{iter}(n-1, fx, f)) && \text{Induktion für } n-1 \\ &= f(\mathit{iter}(n, x, f)) && \text{Definition } \mathit{iter} \end{aligned}$$

■

Aufgabe 10.3 Zeigen Sie, dass *iter* und die unten definierte Prozedur *iter′* semantisch äquivalent sind.

$\mathit{iter}'\colon \mathbb{N} \times X \times (X \to X) \to X$
$\mathit{iter}'(0, x, f) = x$
$\mathit{iter}'(n, x, f) = f(\mathit{iter}'(n-1, x, f))$ für $n > 0$

10.2.1 Iterative Bestimmung von Potenzen

Eine Potenz x^n lässt sich ausgehend von 1 durch n-faches Multiplizieren mit x berechnen:

$$1 \to x^1 \to x^2 \to x^3 \to \ldots \to x^n$$

Diesen Algorithmus können wir mit *iter* und der Schrittfunktion[2] $\lambda a.\, a{\cdot}x$ formulieren. Für den Beweis der Korrektheit benötigen wir Induktion und die Vertauschungseigenschaft von *iter*.

Proposition 10.2 $\forall x \in \mathbb{Z}\ \forall n \in \mathbb{N}\colon\ x^n = \mathit{iter}(n, 1, \lambda a.\, a{\cdot}x)$.

Beweis Sei $x \in \mathbb{Z}$ und $f = \lambda a.\, a{\cdot}x$. Wir zeigen $\forall n \in \mathbb{N}\colon\ \mathit{iter}(n, 1, f) = x^n$ durch Induktion über $n \in \mathbb{N}$. Für $n = 0$ folgt die Behauptung mit der Definition von *iter*. Für $n \geq 1$ gilt:

$$\begin{aligned} \mathit{iter}(n, 1, f) &= f(\mathit{iter}(n-1, 1, f)) && \text{Proposition 10.1} \\ &= f(x^{n-1}) && \text{Induktion für } n-1 \\ &= x^{n-1} \cdot x = x^n && \text{Definition } f \end{aligned}$$

■

Aufgabe 10.4 Man kann Proposition 10.2 auch ohne die Benutzung der Vertauschungseigenschaft beweisen. Dazu zeigt man die allgemeinere Aussage $\forall x \in \mathbb{Z}\ \ \forall n \in \mathbb{N}\ \ \forall s \in \mathbb{Z}\colon\ \ \mathit{iter}(n, s, \lambda a.\, a{\cdot}x) = s{\cdot}x^n$. Beweisen Sie diese Aussage durch Induktion.

[2] Der Lesbarkeit halber verzichten wir bei der Beschreibung von Schrittfunktionen mit der Lambda-Notation auf die Angabe des Definitionsbereichs.

10.2.2 Iterative Bestimmung der Fakultäten

Sei *fac* die Ergebnisfunktion der Prozedur *fac* aus Abbildung 9.1 auf S. 180. Dann liefert die Schrittfunktion $f = \lambda(k,x).(k+1,\ k{\cdot}x)$ die folgende Kette:

$$(1,\ fac\,0) \ \to\ (2,\ fac\,1) \ \to\ (3,\ fac\,2) \ \to\ \ldots \ \to\ (n+1,\ fac\,n)$$

Also gilt $fac\,n = \#2(iter(n,(1,1),f))$ für alle $n \in \mathbb{N}$. Damit sind wir in der Lage, die Fakultäten mithilfe von *iter* endrekursiv zu bestimmen.

Aufgabe 10.5 Seien *fac*, *iter* und f wie oben gegeben. Beweisen Sie:

a) $\forall n \in \mathbb{N}\colon\ f(n+1,\ fac\,n) = (n+2,\ fac(n+1))$

b) $\forall n \in \mathbb{N}\colon\ (n+1,\ fac\,n) = iter(n,(1,1),f)$

Für den Beweis von (a) benötigen Sie nur die Definitionen von f und *fac*. Der Beweis von (b) gelingt mit Induktion sowie Teil (a) und Proposition 10.1 auf S. 202. Orientieren Sie sich am Beweis von Proposition 10.2 auf S. 202.

Aufgabe 10.6 Schreiben Sie in Standard ML eine Prozedur $fac' : int \to int$, die $fac\ n$ für $n \in \mathbb{N}$ mithilfe von *iter* endrekursiv bestimmt.

10.2.3 Iterative Bestimmung der Fibonacci-Zahlen

Sei *fib* die Ergebnisfunktion der Prozedur *fib* aus Abbildung 9.1 auf S. 180. Dann liefert die Schrittfunktion $f = \lambda(x,y).(y,\ x+y)$ die folgende Kette:

$$(fib\,0,\ fib\,1) \ \to\ (fib\,1,\ fib\,2) \ \to\ (fib\,2,\ fib\,3) \ \to\ \ldots \ \to\ (fib\,n,\ fib(n+1))$$

Folglich gilt $fib\,n = \#2(iter(n-1,(0,1),f))$ für alle $n \geq 1$. Also sind wir in der Lage, die Fibonacci-Zahlen mithilfe von *iter* endrekursiv zu bestimmen.

Aufgabe 10.7 Seien *fib*, *iter* und f wie oben gegeben. Beweisen Sie:

a) $\forall n \in \mathbb{N}_+\colon\ f(fib(n-1),\ fib\,n) = (fib\,n,\ fib(n+1))$

b) $\forall n \in \mathbb{N}_+\colon\ (fib(n-1),\ fib\,n) = iter(n-1,(0,1),f)$

Für den Beweis von (a) benötigen Sie nur die Definitionen von f und *fib*. Der Beweis von (b) gelingt mit Induktion über n sowie Teil (a) und Proposition 10.1 auf S. 202. Orientieren Sie sich am Beweis von Proposition 10.2 auf S. 202.

Aufgabe 10.8 Schreiben Sie in Standard ML eine Prozedur $fib' : int \to int$, die $fib\ n$ für $n \in \mathbb{N}$ mithilfe von *iter* endrekursiv bestimmt. Überzeugen Sie sich mit einem Interpreter davon, dass fib' für größere n sehr viel schneller rechnet als *fib*. Beginnen Sie mit $n = 40$.

Aufgabe 10.9 Die Fibonacci-Zahlen lassen sich auch ohne die höherstufige Prozedur *iter* endrekursiv bestimmen. Eine passende endrekursive Hilfsfunktion ergibt sich, indem man *iter* gemäß der für die Fibonacci-Zahlen verwendeten Schrittfunktion f spezialisiert (d.h. die Schrittfunktion nicht mehr als Argument übergibt, sondern sie in der zweiten definierenden Gleichung von *iter* direkt anwendet).

a) Konstruieren Sie eine Prozedur $fibi : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, für die $fibi(0, 1, n-1) = fib\ n$ für alle $n \geq 1$ gilt. Die ersten beiden Argumente dienen dabei als Akkus.

b) Schreiben Sie in Standard ML eine Prozedur fib', die Fibonacci-Zahlen mithilfe der endrekursiven Prozedur *fibi* berechnet.

10.3 Unbestimmte Iteration

Neben bestimmter Iteration haben wir in § 3.3.2 ein weiteres endrekursives Rekursionsschema kennengelernt, das als unbestimmte Iteration bezeichnet wird. Wir wollen jetzt eine allgemeine Formulierung der unbestimmten Iteration betrachten, mit der wir unter anderem größte gemeinsame Teiler berechnen können. Diese Version von unbestimmter Iteration ist ausdrucksstärker als bestimmte Iteration. Der Preis für diese Ausdrucksstärke besteht darin, dass unbestimmte Iteration nicht generell terminiert. Für konkrete Anwendungsfälle kann die Terminierung der unbestimmten Iteration jedoch durch Induktion bewiesen werden, womit wir wieder beim Thema dieses Kapitels wären.

Sei X eine Menge. Unbestimmte Iteration für X wendet eine **Schrittfunktion** $f \in X \to X$ solange auf einen **Anfangswert** $x \in X$ an, bis ein Wert erreicht wird, der eine **Zielbedingung** $p \in X \to \mathbb{B}$ erfüllt. Wir formulieren dieses Rekursionsschema mit einer höherstufigen und endrekursiven Prozedur *first*:

$$first\colon (X \to \mathbb{B}) \times (X \to X) \times X \to X$$
$$first(p, f, x) = \text{if } p\,x \text{ then } x \text{ else } first(p, f, f\,x)$$

Da die Terminierung von *first* in erster Linie von der als Argument übergebenen Zielbedingung abhängt, können keine allgemeinen Terminierungsaussagen über *first* gemacht werden.[3]

Unbestimmte Rekursion ist dahingehend universell, dass mit ihr die Ergebnisfunktion jeder endrekursiven Prozedur berechnet werden kann. Wir zeigen das am Beispiel der Prozedur *euclid* aus Abbildung 9.1 auf S. 180. Die Ergebnisfunktion von *euclid* können wir mit *first* wie folgt bestimmen:

$$\begin{aligned} euclid(x, y) &= \#1(first(p, f, (x, y))) \qquad \text{für } x, y \in \mathbb{N} \\ \text{wobei} \quad p &= \lambda(x, y) \in \mathbb{N}^2.\,(y = 0) \\ \text{und} \quad f &= \lambda(x, y) \in \mathbb{N}^2.\,(y,\ \text{if } y = 0 \text{ then } 0 \text{ else } x \bmod y) \end{aligned}$$

Die Korrektheit dieser Konstruktion beweisen wir durch Induktion.

Proposition 10.3 $\forall y \in \mathbb{N}\ \forall x \in \mathbb{N}\colon\ first(p, f, (x, y)) = (euclid(x, y),\ 0)$.

[3] Für Experten: Eine unbestimmte Iteration *first p f s* entspricht der folgenden While-Schleife: $x := s$; while not $p\,x$ do $x := f\,x$.

Beweis Durch Induktion über $y \in \mathbb{N}$. Sei $x \in \mathbb{N}$. Fallunterscheidung.

Sei $y = 0$. Dann folgt $\mathit{first}(p, f, (x, y)) = (x, y) = (\mathit{euclid}(x, y), 0)$ mit den definierenden Gleichungen von *first* und *euclid*.

Sei $y \geq 1$. Dann gilt:

$$\begin{aligned}
\mathit{first}(p, f, (x, y)) &= \mathit{first}(p, f, f(x, y)) && \text{Definition } \mathit{first},\ p \\
&= \mathit{first}(p, f, (y,\ x \bmod y)) && \text{Definition } f \\
&= (\mathit{euclid}(y,\ x \bmod y),\ 0) && \text{Induktion für } x \bmod y \\
&= (\mathit{euclid}(x, y),\ 0) && \text{Definition } \mathit{euclid}
\end{aligned}$$

■

Der obige Beweis bezieht sich erstmals auf eine Prozedur, deren Terminierung nicht a priori bewiesen werden kann. Stattdessen wird eine Terminierungseigenschaft der Prozedur *first* als Teilergebnis eines Induktionsbeweises etabliert. Das liegt daran, dass die zu beweisende Aussage die Aussage beinhaltet, dass die Ergebnisfunktion von *first* auf $(p, f, (x, y))$ definiert ist. Schauen Sie sich den Beweis insbesondere in Hinblick auf diese Teilaussage an.

Aufgabe 10.10 Zeigen Sie, wie man die Ergebnisfunktion der Prozedur *gcd* aus Abbildung 9.1 auf S. 180 mit *first* berechnen kann. Beweisen Sie die Korrektheit Ihrer Konstruktion.

Aufgabe 10.11 Zeigen Sie, wie man die Ergebnisfunktion der Prozedur *iter* aus § 10.2 mit *first* berechnen kann. Beweisen Sie die Korrektheit Ihrer Konstruktion.

Aufgabe 10.12 Zeigen Sie, wie man die Funktion $\lambda x \in \mathbb{N}.\ \lfloor \sqrt{x} \rfloor$ mit *first* berechnen kann. Beweisen Sie die Korrektheit Ihrer Konstruktion. Orientieren Sie sich an Aufgabe 9.23 auf S. 193.

Aufgabe 10.13 (Hardtsche Identität) Betrachten Sie die Prozedur

$$\begin{aligned}
&p\colon \mathbb{N} \to \mathbb{N} \\
&p\ 0 = 0 \\
&p\ 1 = 1 \\
&p\ n = p(n-1) + p(p(n-1) - p(n-2)) \quad \text{für } n > 1
\end{aligned}$$

Wegen der geschachtelten Rekursion lässt sich die Terminierung der Prozedur nicht ohne Information über die Ergebnisfunktion der Prozedur zeigen. Beweisen Sie durch natürliche Induktion, dass die Prozedur die Identitätsfunktion $\lambda n \in \mathbb{N}.\ n$ berechnet.

Aufgabe 10.14 Sei die folgende Prozedur gegeben:

$$\begin{aligned}
&p : \mathbb{Z}^2 \to \mathbb{Z} \\
&p(x, y) = 0 \quad \text{für } x > y \\
&p(x, y) = x + p(x+1, y) \quad \text{für } x \leq y
\end{aligned}$$

a) Geben Sie eine natürliche Terminierungsfunktion für p an.

b) Zeigen Sie durch Induktion: $\forall (x, y) \in \mathbb{Z}^2\colon\ x \leq y \Rightarrow p(x, y) = p(x, y-1) + y$.

$\mathscr{L}(X) := \{\langle\rangle\} \cup (X \times \mathscr{L}(X))$ *Listen über X*

$|\_| : \mathscr{L}(X) \to \mathbb{N}$ *Länge*

$|nil| = 0$

$|x :: xr| = 1 + |xr|$

$@ : \mathscr{L}(X) \times \mathscr{L}(X) \to \mathscr{L}(X)$ *Konkatenation*

$nil @ ys = ys$

$(x :: xr) @ ys = x :: (xr @ ys)$

$rev : \mathscr{L}(X) \to \mathscr{L}(X)$ *Reversion*

$rev\, nil = nil$

$rev\,(x :: xr) = (rev\, xr) @ [x]$

$foldl : (X \times Y \to Y) \times Y \times \mathscr{L}(X) \to Y$ *Faltung von links*

$foldl(f, y, nil) = y$

$foldl(f, y, x :: xr) = foldl(f, f(x, y), xr)$

Abbildung 10.1: *Länge, Konkatenation, Reversion und Faltung von Listen*

10.4 Listen und strukturelle Induktion

Wir wollen jetzt einige grundlegende Prozeduren für Listen betrachten. Die Terminierung dieser Prozeduren zeigen wir mit strukturellen Terminierungsfunktionen. Eigenschaften dieser Prozeduren werden wir mit struktureller Induktion beweisen.

Listen stellen wir durch geschachtelte Paare dar (§ 8.3). Abbildung 10.1 zeigt die zu betrachtenden Prozeduren. Bei den Prozeduren für Länge und Konkatenation gestatten wir uns die notationale Freiheit, die symbolischen Namen $|\_|$ und @ zu verwenden. Die Prozeduren für Länge, Konkatenation und Reversion fassen wir als Definitionen für diese grundlegenden Begriffe auf. Beachten Sie, dass die Prozedur *rev* mithilfe der Ergebnisfunktion der zuvor definierten Prozedur @ formuliert ist.[4]

Wir stellen uns zunächst die Frage, wie wir beweisen können, dass die Prozeduren in Abbildung 10.1 terminieren. Intuitiv scheint alles klar zu sein, da die Prozeduren bei jedem Rekursionsschritt die Länge einer ihrer Argumentlisten herabsetzen. Formal ist dieses Argument aber zumindest für die erste Prozedur unbrauchbar, da wir die Länge von Listen ja gerade mit dieser Prozedur definieren wollen. Wir lösen das Problem mithilfe von strukturellen Terminierungsfunktionen.

Sei $p : X \to Y$ eine Prozedur. Dann heißt eine Funktion f mit $Dom\, f = X$ **strukturelle**

[4] Zu Erinnerung: Wir betrachten nur Prozeduren ohne Hilfsprozeduren. Dafür können bei der Beschreibung von Prozeduren Funktionen verwendet werden.

Terminierungsfunktion für p, wenn für jeden Rekursionsschritt (x, x') von p gilt: $f\,x'$ ist eine Konstituente von $f\,x$.

Proposition 10.4 *Jede Prozedur, für die eine strukturelle Terminierungsfunktion existiert, terminiert für alle Argumente.*

Beweis Sei p eine Prozedur und f eine strukturelle Terminierungsfunktion für p. Dann können wir aus f eine strukturelle Terminierungsfunktion für die Rekursionsrelation von p erhalten. Also terminiert p gemäß Proposition 8.16 (S. 176) und dem Terminierungssatz 9.2 (S. 187). ∎

Hier sind strukturelle Terminierungsfunktionen für die Prozeduren aus Abbildung 10.1:

$$\begin{aligned} |\_| &: \lambda\, xs \in \mathscr{L}(X).\ xs \\ @ &: \lambda\,(xs, ys) \in \mathscr{L}(x)^2.\ xs \\ rev &: \lambda\, xs \in \mathscr{L}(X).\ xs \\ foldl &: \lambda\,(f, y, xs) \in (X \times Y \to Y) \times Y \times \mathscr{L}(X).\ xs \end{aligned}$$

Wir wollen jetzt die Assoziativität der Konkatenation durch strukturelle Induktion beweisen. Der Beweis erfolgt gemäß der strukturellen Terminierungsfunktion für @.

Proposition 10.5 (Assoziativität der Konkatenation) *Sei X eine Menge und seien $xs, ys, zs \in \mathscr{L}(X)$. Dann gilt:* $(xs@ys)@zs = xs@(ys@zs)$.

Beweis Seien $ys, zs \in \mathscr{L}(X)$. Wir beweisen

$$\forall\, xs \in \mathscr{L}(X):\ (xs@ys)@zs = xs@(ys@zs)$$

durch strukturelle Induktion über $xs \in \mathscr{L}(X)$. Wir unterscheiden zwei Fälle.

Sei $xs = nil$. Dann

$$\begin{aligned} (xs@ys)@zs &= ys@zs && \text{Definition @} \\ &= xs@(ys@zs) && \text{Definition @} \end{aligned}$$

Sei $xs = x :: xr$. Dann

$$\begin{aligned} (xs@ys)@zs &= (x :: (xr@ys))@zs && \text{Definition @} \\ &= x :: ((xr@ys)@zs) && \text{Definition @} \\ &= x :: (xr@(ys@zs)) && \text{Induktion für } xr \\ &= xs@(ys@zs) && \text{Definition @} \end{aligned}$$

∎

Aufgabe 10.15 Geben Sie für die Prozedur *rev* die Rekursionsfunktion, die Rekursionsrelation sowie eine natürliche Terminierungsfunktion an.

Aufgabe 10.16 Sei X eine Menge. Beweisen Sie mit struktureller Induktion, dass für alle Listen $xs, ys \in \mathscr{L}(X)$ gilt:

a) $|xs@ys| = |xs| + |ys|$

b) $|rev\, xs| = |xs|$

Aufgabe 10.17 Sei X eine Menge. Beweisen Sie mit struktureller Induktion, dass für alle Listen $xs, ys \in \mathscr{L}(X)$ gilt:

a) $xs@nil = xs$

b) $rev(xs@ys) = rev\, ys \,@\, rev\, xs$

c) $rev(rev\, xs) = xs$

Aufgabe 10.18 Die Länge von Listen lässt sich mit einer endrekursiven Prozedur bestimmen, die die folgende Funktion berechnet:

$$leni \in \mathbb{N} \to \mathscr{L}(X) \to \mathbb{N}$$
$$leni\ a\ xs = a + |xs|$$

Konstruieren Sie eine endrekursive Prozedur, die die Funktion *leni* berechnet und beweisen Sie die Korrektheit Ihrer Prozedur.

Aufgabe 10.19 Listen lassen sich mit einer endrekursiven Prozedur reversieren, die die folgende Funktion berechnet:

$$revi \in \mathscr{L}(X) \to \mathscr{L}(X) \to \mathscr{L}(X)$$
$$revi\ xs\ ys = (rev\ ys) @ xs$$

Konstruieren Sie eine endrekursive Prozedur, die die Funktion *revi* berechnet und beweisen Sie die Korrektheit Ihrer Prozedur. Verwenden Sie dabei die Assoziativität der Konkatenation (Proposition 10.5).

Aufgabe 10.20 Seien X, Y Mengen und f eine Funktion $X \times Y \to Y$. Beweisen Sie durch strukturelle Induktion über xs, dass gilt:

$$\forall xs \in \mathscr{L}(X)\ \ \forall xs' \in \mathscr{L}(X)\ \ \forall y \in Y:$$
$$foldl(f, y, xs@xs') = foldl(f,\ foldl(f, y, xs),\ xs')$$

10.5 Verstärkung der Korrektheitsaussage

In § 4.4 haben wir gesehen, dass die Länge von Listen mit *foldl* bestimmt werden kann. Wir wollen jetzt die Korrektheit dieses Vorgehens beweisen.

Sei X eine Menge und f die Funktion $\lambda(x, a) \in X \times \mathbb{N}.\, a + 1$. Wir würden gerne die folgende Aussage beweisen:

$$\forall xs \in \mathscr{L}(X):\ |xs| = foldl(f, 0, xs)$$

Wenn Sie versuchen, diese Aussage durch strukturelle Induktion über *xs* zu beweisen, werden Sie scheitern, da Sie Induktion für $foldl(f, 1, xr)$ benötigen. Der Beweis gelingt jedoch, wenn wir die Korrektheitsaussage wie folgt verallgemeinern:

$$\forall xs \in \mathscr{L}(X)\ \ \forall n \in \mathbb{N}\colon\ \ |xs| + n = foldl(f, n, xs)$$

Dabei ist wesentlich, dass die Quantifizierung für n der Quantifizierung für *xs* untergeordnet ist. Das versetzt uns in die Lage, Induktion für jedes $n \in \mathbb{N}$ zu verwenden.

Beachten Sie, dass die gerade gezeigte Verstärkung der Korrektheitsaussage der Einführung einer Hilfsprozedur mit einem Akku entspricht. Die verstärkte Aussage besagt nämlich, dass mit *foldl* die Funktion *leni* aus Aufgabe 10.18 auf S. 208 berechnet werden kann.

Proposition 10.6 *Sei* $f = \lambda(x, a) \in X \times \mathbb{N}.\, a + 1$. *Dann:*
$\forall xs \in \mathscr{L}(X)\ \ \forall n \in \mathbb{N}\colon\ \ |xs| + n = foldl(f, n, xs)$.

Beweis Durch strukturelle Induktion über $xs \in \mathscr{L}(X)$. Sei $n \in \mathbb{N}$. Fallunterscheidung.

Sei $xs = nil$. Dann folgt $foldl(f, n, xs) = n = |xs| + n$ mit den definieren Gleichungen von *foldl* und |_|.

Sei $xs = x :: xr$. Dann:

$$\begin{aligned}
foldl(f, n, xs) &= foldl(f,\ f(x, n),\ xr) && \text{Definition } foldl\\
&= foldl(f,\ n+1,\ xr) && \text{Definition } f\\
&= |xr| + n + 1 && \text{Induktion für } xr\\
&= |xs| + n && \text{Definition } |_|
\end{aligned}$$

■

Aufgabe 10.21 Schauen Sie sich Aufgabe 10.4 auf S. 202 an. Auch dort geht es um die Verstärkung einer Korrektheitsaussage.

Aufgabe 10.22 In § 4.4 haben Sie gelernt, dass Listen mit *foldl* reversiert werden können. Jetzt können Sie die Korrektheit dieses Vorgehens beweisen.
Sei X eine Menge und sei f die Funktion $\lambda(x, xs) \in X \times \mathscr{L}(X).\, x :: xs$. Für die Korrektheit der Reversion mit *foldl* muss die Gültigkeit der Aussage $\forall xs \in \mathscr{L}(X)\colon\ rev\ xs = foldl(f, nil, xs)$ gezeigt werden.
Suchen Sie eine geeignete Verstärkung dieser Korrektheitsaussage und beweisen Sie die Gültigkeit der Verstärkung durch strukturelle Induktion über *xs*. Orientieren Sie sich an der Funktion *revi* aus Aufgabe 10.19.

$\mathscr{T} := \mathscr{L}(\mathscr{T})$ *Bäume*

$s : \mathscr{T} \to \mathbb{N}_+$ *Größe*

$s[t_1, \ldots, t_n] = \text{if } n = 0 \text{ then } 1 \text{ else } 1 + s\,t_1 + \cdots + s\,t_n$

$b : \mathscr{T} \to \mathbb{N}_+$ *Breite*

$b[t_1, \ldots, t_n] = \text{if } n = 0 \text{ then } 1 \text{ else } b\,t_1 + \cdots + b\,t_n$

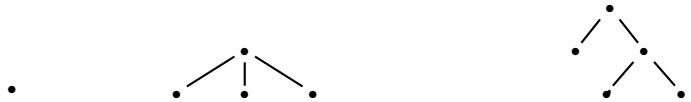
$d : \mathscr{T} \to \mathbb{N}$ *Tiefe*

$d[t_1, \ldots, t_n] = \text{if } n = 0 \text{ then } 0 \text{ else } 1 + \max\{d\,t_1, \ldots, d\,t_n\}$

Abbildung 10.2: *Größe, Breite und Tiefe von Bäumen*

10.6 Größenverhältnisse in Bäumen

Wir wollen jetzt Prozeduren für reine Bäume betrachten. Reine Bäume stellen wir durch geschachtelte Listen dar:



[] [[], [], []] [[], [[], []]]

Abbildung 10.2 zeigt drei Prozeduren, die die Größe, die Breite und die Tiefe von Bäumen liefern. Neben der primären Baumrekursion verwenden diese Prozeduren eine sekundäre Listenrekursion, die mit der Punkt-Punkt-Notation „$\cdots$“ formuliert ist. In Standard ML kann die Listenrekursion mit einer Faltungsprozedur realisiert werden (§ 7.4). In den Anwendungsgleichungen der Prozeduren ist die sekundäre Listenrekursion nicht mehr sichtbar:

$$\begin{aligned} s[t_1, t_2] &= 1 + s\,t_1 + s\,t_2 \\ b[t_1, t_2, t_3] &= b\,t_1 + b\,t_2 + b\,t_3 \\ d[t_1, t_2] &= 1 + \max\{d\,t_1, d\,t_2\} \end{aligned}$$

Dementsprechend ist nur die primäre Baumrekursion in den Rekursionsfunktionen und Rekursionsrelationen der Prozeduren sichtbar. Alle drei Prozeduren haben die gleiche Rekursionsfunktion:

$$\lambda\,[t_1, \ldots, t_n] \in \mathscr{T}\,.\,\langle t_1, \ldots, t_n \rangle$$

Damit folgt die Terminierung der Prozeduren in allen drei Fällen mit der strukturellen Terminierungsfunktion $\lambda\, t \in \mathscr{T}\,.\, t$.

Wir wollen jetzt Aussagen über die Größenverhältnisse in balancierten Binärbäumen beweisen (§ 7.7). Die **Menge $\mathscr{B} \subseteq \mathscr{T}$ der balancierten Binärbäume** definieren wir durch Rekursion:

1. $[] \in \mathscr{B}$.
2. Wenn $t_1 \in \mathscr{B}$, $t_2 \in \mathscr{B}$ und $d\,t_1 = d\,t_2$, dann $[t_1, t_2] \in \mathscr{B}$.

Die Breite eines balancierten Binärbaums hängt exponentiell von seiner Tiefe ab:

Proposition 10.7 (Breite versus Tiefe) $\forall\, t \in \mathscr{B}\colon \quad b\,t = 2^{d\,t}$.

Beweis Durch strukturelle Induktion über $t \in \mathscr{B}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $b\,t = 1 = 2^{d\,t}$ gemäß der Definition von b und d.

Sei $t = [t_1, t_2]$. Dann gilt:

$$
\begin{aligned}
b\,t &= b\,t_1 + b\,t_2 && \text{Definition } b\\
&= 2^{d\,t_1} + 2^{d\,t_2} && \text{Induktion für } t_1 \text{ und } t_2\\
&= 2 \cdot 2^{d\,t_1} && t \text{ balanciert, also } d\,t_1 = d\,t_2\\
&= 2^{1+d\,t_1}\\
&= 2^{1+\max\{d\,t_1,\, d\,t_2\}} && t \text{ balanciert, also } d\,t_1 = d\,t_2\\
&= 2^{d\,t} && \text{Definition } d
\end{aligned}
$$

■

Auch die Größe eines balancierten Binärbaums hängt exponentiell von seiner Tiefe ab:

Proposition 10.8 (Größe versus Tiefe) $\forall\, t \in \mathscr{B}\colon \quad s\,t = 2^{d\,t+1} - 1$.

Beweis Durch strukturelle Induktion über $t \in \mathscr{B}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $s\,t = 1 = 2^{d\,t+1} - 1$ gemäß der Definition von b und d.

Sei $t = [t_1, t_2]$. Dann gilt:

$$
\begin{aligned}
s\,t &= 1 + s\,t_1 + s\,t_2 && \text{Definition } s\\
&= 1 + 2^{d\,t_1+1} - 1 + 2^{d\,t_2+1} - 1 && \text{Induktion für } t_1 \text{ und } t_2\\
&= 2 \cdot 2^{d\,t_1+1} - 1 && t \text{ balanciert, also } d\,t_1 = d\,t_2\\
&= 2^{1+d\,t_1+1} - 1\\
&= 2^{1+\max\{d\,t_1,\, d\,t_2\}+1} - 1 && t \text{ balanciert, also } d\,t_1 = d\,t_2\\
&= 2^{d\,t+1} - 1 && \text{Definition } d
\end{aligned}
$$

■

Schließlich wollen wir noch Bäume betrachten, bei denen jeder innere Knoten mindestens zwei Nachfolger hat. Die Menge $\mathscr{M} \subseteq \mathscr{T}$ dieser Bäume definieren wir rekursiv wie folgt:

1. $[] \in \mathscr{M}$.
2. Wenn $n \geq 2$ und $t_1, \ldots, t_n \in \mathscr{M}$, dann $[t_1, \ldots, t_n] \in \mathscr{M}$.

Offensichtlich gilt $\mathscr{B} \subseteq \mathscr{M}$. Die Bäume in $\mathscr{M}$ haben mehr Blätter als innere Knoten:

Proposition 10.9 (Breite versus Größe) $\forall\, t \in \mathscr{M} :\ \ 2 \cdot bt > st$.

Beweis Durch strukturelle Induktion über $t \in \mathscr{M}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $2 \cdot bt = 2 > 1 = st$ gemäß der Definition von b und s.

Sei $t = [t_1, \ldots, t_n]$ *mit* $n \geq 2$. Dann gilt:

$$\begin{aligned}
2 \cdot bt &= 2(bt_1 + \cdots + bt_n) && \text{Definition } b \\
&= 2 \cdot bt_1 + \cdots + 2 \cdot bt_n \\
&\geq (st_1 + 1) + \cdots + (st_n + 1) && \text{Induktion für } t_1, \ldots, t_n \\
&> st_1 + \cdots + st_n + 1 && n \geq 2 \\
&= st && \text{Definition } s
\end{aligned}$$

Bei den Induktionsschritten von t zu $t_1, \ldots, t_n$ haben wir von der Äquivalenz $2 \cdot bt_i > st_i \iff 2 \cdot bt_i \geq st_i + 1$ Gebrauch gemacht. ∎

Aufgabe 10.23 (Balancierte Binärbäume)

a) Wie viele Blätter hat ein balancierter Binärbaum der Tiefe 10?

b) Wie viele Knoten hat ein balancierter Binärbaum der Tiefe 10?

c) Wie viele innere Knoten hat ein balancierter Binärbaum der Tiefe n?

d) Hat ein balancierter Binärbaum der Tiefe n mehr Blätter oder mehr innere Knoten? Wie viele Knoten beträgt der Unterschied?

Aufgabe 10.24 (Induktionsloser Beweis) Proposition 10.7 besagt, dass die Prozedur b die Funktion $\lambda\, t \in \mathscr{B}.2^{dt}$ berechnet. Gemäß dem Korrektheitssatz ist das der Fall, wenn die Funktion die definierende Gleichung von b für alle $t \in \mathscr{B}$ erfüllt. Zeigen Sie, dass dies der Fall ist.

Aufgabe 10.25 (Induktionsloser Beweis) Beweisen Sie Proposition 10.8 ohne Induktion mithilfe des Korrektheitssatzes.

Aufgabe 10.26 (Binärbäume) Ein Binärbaum ist ein Baum, bei dem jeder innere Knoten genau zwei Nachfolger hat (§ 7.2).

a) Definieren Sie die Menge $\mathscr{M} \subseteq \mathscr{T}$ der Binärbäume.

b) Geben Sie einen Binärbaum an, der nicht balanciert ist.

c) Beweisen Sie: $\forall t \in \mathscr{M} : \; bt \leq 2^{dt}$.

d) Beweisen Sie: $\forall t \in \mathscr{M} : \; st \leq 2^{dt+1} - 1$.

Aufgabe 10.27 (Balancierte Ternärbäume) Unter einem Ternärbaum wollen wir einen Baum verstehen, bei dem jeder innere Knoten genau drei Nachfolger hat.

a) Definieren Sie die Menge $\mathscr{M} \subseteq \mathscr{T}$ der balancierten Ternärbäume.

b) Beweisen Sie: $\forall t \in \mathscr{M} : \; bt = 3^{dt}$.

c) Beweisen Sie: $\forall t \in \mathscr{M} : \; st = \frac{1}{2}(3^{dt+1} - 1)$.

10.7 Binäre Charakterisierung von Bäumen

Gemäß unserer Definition ist ein reiner Baum gerade die Liste seiner Unterbäume. Daraus folgt, dass ein Baum entweder die leere Liste ist, oder aber ein Paar $t :: t'$ aus zwei Bäumen, wobei t der erste Unterbaum und t' die Liste der restlichen Unterbäume ist. Da umgekehrt auch jedes Paar $t :: t'$ von Bäumen wieder ein Baum ist, haben wir eine binäre Charakterisierung von Bäumen.

Mit der binären Charakterisierung von Bäumen können wir binärrekursive Prozeduren konstruieren, die die Größe, Breite und Tiefe von Bäumen berechnen. Dabei entfällt die Sekundärrekursion für die Unterbaumlisten. Betrachten Sie dazu die folgenden Gleichungen für die Ergebnisfunktion der Prozedur s:

$$\begin{aligned} s\,nil &= 1 \\ s(t :: t') &= st + st' \end{aligned}$$

Die Gültigkeit dieser Gleichungen kann leicht mit den definierenden Gleichungen von s gezeigt werden. Da die Gleichungen disjunkt und erschöpfend sind, eignen sie sich als Rekursionsgleichungen für die Berechnung von s. Die auf diesen Gleichungen beruhende Prozedur terminiert, da ihre Rekursionsrelation strukturell ist. Damit sind wir in der Lage, die Größe von Bäumen mit einer binärrekursiven Prozedur zu bestimmen:

$$\begin{aligned} &size : \mathscr{T} \to \mathbb{N}_+ \\ &size\ nil = 1 \\ &size(t :: t') = size\ t + size\ t' \end{aligned}$$

Es ist jetzt auch möglich, eine endrekursive Prozedur anzugeben, mit der die Größe von Bäumen bestimmt werden kann. Dazu betrachten wir die Funktion

$$\begin{aligned} &s' \in \mathbb{N} \times \mathscr{T} \to \mathbb{N}_+ \\ &s'(a, t) = a + st \end{aligned}$$

Für diese Funktion können wir die folgenden Rekursionsgleichungen angeben:

$$\begin{aligned} s'(a, nil) &= a + 1 \\ s'(a, nil :: t) &= s'(a + 1, t) \\ s'(a, (t :: t') :: t'') &= s'(a, t :: t' :: t'') \end{aligned}$$

Die Gültigkeit dieser Gleichungen kann mit den obigen Gleichungen für s gezeigt werden. Für die dritte Gleichung geht das wie folgt:

$$\begin{aligned}
s'(a,(t :: t') :: t'') &= a + s((t :: t') :: t'') && \text{Definition } s' \\
&= a + s(t :: t') + st'' && \text{Gleichung für } s \\
&= a + st + st' + st'' && \text{Gleichung für } s \\
&= a + st + s(t' :: t'') && \text{Gleichung für } s \\
&= a + s(t :: t' :: t'') && \text{Gleichung für } s \\
&= s'(a, t :: t' :: t'') && \text{Definition } s'
\end{aligned}$$

Also bleibt nur die Terminierung der Rekursionsgleichungen für s' zu zeigen. Das gelingt mit der natürlichen Terminierungsfunktion

$$\lambda\,(a,t) \in \mathbb{N} \times \mathscr{T}.\ 2 \cdot st - |t|$$

Dabei bezeichnet $|t|$ die Länge der Liste t.

Aufgabe 10.28 Schreiben Sie eine endrekursive Prozedur $size' : \mathbb{N} \times \mathscr{T} \to \mathbb{N}_+$, sodass $size'(0,t)$ die Größe eines Baums t liefert.

Aufgabe 10.29 Schreiben Sie eine endrekursive Prozedur $size' : int \to tree \to int$ in Standard ML, sodass $size'\ 0\ t$ die Größe eines Baums t liefert.

Aufgabe 10.30 Schreiben Sie eine binärrekursive Prozedur $depth : \mathscr{T} \to \mathbb{N}$, die die Tiefe eines Baums liefert. Beweisen Sie die Korrektheit Ihrer Prozedur.

Aufgabe 10.31 Schreiben Sie eine binärrekursive Prozedur $breadth : \mathscr{T} \to \mathbb{N}_+$, die die Breite eines Baums liefert. Beweisen Sie die Korrektheit Ihrer Prozedur.

Aufgabe 10.32 Schreiben Sie eine endrekursive Prozedur $breadth' : \mathbb{N} \times \mathscr{T} \to \mathbb{N}_+$, die die Funktion $\lambda\,(a,t) \in \mathbb{N} \times \mathscr{T}.\ a + bt$ berechnet. Beweisen Sie die Korrektheit Ihrer Prozedur.

Aufgabe 10.33 Konstruieren Sie eine endrekursive Prozedur $\mathbb{N} \times \mathscr{L}(\mathbb{N} \times \mathscr{T}) \to \mathbb{N}$, die für $(0,[(0,t)])$ die Tiefe des Baums t liefert. Beweisen Sie die Korrektheit Ihrer Prozedur. Hilfestellung: Schreiben Sie die Prozedur so, dass sie für ein Argument $(a,[(k_1,t_1),\ldots,(k_n,t_n)])$ das Ergebnis $\max\{a,\ k_1 + d\,t_1,\ \ldots\ ,\ k_n + d\,t_n\}$ liefert.

Bemerkungen

Die Praxis zeigt, dass die Korrektheitseigenschaften rekursiver Prozeduren oft mit induktiven Beweisen gezeigt werden müssen. Neben den höherstufigen Prozeduren für bestimmte und unbestimmte Iteration haben wir erstmals mathematische Prozeduren für Listen und Bäume betrachtet. Für Bäume haben wir eine binäre Charakterisierung kennengelernt, die ohne Sekundärrekursion für die Unterbaumlisten auskommt.

Induktive Beweise sind rekursive Beweise. Dadurch ergibt sich eine gewisse Analogie zwischen Prozeduren und Beweisen. Anders als bei Prozeduren muss die Rekursion bei Beweisen jedoch stets terminieren. Das wird durch eine terminierende Relation gewährleistet, gemäß der die Induktionsschritte erfolgen müssen. Wenn diese Induktionsrelation strukturell ist, spricht man von struktureller Induktion, und wenn es sich um die Relation *Ter* handelt, spricht man von natürlicher Induktion. Natürliche Induktion wurde früher entdeckt als die allgemeine Form der Induktion, die wir der deutschen Mathematikerin Emmy Noether (1882-1935) verdanken. Die allgemeine Form der Induktion wird auch als **wohlfundierte Induktion** bezeichnet.

Damit ein induktiver Beweis möglich wird, ist es oft erforderlich, die zu beweisende Aussage zu verstärken. Diese Verstärkung der zu beweisenden Aussage entspricht auf der prozeduralen Seite der Generalisierung der zu berechnenden Funktion um Akkus.

Verzeichnis

Induktive Beweise; Induktionsrelation; natürliche und strukturelle Induktion.

Bestimmte und unbestimmte Iteration; iterative Bestimmung von Fakultäten, Fibonacci-Zahlen und größten gemeinsamen Teilern.

Listen und strukturelle Terminierungsfunktionen; strukturelle Induktion; Verstärkung der Korrektheitsaussage.

Größenverhältnisse in Bäumen; binäre Charakterisierung von Bäumen.

11 Laufzeit rekursiver Prozeduren

Prozeduren, die zu lange rechnen, sind in der Praxis wertlos. Wir entwickeln jetzt eine Theorie, mit der wir fundierte Aussagen über die Laufzeit von Prozeduren machen können.

11.1 Laufzeitfunktionen

Auch in diesem Kapitel betrachten wir mathematische Prozeduren. Zunächst definieren wir, was wir unter der Laufzeit einer Prozedur für ein Argument verstehen wollen. Wir beginnen mit einer vorläufigen Definition, die wir später verfeinern werden.

Die **Laufzeit einer Prozedur für ein Argument** x ist die Größe des Rekursionsbaums für x.

Die Laufzeit einer Prozedur für ein Argument x ist also die Anzahl der Anwendungen der Prozedur, die insgesamt ausgeführt werden müssen, um das Ergebnis der Prozedur für x zu bestimmen. Nicht-rekursive Prozeduren haben gemäß dieser Definition für alle Argumente die Laufzeit 1.

Die obige Definition der Laufzeit ist nur dann realistisch, wenn die Zeiten für die Anwendungen der von der Prozedur verwendeten Funktionen vernachlässigt werden können. Das ist natürlich nur für einfache Funktionen möglich, die ohne Rekursion bestimmt werden können. Später werden wir die Definition der Laufzeit so erweitern, dass die Zeiten für die Anwendungen von Funktionen als Nebenkosten eingebracht werden können.

Um vergleichende Aussagen über die Laufzeiten von Prozeduren machen zu können, müssen wir von der konkreten Form der Argumente abstrahieren. Dazu gruppieren wir die Argumente nach Größe und betrachten für jede Größe die maximale Laufzeit für Argumente dieser Größe. Bei der präzisen Umsetzung dieser Idee müssen wir auf einige Details achten.

Eine **Größenfunktion** für eine terminierende Prozedur $p : X \to Y$ ist eine natürliche Terminierungsfunktion $s \in X \to \mathbb{N}$ für p, die die folgende Bedingung erfüllt: $\forall n \in \mathbb{N}\ \exists k \in \mathbb{N}\ \forall x \in X$: wenn $sx = n$, dann ist die Laufzeit von p für x kleiner als k. Die Zahl sx bezeichnen wir als die **Größe** von x.

Die auf den ersten Blick kompliziert aussehende Bedingung für Größenfunktionen ist in Wirklichkeit einfach. Sie verlangt, dass die Laufzeit der Prozedur für alle Argumente

einer festen Größe nach oben beschränkt ist. Mithilfe einer Größenfunktion können wir für eine Prozedur eine Funktion definieren, die zu jeder Größe die maximale Laufzeit für die Argumente dieser Größe liefert.

Sei s eine Größenfunktion für eine Prozedur $p : X \to Y$. Die **Laufzeitfunktion von p gemäß** s ist die Funktion $r \in \mathbb{N} \to \mathbb{N}_+$, die für jedes $n \in \mathbb{N}$ die maximale Laufzeit liefert, die p für Argumente der Größe n benötigt. Damit r auch für Größen n definiert ist, für die keine Argumente existieren, vereinbaren wir:

1. $r0 = 1$, falls es keine Argumente der Größe 0 gibt.
2. $r\,n = r(n-1)$, falls $n > 0$ und es keine Argumente der Größe n gibt.

Wir sagen, dass die Laufzeit einer Prozedur gemäß einer Größenfunktion **uniform** ist, wenn für jede Größe gilt, dass die Prozedur für alle Argumente dieser Größe die gleiche Laufzeit hat.

Wenn die Laufzeit einer Prozedur nicht uniform ist, liegt der Definition der Laufzeitfunktion die sogenannte **Worst-Case-Annahme** zugrunde: $r\,n$ ist die maximale Laufzeit von p für Argumente der Größe n.

11.2 Beispiele

Um für eine Prozedur eine angemessene Laufzeitfunktion zu erhalten, muss eine angemessene Größenfunktion zugrunde gelegt werden. Wir erläutern die Wahl der Größenfunktion und die Bestimmung der Laufzeitfunktion anhand einiger Beispiele.

11.2.1 Konkatenation von Listen

Bei linearer Rekursion ist es oft sinnvoll, die Größenfunktion so zu wählen, dass die Rekursionsschritte die Argumentgröße um 1 verringern. Wir betrachten die Prozedur @, die zwei Listen konkateniert:

$$@ : \mathscr{L}(X) \times \mathscr{L}(X) \to \mathscr{L}(X)$$
$$nil@ys = ys$$
$$(x :: xr)@ys = x :: (xr@ys)$$

Da die Prozedur linear-rekursiv ist, nimmt der Rekursionsbaum die Form einer Folge an:

$$([1,2,3],\ ys) \to ([2,3],\ ys) \to ([3],\ ys) \to ([],\ ys)$$

Bei jedem Rekursionsschritt wird die Länge der ersten Argumentliste um eins verringert. Die zweite Argumentliste hat keinen Einfluss auf die Laufzeit, da sie nur weitergereicht wird. Daher wählen wir die Größenfunktion $\lambda(xs, ys).|xs|$. Offensichtlich hat @ für alle Argumente der Größe n uniform die Laufzeit $n+1$. Also hat @ die Laufzeitfunktion $\lambda n.\, n+1$.

Aufgabe 11.1 Geben Sie die Rekursionsfolge und die Laufzeit der Prozedur @ für das Argument $([1,2],\ [3,4,5,6])$ an.

Aufgabe 11.2 Geben Sie für die folgende Prozedur eine Größenfunktion und die entsprechende Laufzeitfunktion an:

$$\begin{aligned}&\mathit{revi} : \mathscr{L}(X) \times \mathscr{L}(X) \to \mathscr{L}(X)\\ &\mathit{revi}(\mathit{xs},\ \mathit{nil}) = \mathit{xs}\\ &\mathit{revi}(\mathit{xs},\ y :: \mathit{yr}) = \mathit{revi}(y :: \mathit{xs},\ \mathit{yr})\end{aligned}$$

Aufgabe 11.3 Geben Sie für die Fakultätsprozedur eine Größenfunktion und die entsprechende Laufzeitfunktion an:

$$\begin{aligned}&\mathit{fac} : \mathbb{N} \to \mathbb{N}\\ &\mathit{fac}\ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \mathit{fac}(n-1)\end{aligned}$$

11.2.2 Faltung von Listen

Wir betrachten die höherstufige und endrekursive Prozedur *foldl*, die Listen von links her faltet:

$$\begin{aligned}&\mathit{foldl} : (X \times Y \to Y) \times Y \times \mathscr{L}(X) \to Y\\ &\mathit{foldl}(f, s, \mathit{nil}) \ = \ s\\ &\mathit{foldl}(f,\ s,\ x :: \mathit{xr}) \ = \ \mathit{foldl}(f,\ f(x,s),\ \mathit{xr})\end{aligned}$$

Bei jedem Rekursionsschritt wird die Länge der Argumentliste um eins verringert. Die restlichen Argumente haben keinen Einfluss auf die Laufzeit, da sie nur weitergereicht werden. Daher wählen wir die Größenfunktion $\lambda(f, s, \mathit{xs}).\,|\mathit{xs}|$. Da *foldl* für alle Argumente der Größe n uniform die Laufzeit $n+1$ hat, hat *foldl* die Laufzeitfunktion $\lambda n.\, n+1$.

Aufgabe 11.4 Geben Sie die Rekursionsfolge und die Laufzeit von *foldl* für das Argument $(f,\ 5,\ [2,6,3])$ mit $f = \lambda(x,y).\, x+y$ an.

Aufgabe 11.5 Geben Sie für die Prozedur *iter* eine Größenfunktion und die entsprechende Laufzeitfunktion an:

$$\begin{aligned}&\mathit{iter} : \mathbb{N} \times X \times (X \to X) \to X\\ &\mathit{iter}(0, x, f) \ = \ x\\ &\mathit{iter}(n, x, f) \ = \ \mathit{iter}\,(n-1, f\,x, f) \quad \text{für } n > 0\end{aligned}$$

11.2.3 Elementtest für Listen

Bisher haben wir nur Prozeduren mit uniformer Laufzeit betrachtet. Um ein Beispiel für eine Prozedur mit nicht-uniformer Laufzeit zu geben, betrachten wir eine Prozedur, die testet, ob ein Wert in einer Liste vorkommt:

$$\begin{aligned}&\mathit{member} : \mathbb{Z} \times \mathscr{L}(\mathbb{Z}) \to \mathbb{B}\\ &\mathit{member}(x, \mathit{nil}) \ = \ 0\\ &\mathit{member}(x, y :: \mathit{yr}) \ = \ \text{if } x = y \text{ then } 1 \text{ else } \mathit{member}(x, \mathit{yr})\end{aligned}$$

Die Rekursion dieser Prozedur läuft über die Argumentliste. Daher wählen wir die Größenfunktion $\lambda(x, xs).|xs|$. Da *member* nur dann rekurriert, wenn der zu testende Wert verschieden vom Kopf der Argumentliste ist, kann die Laufzeit von *member* für ein Argument der Größe n jeden Wert zwischen 1 und $n+1$ annehmen. Also hat *member* gemäß der Worst-Case-Annahme die Laufzeitfunktion $\lambda n.\, n+1$.

Aufgabe 11.6 Betrachten Sie die Prozedur *insert*, die ein neues Element in eine Liste einfügt (Sortieren durch Einfügen, § 5.1):

$$
\begin{aligned}
&\mathit{insert} : \mathbb{Z} \times \mathscr{L}(\mathbb{Z}) \to \mathscr{L}(\mathbb{Z}) \\
&\mathit{insert}(x, \mathit{nil}) = [x] \\
&\mathit{insert}(x, y :: yr) = \text{if } x \le y \text{ then } x :: y :: yr \text{ else } y :: \mathit{insert}(x, yr)
\end{aligned}
$$

a) Wählen Sie eine Größenfunktion für *insert*.

b) Geben Sie für die Größe 4 ein Argument mit minimaler Laufzeit und ein Argument mit maximaler Laufzeit an (in Bezug auf Argumente der Größe 4).

c) Welche minimale und maximale Laufzeit hat *insert* für Argumente der Größe n?

d) Geben Sie die Laufzeitfunktion von *insert* an.

Aufgabe 11.7 Geben Sie eine Prozedur $p : \mathscr{L}(\mathbb{N}) \to \{0\}$ wie folgt an:

1. $\lambda xs.|xs|$ eine zulässige Größenfunktion für p.
2. Bezüglich dieser Größenfunktion hat p die Laufzeitfunktion $\lambda n.\, n+1$.
3. Für alle $n \in \mathbb{N}$ gibt es eine Liste xs der Länge n, für die p die Laufzeit 1 hat.

Aufgabe 11.8 Betrachten Sie die folgende Prozedur:

$$
\begin{aligned}
&p : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
&p(0, k) = 0 \\
&p(n, k) = p(n-1, 0) + \cdots + p(n-1, k) \quad \text{für } n > 0
\end{aligned}
$$

a) Geben Sie die Laufzeit von p für das Argument $(1, k)$ an.

b) Offensichtlich ist $\lambda(n, k).\, n$ eine natürliche Terminierungsfunktion für p. Machen Sie sich klar, dass $\lambda(n, k).\, n$ aber keine Größenfunktion für p ist.

c) Geben Sie eine Größenfunktion für p an.

11.3 Rekursive Darstellung von Laufzeitfunktionen

Die Laufzeitfunktion einer Prozedur lässt sich in der Regel recht einfach durch eine rekursive Prozedur darstellen. Dagegen kann es sehr schwer oder sogar unmöglich sein, die Laufzeitfunktion ohne Rekursion zu beschreiben. Als Beispiel betrachten wir eine binärrekursive Prozedur, die für $n \in \mathbb{N}$ den balancierten Binärbaum der Tiefe n auf naive

Weise bestimmt:

$$\begin{array}{l} \mathit{ntree} : \mathbb{N} \to \mathcal{T} \\ \mathit{ntree}\, 0 \;=\; \mathit{nil} \\ \mathit{ntree}\, n \;=\; [\mathit{ntree}(n-1),\ \mathit{ntree}(n-1)] \quad \text{für } n > 0 \end{array}$$

Wir halten gleich fest, dass es sich hier um eine pathologische Prozedur handelt, die unnötig viel Laufzeit benötigt. Eine semantisch äquivalente Prozedur, die sehr viel schneller rechnet, kann mit einem Let-Ausdruck oder noch besser mit *iter* realisiert werden (Aufgabe 11.10).

Als Größenfunktion wählen wir $\lambda\, n.n$. Damit hat *ntree* uniforme Laufzeit. Die Laufzeitfunktion von *ntree* können wir durch die folgende Prozedur beschreiben:

$$\begin{array}{l} r : \mathbb{N} \to \mathbb{N}_+ \\ r0 = 1 \\ r\,n = 1 + r(n-1) + r(n-1) \quad \text{für } n > 0 \end{array}$$

Damit haben wir eine rekursive Darstellung der Laufzeitfunktion von *ntree*. Eine rekursionsfreie Darstellung der Laufzeitfunktion ist nicht offensichtlich. Wenn wir uns aber klar machen, dass die Prozedur r gerade die Größe des balancierten Binärbaums der Tiefe n berechnet, legt uns Proposition 10.8 (S. 211) nahe, dass $r\,n = 2^{n+1} - 1$ für $n \in \mathbb{N}$ gilt. Mit dem Korrektheitssatz können wir diese Vermutung leicht beweisen. Damit haben wir mit $\lambda\, n \in \mathbb{N}.2^{n+1} - 1$ eine **explizite Darstellung** der Laufzeitfunktion von *ntree*.

Für praktische Zwecke genügt in aller Regel eine rekursive Beschreibung der Laufzeitfunktion. Die rekursive Darstellung hat zudem den Vorteil, dass eventuelle Nebenkosten für die Anwendung von Funktionen problemlos eingebracht werden können (§ 11.7).

Aufgabe 11.9 Geben Sie eine rekursive Beschreibung der Laufzeitfunktion der Prozedur @ gemäß der in § 11.2.1 angegebenen Größenfunktion an.

Aufgabe 11.10 Geben Sie eine zu *ntree* semantisch äquivalente Prozedur $\mathit{tree} : \mathbb{N} \to \mathcal{T}$ an, die die Laufzeitfunktion $\lambda\, n.n+1$ hat. Schreiben Sie die Prozeduren *tree* und *ntree* in Standard ML und überzeugen Sie sich mit einem Interpreter davon, dass *tree* sehr viel schneller rechnet als *ntree*.

11.4 Laufzeiten und Komplexitäten

Um die Laufzeit einer Prozedur beurteilen zu können, genügt es, die sogenannte Komplexität ihrer Laufzeitfunktion zu kennen. Die Komplexität beschreibt ohne unnötige Details, wie schnell die Laufzeit mit größer werdenden Argumenten wächst (z.B. linear, quadratisch, exponentiell). Die möglichen Komplexitäten sind geordnet, wobei größere Komplexität größeres Wachstum bedeutet. Was unter einer Komplexität genau zu

Anzahl Prozeduraufrufe (PA)	Ausführungszeit bei 10^9 PA pro Sekunde	
	in Sekunden	etwa
10^4	10^{-5}	10 Mikrosekunden
10^6	10^{-3}	1 Millisekunde
10^9	10^0	1 Sekunde
10^{11}	10^2	2 Minuten
10^{13}	10^4	3 Stunden
10^{14}	10^5	1 Tag
10^{15}	10^6	2 Wochen
10^{16}	10^7	4 Monate
10^{17}	10^8	3 Jahre
10^{19}	10^{10}	3 Jahrhunderte
10^{20}	10^{11}	3 Jahrtausende
10^{21}	10^{12}	ewig

Abbildung 11.1: *Ausführungszeit in Abhängigkeit von der Anzahl der PA*

verstehen ist, werden wir in § 11.5 definieren. Vorerst genügt uns ein ungefähres Verständnis von Komplexitäten. Hier sind Beispiele für Laufzeitfunktionen und ihre Komplexitäten:

$\lambda n.n$	lineare Komplexität
$\lambda n.n^2$	quadratische Komplexität
$\lambda n.n^3$	kubische Komplexität
$\lambda n.2^n$	exponentielle Komplexität

Um eine Intuition für diese Komplexitäten zu erhalten, wollen wir konkrete Zahlen betrachten. Abbildung 11.1 zeigt die Ausführungszeit einer Berechnung in Abhängigkeit von der Anzahl der auszuführenden Prozeduraufrufe, wobei eine Ausführungsgeschwindigkeit von 10^9 Prozeduraufrufen pro Sekunde zugrunde gelegt ist. Diese Geschwindigkeit setzt einen Computer voraus, der mit mindestens 10 GHz getaktet ist.

Abbildung 11.2 zeigt, wie die maximale Ausführungszeit für die obigen Laufzeitfunktionen von der Argumentgröße abhängt. Dabei legen wir wieder eine Ausführungsgeschwindigkeit von 10^9 Prozeduraufrufen pro Sekunde zugrunde. Die Zahlen zeigen, dass die Komplexität der Laufzeitfunktion (linear, quadratisch, kubisch, exponentiell) einen gravierenden Einfluss auf die Ausführungszeit der Prozedur hat. Beispielsweise benötigt

Größe	Laufzeitfunktion			
n	linear n	quadratisch n^2	kubisch n^3	exponentiell 2^n
	Ausführungszeit bei 10^9 Prozeduraufrufen pro Sekunde			
10^3	10^{-6} Sekunden	10^{-3} Sekunden	1 Sekunde	ewig
10^4	10^{-5} Sekunden	10^{-1} Sekunden	20 Minuten	ewig
10^5	10^{-4} Sekunden	10 Sekunden	10 Tage	ewig
10^6	10^{-3} Sekunden	20 Minuten	30 Jahre	ewig
10^7	10^{-2} Sekunden	1 Tag	ewig	ewig

Abbildung 11.2: *Ausführungszeit in Abhängigkeit von der Argumentgröße*

$O(1)$	konstante Komplexität
$O(\log n)$	logarithmische Komplexität
$O(n)$	lineare Komplexität
$O(n \cdot \log n)$	linear-logarithmische Komplexität
$O(n^2)$	quadratische Komplexität
$O(n^3)$	kubische Komplexität
$O(b^n)$	exponentielle Komplexität ($b > 1$)

Abbildung 11.3: *Die wichtigsten prozeduralen Komplexitäten*

eine Prozedur für Argumente der Größe 10^4 höchstens 10 Mikrosekunden, wenn wir die lineare Laufzeitfunktion zugrunde legen. Wenn wir die kubische Laufzeitfunktion zugrunde legen, bekommen wir bereits eine Ausführungszeit von 20 Minuten. Und mit der exponentiellen Laufzeitfunktion schnellt die Ausführungszeit ins Unermeßliche.

Unter der **Komplexität einer Prozedur** verstehen wir die Komplexität ihrer Laufzeitfunktion. Statt von der Komplexität einer Prozedur spricht man auch von der **Laufzeit einer Prozedur**.

In der Praxis treten nur wenige Komplexitäten für Prozeduren auf. Abbildung 11.3 zeigt die Bezeichnungen und Sprechweisen für die wichtigsten prozeduralen Komplexitäten. Die angegebenen Komplexitäten sind linear geordnet, wobei $O(1)$ am kleinsten und $O(b^n)$ am größten ist. Prozeduren mit größerer Komplexität benötigen für größere Argumente signifikant mehr Ausführungszeit als Prozeduren mit kleinerer Komplexität.

Hier sind noch ein paar Bemerkungen zu den mit dem Großbuchstaben O beginnenden Bezeichnungen für die Komplexitäten: Eine Bezeichnung wie O(n) wird als “O von n”

gelesen. Der Buchstabe O kommt von Ordnung. Man spricht auch von der **O-Notation** für Komplexitäten. Im nächsten Abschnitt definieren wir die O-Notation präzise.

Mit den prozeduralen Komplexitäten lassen sich die in Kapitel 5 beobachteten Laufzeitunterschiede zwischen verschiedenen Prozeduren zum Reversieren und Sortieren von Listen vorhersagen (§ 5.7, § 5.4). Wir werden zeigen, dass gilt:

- Reversieren mit *foldl* hat lineare Komplexität und Reversieren mit *rev* hat quadratische Komplexität.
- Sortieren durch Mischen hat linear-logarithmische Komplexität und Sortieren durch Einfügen hat quadratische Komplexität.

11.5 Komplexität von Laufzeitfunktionen

Wir wollen jetzt die Komplexität von Laufzeitfunktionen ordentlich definieren. Dabei erweist es sich als sinnvoll, etwas allgemeiner als notwendig Funktionen des Typs $\mathbb{N} \to \mathbb{R}$ zu betrachten, die bis auf endlich viele Ausnahmen nicht-negative Ergebnisse liefern. Solche Funktionen wollen wir als **O-Funktionen** bezeichnen. Die Menge der O-Funktionen definieren wir wie folgt:

$$OF := \{f \in \mathbb{N} \to \mathbb{R} \mid \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 :\ f\,n \geq 0\}$$

Machen Sie sich klar, wie diese Definition die Eigenschaft „bis auf endlich viele Ausnahmen“ formal formuliert.

Für O-Funktionen definieren wir eine Relation, die wir als **Dominanzrelation** bezeichnen:

$$f \preceq g \quad :\Longleftrightarrow \quad \exists n_0 \in \mathbb{N}\ \exists c \in \mathbb{N}\ \forall n \geq n_0 :\ \ f\,n \leq c(g\,n)$$

Eine **Dominanz** $f \preceq g$ gilt also genau dann, wenn es einen Faktor c gibt, für den bis auf endlich viele Ausnahmen $f\,n \leq c(g\,n)$ gilt. Hier sind Beispiele für gültige Dominanzen:

$$\begin{aligned} \lambda n \in \mathbb{N}.\, 270 &\preceq \lambda n \in \mathbb{N}.\, 1 \\ \lambda n \in \mathbb{N}.\, 11n + 273 &\preceq \lambda n \in \mathbb{N}.\, n \\ \lambda n \in \mathbb{N}.\, 33n^3 + 22n^2 + 11 &\preceq \lambda n \in \mathbb{N}.\, n^3 \end{aligned}$$

Wenn $f \preceq g$ gilt, sagen wir auch, dass f **von** g **dominiert** wird, oder dass f **höchstens so komplex wie** g ist. Manchmal sagt man auch, dass f asymptotisch höchstens so stark wächst wie g, wobei asymptotisch sich auf die Maßgabe bezieht, dass endlich viele Ausnahmen zulässig sind.

Proposition 11.1 *Für alle $f, g, h \in OF$ gilt:*

1. $f \preceq f$ *(Reflexivität von $\preceq$)*
2. $f \preceq g \wedge g \preceq h \implies f \preceq h$ *(Transitivität von $\preceq$)*

Da die Dominanzrelation $\preceq$ nicht antisymmetrisch ist, kann sie von unnötigen Details abstrahieren. Beispielsweise gilt:

$$\lambda n \in \mathbb{N}.\, n^3 \;\preceq\; \lambda n \in \mathbb{N}.\, 33n^3 + 22n^2 + 11 \;\preceq\; \lambda n \in \mathbb{N}.\, n^3$$

Die **Komplexität einer O-Funktion** f definieren wir als die Menge aller O-Funktionen, die höchstens so komplex wie f sind:

$$O(f) := \{g \in OF \mid g \preceq f\}$$

Da die Komplexitäten gemäß der Teilmengenbeziehung für Mengen geordnet sind, sind wir sofort in der Lage, über größere und kleinere Komplexitäten zu reden. Beispielsweise gilt:

$$O(\lambda n.1) = O(\lambda n.133) \subset O(\lambda n.n) = O(\lambda n.7n - 26) \subset O(\lambda n.n^2)$$

Proposition 11.2 *Für alle O-Funktionen f, g gilt:*

$$O(f) \subseteq O(g) \iff f \preceq g \iff f \in O(g)$$

Beweis Die erste Äquivalenz folgt aus der Reflexivität und Transitivität der Dominanzrelation (Proposition 11.1). Die zweite Äquivalenz folgt unmittelbar aus der Definition von $O(f)$. ∎

Da der Lambda-Präfix bei $O(\lambda n \in \mathbb{N}.\, f\, n)$ die Lesbarkeit behindert und aus dem Kontext leicht ableitbar ist, lässt man ihn üblicherweise weg und schreibt kürzer $O(f\, n)$. Beispielsweise steht $O(n^2)$ für $O(\lambda n \in \mathbb{N}.\, n^2)$ und $O(2^n)$ für $O(\lambda n \in \mathbb{N}.\, 2^n)$. Außerdem definieren wir für $n \in \mathbb{N}$:

$$\log n := \text{if } n = 0 \text{ then } 0 \text{ else } \log_2 n$$

Damit sind alle Komplexitäten in Abbildung 11.3 auf S. 223 sauber definiert. Die folgende Proposition gibt die Ordnungsbeziehungen zwischen den wichtigsten Komplexitäten an:

Proposition 11.3 *Für alle $a, b, c \in \mathbb{R}$ mit $a, b, c > 1$ gilt:*

1. $O(0) \subset O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(b^n)$
2. $O(n \cdot \log n) \subset O(n^a) \subset O(n^b)$ *für* $a < b$
3. $O(n^a) \subset O(b^n) \subset O(c^n)$ *für* $b < c$

Die Komplexität einer Funktion ändert sich nicht, wenn man sie mit einem positiven Faktor skaliert oder ihr eine höchstens so komplexe Funktion hinzuaddiert:

Proposition 11.4 *Sei c eine positive reelle Zahl und seien f, g O-Funktionen mit $O(g) \subseteq O(f)$. Dann gilt:*

$$O(f\, n) = O(c \cdot f\, n) = O(f\, n + g\, n) = O(c \cdot f\, n + g\, n)$$

Mit Proposition 11.4 und den Ordnungsbeziehungen aus Proposition 11.3 haben wir Werkzeuge, mit denen wir die Komplexität von Funktionen einfach darstellen können. Hier ist ein Beispiel.

Behauptung $O(7n + \log n + 45) = O(n)$.

Beweis Gemäß Proposition 11.4 gilt $O(45) = O(1)$. Also folgt mit $O(1) \subset O(\log n)$ und Proposition 11.4 $O(\log n + 45) = O(\log n)$. Also folgt mit $O(\log n) \subset O(n)$ und Proposition 11.4 $O(7n + \log n + 45) = O(n)$. ∎

Aufgabe 11.11 Beweisen Sie die folgenden Gleichungen:

a) $O(\frac{n^2}{7} + 789\,n \cdot \log n + 45n + 77) = O(n^2)$.

b) $O(\frac{2^n}{23} + 12n^2 + 789n + 77) = O(2^n)$.

Aufgabe 11.12 Beweisen Sie Proposition 11.1.

Aufgabe 11.13 Beweisen Sie Proposition 11.4.

Aufgabe 11.14 Sei $O(1) \subseteq O(f)$. Beweisen Sie: $O(fn) \subset O(n \cdot fn)$. Hinweis: Zeigen Sie zuerst $O(fn) \subseteq O(n \cdot fn)$ und zeigen Sie dann $O(n \cdot fn)) \not\subseteq O(fn)$ durch Widerspruch. Benutzen Sie dabei jeweils Proposition 11.2.

11.6 Naive Komplexitätsbestimmung

In § 11.1 haben wir für einige Prozeduren explizite Darstellungen ihrer Laufzeitfunktionen ermittelt. Die Komplexitäten der Laufzeitfunktionen und damit die der Prozeduren ergeben sich aus diesen Darstellungen mithilfe der Propositionen 11.3 und 11.4:

Prozedur	Größenfunktion	Laufzeitfunktion	Komplexität
@	$\lambda\,(xs, ys).\,\|xs\|$	$\lambda\,n.\,n+1$	$O(n)$
foldl	$\lambda\,(f, s, xs).\,\|xs\|$	$\lambda\,n.\,n+1$	$O(n)$
member	$\lambda\,(x, xs).\,\|xs\|$	$\lambda\,n.\,n+1$	$O(n)$
ntree	$\lambda\,n.n$	$\lambda\,n.2^{n+1}-1$	$O(2^n)$

Im nächsten Abschnitt werden wir sehen, dass die Komplexität einer Laufzeitfunktion oft direkt aus ihrer rekursiven Darstellung ermittelt werden kann. Damit wird die mühsame und oft sogar unmögliche explizite Bestimmung der Laufzeitfunktion überflüssig. Eine Komplexitätsbestimmung, die die Herleitung einer expliziten Darstellung der Laufzeitfunktion beinhaltet, bezeichnen wir als naiv.

Aufgabe 11.15 Geben Sie die Komplexitäten der Prozeduren *revi*, *fac*, *iter* und *insert* aus den Aufgaben 11.2 und 11.3 (S. 219) sowie 11.5 und 11.6 (S. 219) an.

Aufgabe 11.16 Geben Sie eine baumrekursive Prozedur $\mathbb{N} \to \mathbb{N}$ an, deren Komplexität konstant ist. Hinweis: Formulieren Sie die Prozedur so, dass sie ab einer bestimmten Argumentgröße nicht mehr rekurriert. Das Beispiel zeigt, dass man pathologische Prozeduren konstanter Komplexität konstruieren kann, die für kleine Argumente sehr hohe Laufzeiten haben.

Aufgabe 11.17 Betrachten Sie die Laufzeit der folgenden Prozedur für die Größenfunktion $\lambda\, n.\, n$:

$p : \mathbb{N} \to \mathbb{N}$
$p\, n = \text{if } n < 5 \text{ then } n \text{ else } p(n-1)$

a) Geben Sie die Laufzeitfunktion der Prozedur an.

b) Geben Sie die Komplexität der Prozedur an.

c) Geben Sie die Ergebnisfunktion der Prozedur an.

d) Geben Sie die Komplexität der Ergebnisfunktion der Prozedur an.

Aufgabe 11.18 Geben Sie Komplexitäten der durch die folgenden Gleichungen rekursiv definierten Funktionen $f \in \mathbb{N} \to \mathbb{N}$ an:

a) $f\, n = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1)$

b) $f\, n = \text{if } n = 0 \text{ then } 1 \text{ else } f(n-1)$

c) $f\, n = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1) + 1$

Aufgabe 11.19 Geben Sie Komplexitäten der durch die folgenden Gleichungen definierten Prozeduren $f : \mathbb{N} \to \mathbb{N}$ an:

a) $f\, n = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1)$

b) $f\, n = \text{if } n = 0 \text{ then } 1 \text{ else } f(n-1)$

c) $f\, n = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1) + 1$

Vergleichen Sie Ihre Ergebnisse mit denen aus Aufgabe 11.18.

Einfluss der Größenfunktion

Während die Laufzeitfunktion einer Prozedur unmittelbar von den Details der gewählten Größenfunktion abhängt, ist die Komplexität der Laufzeitfunktion von diesen Details weitgehend unabhängig. Als Beispiel betrachten wir die im letzten Abschnitt analysierte Prozedur @. Die einfachst mögliche Größenfunktion für @ ist $\lambda\,(xs, ys).\, |xs|$. Sie liefert die Laufzeitfunktion $\lambda\, n.\, n+1$. Diese hat die Komplexität $O(n)$. Alternativ können wir auch die (unnötig komplizierte) Größenfunktion $\lambda\,(xs, ys).\, 3|xs|+7$ verwenden. Diese liefert eine Laufzeitfunktion, die umständlich hinzuschreiben ist, aber wieder die Komplexität $O(n)$ hat.

Wenn man die Größenfunktion grundlegend verändert, ändert sich auch die Komplexität der Laufzeitfunktion. Wenn wir beispielsweise für die Prozedur @ die Größenfunktion $\lambda\,(xs, ys).\, 2^{|xs|}$ zugrunde legen, ergibt sich eine Laufzeitfunktion mit der Komplexität $O(\log n)$.

Aufgabe 11.20 Geben Sie die Laufzeitfunktion für die Prozedur @ und die Größenfunktion $\lambda(xs, ys).3|xs| + 7$ an.

Aufgabe 11.21 Geben Sie die Laufzeitfunktion für die Prozedur @ und die Größenfunktion $\lambda(xs, ys).2^{|xs|}$ an.

11.7 Nebenkosten

Wir erweitern unser Modell jetzt so, dass auch für die Anwendung von Funktionen Laufzeit veranschlagt werden kann. Als Beispiel betrachten wir die Prozedur *rev*, die Listen mithilfe der Konkatenationsfunktion @ reversiert:

$$
\begin{aligned}
&rev : \mathscr{L}(X) \to \mathscr{L}(X)\\
&rev\ nil = nil\\
&rev(x :: xr) = rev\, xr @ [x]
\end{aligned}
$$

Da die Konkatenationsfunktion @ mit Rekursion berechnet werden muss, kann ihre Anwendung erhebliche Laufzeitkosten verursachen. Diese sogenannten **Nebenkosten** stellen wir bei der rekursiven Beschreibung der Laufzeitfunktion mithilfe einer **Kostenfunktion** $g \in \mathbb{N} \to \mathbb{N}_+$ dar:

$$
\begin{aligned}
&r \in \mathbb{N} \to \mathbb{N}_+\\
&r0 = g0\\
&r\,n = r(n-1) + g\,n \quad \text{für } n > 0
\end{aligned}
$$

Dabei fallen für Argumente der Größe n Nebenkosten der Höhe $g\,n - 1$ an. Als Größenfunktion haben wir $\lambda\, xs.|xs|$ zugrunde gelegt. Da sich die Länge der Liste beim Reversieren nicht ändert, ergibt sich mit der Laufzeitfunktion der Prozedur @ (§ 11.2.1), dass die Anwendung von @ auf *rev xr* und $[x]$ die Nebenkosten $|xr| + 1$ verursacht. Damit ergibt sich die Kostenfunktion wie folgt:

$$
\begin{aligned}
&g \in \mathbb{N} \to \mathbb{N}_+\\
&g\,n = n + 1
\end{aligned}
$$

Das mathematisch geschulte Auge erkennt jetzt, dass für alle $n \in \mathbb{N}$ gilt:

$$
\begin{aligned}
r\,n &= g0 + \cdots + g\,n\\
&= (0+1) + \cdots + (n+1)\\
&= (0 + \cdots + n) + \underbrace{(1 + \cdots + 1)}_{(n+1)\text{–mal}}\\
&= \frac{n}{2}(n+1) + (n+1) && \text{Gaußsche Formel}\\
&= \frac{1}{2}(n^2 + 3n + 2)
\end{aligned}
$$

Damit haben wir eine explizite Darstellung der Laufzeitfunktion von *rev*. Mit Proposition 11.4 (S. 225) ergibt sich, dass die Prozedur *rev* quadratische Komplexität hat.

Wir haben jetzt eine vortreffliche Erklärung für den in § 5.7 beobachteten Laufzeitunterschied zwischen der gerade analysierten Prozedur *rev* und einer mit *foldl* formulierten Reversionsprozedur: *rev* hat quadratische Komplexität und rechnet damit für größere Argumente deutlich langsamer als die mit *foldl* formulierte Reversionsprozedur, die gemäß der Analyse in § 11.2.2 lineare Komplexität hat. Die endrekursive Prozedur *revi* aus Aufgabe 11.2 (S. 219) reversiert Listen ebenfalls mit linearer Laufzeit.

Da die Laufzeit der Prozedur *rev* unnötig hoch ist, wird sie oft als **naive Reversionsprozedur** bezeichnet. Interpreter für Standard ML realisieren die vordeklarierte Prozedur *rev* natürlich mit linearer Laufzeit.

Wir merken uns, dass der Konkatenationsoperator @ in Bezug auf Effizienz mit Bedacht eingesetzt werden muss, da er mit linearer Laufzeit zu Buche schlägt.

Aufgabe 11.22 Beweisen Sie mit dem Korrektheitssatz, dass für die oben rekursiv dargestellte Laufzeitfunktion r von *rev* gilt: $\forall n \in \mathbb{N}\colon\ r\,n = \frac{1}{2}(n^2 + 3n + 2)$.

11.7.1 Beispiel: Aufteilen von Listen

Wir können jetzt auch die Laufzeit der Prozedur *split* analysieren, die beim Sortieren durch Mischen verwendet wird:

$$\begin{array}{l} split : \mathscr{L}(X) \to \mathscr{L}(X) \times \mathscr{L}(X) \\ split\ xs = foldl(\lambda\,(x,(ys,zs)).\,(zs,\ x::ys),\ (nil,nil),\ xs) \end{array}$$

Wir legen die Größenfunktion $\lambda\, xs.|xs|$ zugrunde. Da *split* nicht rekursiv ist, gilt $r = g$ und die Laufzeit ergibt sich als eins plus die für die Anwendung der Funktion *foldl* anfallenden Nebenkosten. Wenn wir diese gemäß der Analyse in § 11.2.2 veranschlagen, ergibt sich die Laufzeitfunktion von *split* wie folgt:

$$r\,n = g\,n = 1 + (n+1) = n+2$$

Also hat *split* lineare Komplexität.

11.7.2 Beispiel: Sortieren durch Einfügen

Wir sind jetzt in der Lage, die Laufzeit einer Prozedur zu analysieren, die Listen durch Einfügen sortiert:

$$\begin{array}{l} isort : \mathscr{L}(\mathbb{Z}) \to \mathscr{L}(\mathbb{Z}) \\ isort\ nil\ =\ nil \\ isort(x::xr)\ =\ insert(x,\ isort\ xr) \end{array}$$

Wir legen die Größenfunktion $\lambda\, xs.|xs|$ zugrunde. Damit ergibt sich die Laufzeitfunktion für *isort* wie folgt:

$$r0 = g0$$
$$r\,n = r(n-1) + g\,n \quad \text{für } n > 0$$

Wir nehmen an, dass die Funktion *insert* durch die Prozedur aus Aufgabe 11.6 auf S. 220 berechnet wird. Da die Laufzeit dieser Prozedur nicht uniform ist, legen wir wie üblich die Worst-Case-Annahme zugrunde. Maximale Nebenkosten ergeben sich für eine absteigend sortierte Argumentliste, da dann der Kopf jedes Mal am Ende der bereits sortierten Teilliste eingefügt werden muss. Also bekommen wir die Kostenfunktion

$$g\,n = 1 + n$$

Damit haben wir für *isort* dieselbe Kosten- und Laufzeitfunktion wie für die naive Reversionsprozedur. Also hat *isort* ebenfalls quadratische Komplexität.

Die Laufzeit von *isort* ist nicht uniform, da die für die Anwendung der Funktion *insert* anfallenden Nebenkosten nicht uniform sind. Wir wollen daher zusätzlich zu der gerade ermittelten **Worst-Case-Laufzeitfunktion** auch die **Best-Case-Laufzeitfunktion** bestimmen. Da bei einer aufsteigend sortierten Argumentliste der Kopf der Argumentliste bei jedem Rekursionsschritt am Anfang der bereits sortierten Teilliste eingefügt werden kann, bekommen wir in diesem Fall die Kostenfunktion

$$g\,n = \text{if } n = 0 \text{ then } 1 \text{ else } 2$$

Damit ergibt sich die Best-Case-Laufzeitfunktion von *isort* wie folgt:

$$r\,n = g0 + \cdots + g\,n = 1 + \underbrace{(2 + \cdots + 2)}_{n\text{-mal}} = 2n + 1$$

Also hat *isort* lineare **Best-Case-Komplexität**.

Aufgabe 11.23 Geben Sie eine explizite Darstellung der Worst-Case-Laufzeitfunktion von *isort* an.

Aufgabe 11.24 Geben Sie für eine beliebige Größe $n \in \mathbb{N}$ Argumente an, für die *isort* minimale beziehungsweise maximale Laufzeit hat.

Aufgabe 11.25 Sie sollen die Best- und die Worst-Case-Laufzeit der folgenden Prozedur bestimmen:

$$gcd : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+$$
$$gcd(x,x) = x$$
$$gcd(x,y) = gcd(x-y,\, y) \quad \text{für } x > y$$
$$gcd(x,y) = gcd(x,\, y-x) \quad \text{für } x < y$$

Dabei soll die Größenfunktion $\lambda\,(x,y).\ \max\{x,y\}$ zugrunde gelegt werden.

a) Geben Sie Argumente für die Größe $n = 4$ an, für die die Laufzeit minimal beziehungsweise maximal ist. Geben Sie die jeweiligen Rekursionsfolgen an.
b) Geben Sie für eine beliebige Größe $n \geq 1$ Argumente an, für die die Laufzeit minimal beziehungsweise maximal ist.
c) Geben Sie die Best-Case-Laufzeitfunktion und deren Komplexität an.
d) Geben Sie die Worst-Case-Laufzeitfunktion und deren Komplexität an.

Aufgabe 11.26 Sie sollen die Best- und die Worst-Case-Laufzeit der Prozedur *merge* analysieren, die beim Sortieren durch Mischen verwendet wird:

$merge : \mathscr{L}(\mathbb{Z}) \times \mathscr{L}(\mathbb{Z}) \to \mathscr{L}(\mathbb{Z})$
$merge(nil, ys) = ys$
$merge(xs, nil) = xs$
$merge(x :: xr,\ y :: yr) = \text{if } x \leq y \text{ then } x :: merge(xr,\ y :: yr) \text{ else } y :: merge(x :: xr,\ yr)$

Dabei soll die Größenfunktion $\lambda(xs, ys).\,|xs| + |ys|$ zugrunde gelegt werden.

a) Geben Sie Argumente für die Größe $n = 4$ an, für die die Laufzeit minimal beziehungsweise maximal ist. Geben Sie die jeweiligen Rekursionsfolgen an.
b) Geben Sie für eine beliebige Größe $n \geq 1$ Argumente an, für die die Laufzeit minimal beziehungsweise maximal ist.
c) Geben Sie die Best-Case-Laufzeitfunktion und deren Komplexität an.
d) Geben Sie die Worst-Case-Laufzeitfunktion und deren Komplexität an.

Aufgabe 11.27 Man kann Sortieren durch Einfügen auch wie in § 5.1 gezeigt mithilfe von *foldl* realisieren:

$isort' : \mathscr{L}(\mathbb{Z}) \to \mathscr{L}(\mathbb{Z})$
$isort'\ xs = foldl(insert, nil, xs)$

Die Prozedur $isort'$ hat dieselbe Best- und Worst-Case-Komplexität wie *isort*. Allerdings reichen unsere Methoden für die Laufzeitanalyse von $isort'$ nicht aus. Wir wollen uns daher mit einer Pi-mal-Daumen-Argumentation behelfen.

a) Geben Sie die Rekursionsfolge für *foldl* und $(insert,\ nil,\ [1,2,3,4])$ an. Annotieren Sie für jeden Rekursionsschritt die anfallenden Nebenkosten. Machen Sie sich klar, dass dieses Beispiel für Argumentlisten der Länge 4 maximale Nebenkosten hat.
b) Leiten Sie aus dem obigen Beispiel eine Worst-Case-Laufzeitfunktion für $isort'$ ab. Überprüfen Sie Ihre Laufzeitfunktion für die Größen 0 bis 4.
c) Leiten Sie nach dem gerade für den Worst-Case-Fall gezeigten Muster eine Best-Case-Laufzeitfunktion für $isort'$ ab.

11.8 Polynomieller Rekurrenzsatz

Die explizite Bestimmung der Laufzeitfunktion ist für die meisten Prozeduren schwierig oder sogar unmöglich. Glücklicherweise kann die Komplexität einer Prozedur aber auch direkt aus einer rekursiven Beschreibung der Laufzeitfunktion bestimmt werden. Dabei kommen sogenannte Rekurrenzsätze zum Einsatz. Wir beginnen mit dem Rekurrenzsatz für polynomielle Rekurrenzen:[1]

Satz 11.5 (Polynomielle Rekurrenzen) *Seien r, g Funktionen $\mathbb{N} \to \mathbb{N}_+$ und n_0, k, s natürliche Zahlen mit $s \geq 1$. Dann gilt:*

$$\left.\begin{array}{r} rn = r(n-s) + gn \text{ für alle } n \geq n_0 \\ O(g) = O(n^k) \end{array}\right\} \implies O(r) = O(n^{k+1})$$

Die bei der Formulierung des Satzes verwendete Gleichung $rn = r(n-s) + gn$ wird in diesem Zusammenhang als **Rekurrenz** bezeichnet, da sie als Teil einer rekursiven Beschreibung der Funktion r zu verstehen ist. Aus der Form der Rekurrenz ergibt sich, dass der polynomielle Rekurrenzsatz auf linear-rekursive Prozeduren anwendbar ist.

Wir erläutern die Anwendung des Satzes an zwei Beispielen. Zunächst betrachten wir die Konkatenationsprozedur @ für Listen (§ 11.2.1). Aus den definierenden Gleichungen von @ ergibt sich die rekursive Beschreibung der Laufzeitfunktion wie folgt:

$$\begin{array}{l} r \in \mathbb{N} \to \mathbb{N}_+ \\ r0 = g0 \\ rn = r(n-1) + gn \quad \text{für } n > 0 \end{array}$$

Da keine Nebenkosten anfallen, gilt $gn = 1$ und folglich $O(g) = O(1) = O(n^0)$. Also folgt mit dem obigen Rekurrenzsatz, dass $O(r) = O(n)$.

Als zweites Beispiel betrachten wir die naive Reversionsprozedur *rev* für Listen (§ 11.7). Da die Prozedur *rev* dieselbe Rekursionsstruktur wie @ hat, ist die rekursive Darstellung ihrer Laufzeitfunktion bis auf die Kostenfunktion g mit der von @ identisch. Die Kostenfunktion haben wir in § 11.7 als $gn = n+1$ bestimmt. Also $O(g) = O(n+1) = O(n^1)$. Also folgt mit dem obigen Rekurrenzsatz, dass *rev* quadratische Komplexität hat. Das stimmt mit dem in § 11.7 erzielten Ergebnis überein.

Aufgabe 11.28 Bestimmen Sie die Komplexitäten der Prozeduren *revi* (Aufgabe 11.2 auf S. 219) und *fac* (Aufgabe 11.3) mithilfe des polynomiellen Rekurrenzsatzes. Geben Sie jeweils eine rekursive Beschreibung der Laufzeitfunktion und eine explizite Beschreibung der Kostenfunktion an.

Aufgabe 11.29 Geben Sie die Komplexitäten der im Folgenden rekursiv definierten Funktionen $f \in \mathbb{N} \to \mathbb{R}$ möglichst einfach an.

[1] Wir werden diesen und die nachfolgenden Rekurrenzsätze nicht beweisen. In Rosens Buch [6] finden Sie Beweise ähnlicher Sätze unter der Überschrift *Advanced Counting Techniques*.

a) $f\,n = \text{if } n < 3 \text{ then } 1 \text{ else } f(n-2) + 5$
b) $f\,n = \text{if } n < 30 \text{ then } 1 \text{ else } f(n-6) + 3n + 7n^2$
c) $f\,n = \text{if } n < 27 \text{ then } 1 \text{ else } f(n-7) + 3n^7 + 7n^3$

Aufgabe 11.30 Geben Sie möglichst einfache Prozeduren $\mathbb{N} \to \{0\}$ an, die für die Größenfunktion $\lambda\, n.\, n$ die Komplexitäten $O(1)$, $O(n)$, $O(n^2)$ und $O(n^3)$ haben. Für $O(1)$ und $O(n)$ sollen keine Nebenkosten anfallen. Für $O(n^2)$ und $O(n^3)$ sollen lineare beziehungsweise quadratische Nebenkosten anfallen.

11.9 Exponentieller Rekurrenzsatz

Baumrekursive Prozeduren, die die Größe ihres Arguments bei jedem Rekursionsschritt nur geringfügig reduzieren, sind Kandidaten für exponentielle Laufzeit. Der folgende Satz formuliert ein Konstruktionsschema für Prozeduren exponentieller Komplexität:

Satz 11.6 (Exponentielle Rekurrenzen) *Seien r, g Funktionen $\mathbb{N} \to \mathbb{N}_+$ und b, n_0, k natürliche Zahlen mit $b \geq 2$. Dann gilt:*

$$\left.\begin{array}{r} r\,n = b \cdot r(n-1) + g\,n \text{ für alle } n \geq n_0 \\ O(g) \subseteq O(n^k) \end{array}\right\} \implies O(r) = O(b^n)$$

Als Beispiel für die Anwendung des Satzes betrachten wir die Prozedur *ntree* aus § 11.3. Da deren Laufzeitfunktion die Rekurrenz des obigen Satzes mit $b = 2$ und $g = \lambda\, n.\, 1$ erfüllt, folgt, dass *ntree* die Komplexität $O(2^n)$ hat. Das stimmt mit dem Ergebnis aus § 11.3 überein.

Auch die Fibonacci-Prozedur

$$\begin{array}{l} \mathit{fib}: \mathbb{N} \to \mathbb{N} \\ \mathit{fib}\; n = \text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2) \end{array}$$

hat exponentielle Komplexität, wenn wir die Größenfunktion $\lambda\, n.\, n$ zugrunde legen. Allerdings kann das nicht mit dem obigen Rekurrenzsatz gezeigt werden, da die rekursive Beschreibung der Laufzeitfunktion

$$\begin{array}{l} r\,n = 1 \quad \text{für } n \leq 1 \\ r\,n = r(n-1) + r(n-2) + 1 \quad \text{für } n \geq 2 \end{array}$$

nicht der Rekurrenz des Satzes entspricht. Mit etwas Rechnerei kann man zeigen,[2] dass die Laufzeitfunktion von *fib* die Komplexität $O(\phi^n)$ mit $\phi = \frac{1+\sqrt{5}}{2} \approx 1{,}62$ hat. Damit hat *fib* eine exponentielle Komplexität, die kleiner als $O(2^n)$ ist (Proposition 11.3, S. 225).

[2] Siehe zum Beispiel das Buch von Graham, Knuth und Patashnik [2].

Aufgabe 11.31 Geben Sie möglichst einfache Prozeduren $\mathbb{N} \to \{0\}$ mit den Komplexitäten $O(2^n)$ und $O(3^n)$ an.

Aufgabe 11.32 Seien zwei Prozeduren wie folgt gegeben:

$p : \mathbb{N} \to \mathbb{N}$
$pn =$ if $n < 5$ then n else $p(n-2)$

$q : \mathbb{N} \to \mathbb{N}$
$qn =$ if $n < 12$ then $3n$ else $q(n-1) + q(n-1) + pn$

Sie sollen die Laufzeitfunktionen und die Komplexitäten der Prozeduren für die Größenfunktion $\lambda\, n.\, n$ bestimmen.

a) Beschreiben Sie die Laufzeitfunktion von p rekursiv.
b) Beschreiben Sie die Laufzeitfunktion von q rekursiv. Machen Sie dabei von der Laufzeitfunktion für p Gebrauch, um die für die Anwendung der Funktion p anfallenden Nebenkosten einzubringen.
c) Geben Sie die Komplexität der Prozedur p an.
d) Geben Sie die Komplexität der Prozedur q an.

11.10 Logarithmischer Rekurrenzsatz

Wir betrachten jetzt Prozeduren mit logarithmischer Komplexität.

Satz 11.7 (Logarithmische Rekurrenzen) *Seien r, g Funktionen $\mathbb{N} \to \mathbb{N}_+$ und b, n_0 natürliche Zahlen. Sei r monoton wachsend*[3] *und $b \geq 2$. Dann gilt:*

$$\left.\begin{array}{r} r(b{\cdot}n) = rn + gn \;\text{ für alle } n \geq n_0 \\ O(g) = O(1) \end{array}\right\} \implies O(r) = O(\log n)$$

Der Satz besagt, dass eine Prozedur logarithmische Laufzeit hat, wenn sie die Größe ihres Arguments bei jedem Rekursionsschritt um einen konstanten Faktor $b \geq 2$ reduziert und höchstens konstante Nebenkosten hat.

11.10.1 Beispiel: Schnelles Potenzieren

Als Beispiel für logarithmische Komplexität betrachten wir eine Prozedur, die zu x und n die Potenz x^n liefert:

$exp : \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$
$exp(x, 0) = 1$
$exp(x, n) = exp(x^2,\, n/2)$ für $n \geq 2$ und n gerade
$exp(x, n) = x \cdot exp(x^2,\, (n-1)/2)$ für n ungerade

[3] Eine Funktion $f \in \mathbb{R} \to \mathbb{R}$ heißt *monoton wachsend*, wenn $\forall x, y \in \mathbb{R}\colon\ x \leq y \implies fx \leq fy$.

Da die Rekursion der Prozedur nur von n abhängt, wählen wir die Größenfunktion $\lambda(x,n).n$. Damit hat *exp* uniforme Laufzeit. Außerdem hat *exp* die für logarithmische Laufzeit typische Eigenschaft, dass die Argumentgröße bei jedem Rekursionsschritt mindestens halbiert wird. Die Laufzeitfunktion von *exp* können wir mit der Rekurrenz

$$r(2n) = rn + gn \quad \text{für } n > 0$$

und $O(g) = O(1)$ charakterisieren. Da r zudem monoton wachsend ist (Aufgabe 11.33), folgt mit dem logarithmischen Rekurrenzsatz, dass *exp* logarithmische Laufzeit hat.

Aufgabe 11.33 Stellen Sie die Laufzeitfunktion r von *exp* rekursiv dar und beweisen Sie, dass r monoton ist. Zeigen Sie dafür durch Induktion, dass gilt: $\forall n \in \mathbb{N}\ \forall k \in \mathbb{N}\colon\ k \leq n \Longrightarrow rk \leq rn$.

Aufgabe 11.34 Beweisen Sie, dass *exp* die Funktion $\lambda(x,n) \in \mathbb{Z} \times \mathbb{N}.\, x^n$ berechnet.

Aufgabe 11.35 Geben Sie eine endrekursive Prozedur $expi : \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$ an, die Potenzen x^n mit logarithmischer Laufzeit $O(\log n)$ bestimmt. Es soll $expi\ 1\ x\ n = x^n$ gelten. Geben Sie zuerst die Funktion an, die *expi* berechnen soll.

Aufgabe 11.36 Geben Sie eine möglichst einfache Prozedur $\mathbb{N} \to \{0\}$ an, die für die Größenfunktion $\lambda n.\, n$ die Komplexität $O(\log n)$ hat.

Aufgabe 11.37 Betrachten Sie die Prozedur

$$p : \mathbb{N} \to \mathbb{N}$$
$$pn = \text{if } n < 23 \text{ then } 7 \text{ else } p(\left\lfloor \frac{n}{3} \right\rfloor)$$

a) Geben Sie die Komplexität der Prozedur für die Größenfunktion $\lambda n.\, n$ an.

b) Geben Sie die Komplexität der Ergebnisfunktion der Prozedur an.

11.10.2 Beispiel: Euklidscher Algorithmus

Mit etwas Rechnerei werden wir jetzt zeigen, dass der Euklidsche Algorithmus höchstens logarithmische Komplexität hat. Dazu analysieren wir die Laufzeit der folgenden Prozedur.

$$euclid : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$euclid(x,0) = x$$
$$euclid(x,y) = euclid(y,\ x \bmod y) \quad \text{für } y > 0$$

Als Größenfunktion wählen wir $\lambda(x,y).\, y$. Außerdem nehmen wir an, dass für die Bestimmung von mod keine Nebenkosten anfallen.

Da *euclid* für (x,x) höchstens einmal rekurriert, ist die Best-Case-Laufzeit von *euclid* konstant.

Für die Worst-Case-Laufzeit von *euclid* ist mehr Aufwand erforderlich. Zunächst stellen wir fest, dass sich die Worst-Case-Laufzeitfunktion nicht rekursiv darstellen lässt (zumindest nicht auf offensichtliche Weise).

Wir werden zeigen, dass für die Worst-Case-Laufzeitfunktion r von *euclid* $O(r) \subseteq O(\log n)$ gilt. Für den Beweis dieser Tatsache benötigen wir den folgenden Satz:

Satz 11.8 *Sei p eine linear-rekursive Prozedur, für deren Laufzeit keine Nebenkosten anfallen. Weiter sei eine Größenfunktion und eine Zahl $b \geq 1$ gegeben, sodass p die Größe seines Arguments nach jeweils b Rekursionsschritten mindestens halbiert. Dann hat p höchstens logarithmische Komplexität.*

Beweis Sei $r \in \mathbb{N} \to \mathbb{N}_+$ die Laufzeitfunktion von p. Nach Voraussetzung gilt $r0 = 1$ und $\forall n \geq 1\ \exists m \leq \frac{n}{2}\colon\ rn \leq b + rm$. Wir definieren eine Funktion r':

$$
\begin{aligned}
&r' \in \mathbb{N} \to \mathbb{N}_+ \\
&r'0 = 1 \\
&r'n = b + r'\left\lfloor \frac{n}{2} \right\rfloor \quad \text{für } n > 0
\end{aligned}
$$

Da r' monoton wachsend ist (Aufgabe 11.33), folgt mit dem logarithmischen Rekurrenzsatz $O(r') = O(\log n)$. Um die Behauptung des Satzes zu beweisen, genügt es jetzt, $rn \leq r'n$ für alle $n \in \mathbb{N}$ zu zeigen. Wir tun dies durch Induktion über n. Für $n = 0$ gilt $rn \leq r'n$, da $r0 = 1$ nach Voraussetzung und $r'0 = 1$ nach Definition von r'. Für $n > 0$ ergibt sich $rn \leq r'n$ wie folgt:

$$
\begin{aligned}
rn &\leq rm + b && \text{nach Voraussetzung für ein } m \leq \left\lfloor \frac{n}{2} \right\rfloor \\
&\leq r'm && \text{Induktion für } m \\
&\leq b + r'\left\lfloor \frac{n}{2} \right\rfloor && r' \text{ monoton wachsend} \\
&= r'n && \text{Definition } r'
\end{aligned}
$$

■

Wir zeigen jetzt, dass die Prozedur *euclid* die Größe ihres Arguments nach jeweils zwei Rekursionsschritten mindestens halbiert. Seien $x \in \mathbb{N}$, $y \in \mathbb{N}_+$ und $x \bmod y \geq 1$. Zwei Rekursionsschritte führen von dem Argument (x, y) zu dem Argument $(x \bmod y,\ y \bmod (x \bmod y))$. Wir werden zeigen, dass

$$y > 2(y \bmod (x \bmod y)) \qquad (*)$$

gilt. Offensichtlich gilt $\frac{y}{x \bmod y} < \left\lfloor \frac{y}{x \bmod y} \right\rfloor + 1$. Also gilt

$$x \bmod y > y - \left\lfloor \frac{y}{x \bmod y} \right\rfloor (x \bmod y)$$

und folglich $x \bmod y > y \bmod (x \bmod y)$. Damit zeigen wir $(*)$ wie folgt:

$$\begin{aligned} y &= \left\lfloor \frac{y}{x \bmod y} \right\rfloor (x \bmod y) + y \bmod (x \bmod y) \\ &\geq 1(x \bmod y) + y \bmod (x \bmod y) && \text{da } y > (x \bmod y) \\ &> y \bmod (x \bmod y) + y \bmod (x \bmod y) && \text{da } x \bmod y > y \bmod (x \bmod y) \\ &= 2(y \bmod (x \bmod y)) \end{aligned}$$

11.11 Linear-logarithmischer Rekurrenzsatz

Zu guter Letzt wollen wir die Laufzeit von Sortieren durch Mischen (§ 5.4) analysieren. Dazu benötigen wir einen weiteren Rekurrenzsatz:

Satz 11.9 (Linear-logarithmische Rekurrenzen) *Seien r, g Funktionen $\mathbb{N} \to \mathbb{N}_+$ und seien b, n_0 natürliche Zahlen. Sei r monoton wachsend und $b \geq 2$. Dann gilt:*

$$\left.\begin{aligned} r(b \cdot n) = b \cdot r\, n + g\, n \;\; \text{für alle } n \geq n_0 \\ O(g) = O(n) \end{aligned}\right\} \implies O(r) = O(n \cdot \log n)$$

Sortieren durch Mischen realisieren wir mit der folgenden Prozedur:

$$\begin{aligned} &\mathit{msort} : \mathscr{L}(\mathbb{Z}) \to \mathscr{L}(\mathbb{Z}) \\ &\mathit{msort}\ [\,] = [\,] \\ &\mathit{msort}\ [x] = [x] \\ &\mathit{msort}\ xs = \text{let } (ys, zs) = \mathit{split}\ xs \text{ in } \mathit{merge}(\mathit{msort}\ ys,\ \mathit{msort}\ zs) \qquad \text{für } |xs| \geq 2 \end{aligned}$$

Als Größenfunktion wählen wir $\lambda\, xs.|xs|$. Da die Funktion *split* (§ 11.7.1) eine Liste gerader Länge in zwei gleich lange Listen zerlegt, erfüllt die Laufzeitfunktion von *msort* für $n \geq 1$ die Gleichung

$$r(2n) = 2r\, n + g\, n$$

Da für *split* lineare und für *merge* höchstens lineare Nebenkosten anfallen (§ 11.7.1 und Aufgabe 11.26, S. 231), gilt $O(g) = O(n)$. Da r monoton wachsend ist (wir verlassen uns hier auf unsere Intuition), folgt mit dem linear-logarithmischen Rekurrenzsatz, dass *msort* linear-logarithmische Laufzeit hat.

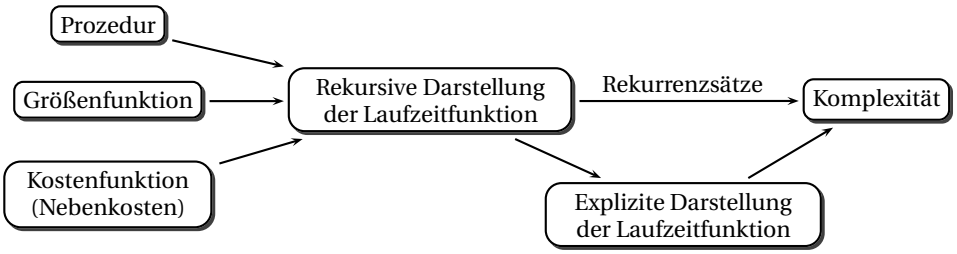
Aufgabe 11.38 Geben Sie die Komplexitäten der im Folgenden rekursiv definierten Funktionen $f \in \mathbb{N} \to \mathbb{N}$ möglichst einfach an.

a) $f\, n = \text{if } n < 7 \text{ then } 1 \text{ else } 3\, f(n-1) + n \cdot \log n + 5$

b) $f\, n = \text{if } n < 2 \text{ then } 1 \text{ else } 2\, f(\lfloor \frac{n}{2} \rfloor) + 3n + \log n + 5$

Aufgabe 11.39 Geben Sie eine möglichst einfache Prozedur $\mathbb{N} \to \{0\}$ an, die für die Größenfunktion $\lambda\, n.\, n$ die Komplexität $O(n \cdot \log n)$ hat. Benutzen Sie eine Funktion, für die lineare Nebenkosten anfallen.

Komplexitätsbestimmung im Überblick



Rekurrenzsätze kurz gefasst

Rekurrenz	$O(r)$	
$r\,n = r(n-s) + O(n^k)$	$O(n^{k+1})$	polynomiell
$r\,n = b \cdot r(n-1) + O(n^k)$	$O(b^n)$	exponentiell
$r(b \cdot n) = r\,n + O(1)$	$O(\log n)$	logarithmisch
$r(b \cdot n) = b \cdot r\,n + O(n)$	$O(n \cdot \log n)$	linear-logarithmisch

Bemerkungen

In diesem Kapitel haben wir ein mathematisches Modell kennengelernt, mit dem wir fundierte Aussagen über die Laufzeit von Prozeduren machen können. Dabei wird die Laufzeit einer Prozedur durch die Komplexität ihrer Laufzeitfunktion bewertet. Der für Laufzeitfunktionen verwendete Komplexitätsbegriff gehört seit mehr als 100 Jahren zum mathematischen Fundus. Die Idee, die Laufzeit von Algorithmen gemäß dieses Komplexitätsbegriffs zu bewerten, wurde von den Informatikern Juris Hartmanis, Donald Knuth und Richard Stearns entwickelt.

Die Laufzeit einer Prozedur für ein Argument x veranschlagen wir als die Größe des Rekursionsbaums für x zuzüglich etwaiger Nebenkosten für die Berechnung von Funktionen. Die Laufzeitfunktion einer Prozedur liefert zu jeder Argumentgröße die maximale Laufzeit, die die Prozedur für Argumente dieser Größe benötigt (Worst-Case-Annahme). Die Argumentgröße ergibt sich dabei gemäß einer passend gewählten Größenfunktion.

Die Komplexität von Prozeduren kann am einfachsten mithilfe von Rekurrenzsätzen ermittelt werden. Dafür benötigt man eine rekursive Darstellung der Laufzeitfunktion der Prozedur, die mit einer Kostenfunktion formuliert ist. Von der Kostenfunktion muss nur die Komplexität bekannt sein. Wir haben Rekurrenzsätze für polynomielle,

Komplexitäten kurz gefasst

- Prozeduren *konstanter Komplexität* sind normalerweise nicht rekursiv und rechnen extrem schnell.
- Prozeduren *logarithmischer Komplexität* sind normalerweise linear-rekursiv und rechnen extrem schnell. Typischerweise reduzieren sie die Argumentgröße bei jedem Rekursionsschritt mindestens um die Hälfte. Ein typisches Beispiel ist die Prozedur *exp* zur Berechnung von Potenzen.
- Prozeduren *linearer Komplexität* sind oft linear-rekursiv und rechnen sehr schnell. Bei einer Verdoppelung der Argumentgröße verdoppelt sich ihre Laufzeit. Typische Beispiele sind die Prozeduren zum Konkatenieren und Falten von Listen.
- Prozeduren *linear-logarithmischer Komplexität* rechnen fast so schnell wie Prozeduren linearer Komplexität. Sie sind oft binärrekursiv, wobei bei jedem Rekursionsschritt lineare Nebenkosten anfallen und die Argumentgröße halbiert wird. Ein typisches Beispiel ist die Prozedur *msort*, die Listen durch Mischen sortiert.
- Prozeduren *quadratischer Komplexität* rechnen für größere Argumente oft zu langsam. Bei einer Verdoppelung der Argumentgröße vervierfacht sich ihre Laufzeit. Sie sind typischerweise linear-rekursiv, wobei bei jedem Rekursionsschritt lineare Nebenkosten anfallen und die Argumentgröße um einen fixen Betrag reduziert wird. Typische Beispiele sind die naive Reversionsprozedur für Listen und Sortieren durch Einfügen.
- Prozeduren *exponentieller Komplexität* rechnen normalerweise bereits für mittelgroße Argumente zu langsam. Bei einer Verdoppelung der Argumentgröße quadriert sich bei $O(2^n)$ die Laufzeit. Sie sind typischerweise baumrekursiv und reduzieren die Argumentgröße bei jedem Rekursionsschritt nur um einen kleinen Betrag. Typische Beispiele für exponentielle Komplexität sind die Prozeduren für die naive Berechnung von Binärbäumen und Fibonacci-Zahlen.

Prozeduren mit quadratischer oder größerer Komplexität sind Kandidaten für Laufzeitprobleme. Oft ist es möglich, eine langsame Prozedur durch eine schnellere Prozedur zu ersetzen. Wir haben dafür einige Beispiele gesehen: Reversieren von Listen (lineare statt quadratische Laufzeit), Sortieren von Listen (linear-logarithmische statt quadratische Laufzeit), Fibonacci-Zahlen (lineare statt exponentielle Laufzeit) und Potenzen (logarithmische statt lineare Laufzeit).

exponentielle, logarithmische und linear-logarithmische Komplexitäten kennengelernt.

Wir merken noch an, dass viele in der Praxis relevanten Probleme nur mit exponentieller Laufzeit gelöst werden können. Dazu gehören die Erstellung optimaler Stundenpläne und die Planung optimaler Routen für LKWs.

Verzeichnis

Laufzeit einer Prozedur für ein Argument.

Laufzeitfunktion einer Prozedur; Größenfunktion; Worst-Case-Annahme; Worst-Case- und Best-Case-Laufzeit; uniforme Laufzeit.

Rekursive und explizite Darstellung von Laufzeitfunktionen; Nebenkosten und Kostenfunktionen.

Komplexität von Funktionen und Prozeduren; Dominanzrelation $f \prec g$; Komplexitäten $O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(b^n)$.

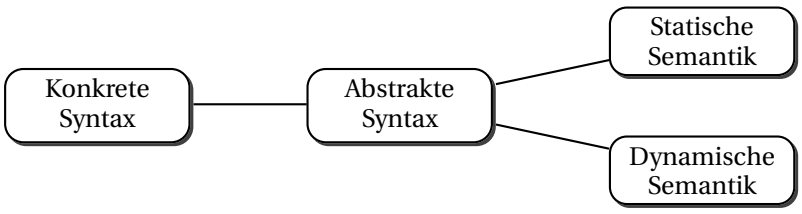
Rekurrenzsätze.

12 Statische und dynamische Semantik

Wir wollen jetzt genauer auf die Struktur und die Definition von Programmiersprachen eingehen. Damit setzen wir ein Thema fort, das wir in Kapitel 2 begonnen haben. Diesmal geht es uns vor allem um die präzise Definition von Programmiersprachen. Dazu betrachten wir eine kleine Teilsprache von Standard ML, die wir F nennen. Für die Syntax und Semantik von F entwickeln wir mathematische Definitionen, die wir mit zwei Beschreibungswerkzeugen formulieren, die als Grammatiken und Inferenzregeln bezeichnet werden. Die syntaktische und die semantische Beschreibung von F implementieren wir in Standard ML und erhalten damit einen Interpreter für F.

12.1 Abstrakte Syntax

Man unterscheidet zwischen der abstrakten und der konkreten Syntax einer Sprache. Die **abstrakte Syntax** beschreibt die Phrasen der Sprache in Baumform. Darauf aufbauend regelt die **konkrete Syntax**, wie die Phrasen der Sprache durch Wörter und Zeichen dargestellt werden. Die abstrakte Syntax bildet auch die Grundlage für die Formulierung der statischen und der dynamischen Semantik. Damit spielt sie die zentrale Rolle bei der Beschreibung einer Sprache:



Die abstrakte Syntax einer Sprache können wir in Standard ML durch Typdeklarationen beschreiben. Dieses Vorgehen haben wir bereits bei der Darstellung arithmetischer Ausdrücke in § 6.4 kennengelernt. Abbildung 12.1 zeigt die Typdeklarationen für die abstrakte Syntax von F. Hier sind Beispiele für die konkrete und die abstrakte Darstellung

```
datatype con = False | True | IC of int     (* constants *)
type     id  = string                       (* identifiers *)
datatype opr = Add | Sub | Mul | Leq        (* operators *)

datatype ty  =                              (* types *)
    Bool
  | Int
  | Arrow of ty * ty                        (* procedure type *)

datatype exp =                              (* expressions *)
    Con of con                              (* constant *)
  | Id  of id                               (* identifier *)
  | Opr of opr * exp * exp                  (* operator application *)
  | If  of exp * exp * exp                  (* conditional *)
  | Abs of id  * ty  * exp                  (* abstraction *)
  | App of exp * exp                        (* procedure application *)
```

Abbildung 12.1: *Abstrakte Syntax von F (Typdeklarationen)*

von Phrasen:

$int \to bool$	$Arrow(Int, Bool)$
$x \leq 3$	$Opr(Leq, Id"x", Con(IC\ 3))$
$if\ b\ then\ x\ else\ y$	$If(Id"b", Id"x", Id"y")$
$fn\ x : int \Rightarrow f\ x$	$Abs("x", Int, App(Id"f", Id"x"))$

Aufgabe 12.1 Geben Sie Deklarationen an (in Standard ML), die den Bezeichner *e* an die abstrakte Darstellung des Ausdrucks

$$fn\ f : int \to int \Rightarrow\ fn\ n : int \Rightarrow\ if\ n \leq 0\ then\ 1\ else\ n * f(n-1)$$

binden. Gehen Sie dabei schrittweise vor und beginnen Sie mit der Deklaration des Teilausdrucks $n \leq 0$:

```
val e1 = Opr(Leq, Id"n", Con(IC 0))
```

Aufgabe 12.2 In § 2.4 und § 3.8 haben wir definiert, was wir unter offenen und geschlossenen Ausdrücken und den freien Variablen eines Ausdrucks verstehen wollen.

a) Schreiben Sie eine Prozedur $closed : exp \to bool$, die testet, ob ein Ausdruck geschlossen ist. Verwenden Sie dabei eine Hilfsprozedur $closed' : exp \to id\ list \to bool$, die testet, ob alle freien Bezeichner eines Ausdrucks in einer Liste vorkommen.

b) Schreiben Sie eine Prozedur $free : exp \to id\ list$, die zu einem Ausdruck eine Liste liefert, die die in diesem Ausdruck frei auftretenden Bezeichner enthält. Die Liste darf denselben Bezeichner mehrfach enthalten. Verwenden Sie eine Hilfsprozedur $free' : id\ list \to exp \to id\ list$, die nur die frei auftretenden Bezeichner liefert, die nicht in einer Liste von "gebundenen" Bezeichnern enthalten sind.

z	$\in$	$\mathbb{Z}$	Zahlen
c	$\in$	$Con = false \mid true \mid z$	Konstanten
x	$\in$	$Id = \mathbb{N}$	Bezeichner
o	$\in$	$Opr = + \mid - \mid * \mid \leq$	Operatoren
t	$\in$	$Ty = bool \mid int \mid t \to t$	Typen
e	$\in$	$Exp =$	Ausdrücke
		c	Konstante
		$\mid x$	Bezeichner
		$\mid eoe$	Operatoranwendung
		$\mid$ *if* e *then* e *else* e	Konditional
		$\mid fn\ x : t \Rightarrow e$	Abstraktion
		$\mid ee$	Prozeduranwendung

Abbildung 12.2: *Abstrakte Syntax von F (Abstrakte Grammatik)*

12.2 Abstrakte Grammatiken

Üblicherweise definiert man die abstrakte Syntax einer Sprache mithilfe einer schematischen Darstellung, die als **abstrakte Grammatik** bezeichnet wird. Abbildung 12.2 zeigt eine abstrakte Grammatik für die abstrakte Syntax von F. Die Grammatik entspricht im Wesentlichen den Typdeklarationen in Abbildung 12.1:

- Die Grammatik definiert die Mengen *Con*, *Id*, *Opr*, *Ty* und *Exp*, die den Typen *con*, *id*, *opr*, *ty* und *exp* entsprechen.
- Die Grammatik realisiert Bezeichner durch natürliche Zahlen, die Typdeklarationen realisieren Bezeichner durch Strings. Es ist die Aufgabe der konkreten Syntax, die konkrete Darstellung der Bezeichner zu regeln. Bei der abstrakten Syntax genügt es, einen unendlichen Vorrat an Bezeichnern bereitzustellen.

Die Grammatik legt eine textuelle Notation für Phrasen fest, die sich an die konkrete Syntax von Standard ML anlehnt. Auf die Angabe von Klammersparregeln wird verzichtet. Die durch die Grammatik eingeführte Notation findet bei der Definition der statischen und der dynamischen Semantik Verwendung. Die **Metavariablen** z, c, x, o, t, e bezeichnen im Kontext der Grammatik stets Objekte der jeweiligen Wertemenge. Als Beispiel für die Verwendung der durch die Grammatik definierten Notation geben wir in Abbildung 12.3 die Definition einer Funktion an, die zu einem Ausdruck die Menge der in ihm frei auftretenden Bezeichner liefert.

Die durch die abstrakte Grammatik festgelegte Notation ist äußerst kompakt. Das wird mit diversen notationalen Tricks erkauft, die auf die Intelligenz des menschlichen Lesers vertrauen. Insbesondere wird auf die explizite Angabe der Konstruktoren *Con* und *Id* verzichtet, wie man in Abbildung 12.3 sehen kann.

$$
\begin{aligned}
FI \in Exp \to \mathscr{P}(Id)& \\
FI\ c &= \emptyset \\
FI\ x &= \{x\} \\
FI(e_1\, o\, e_2) &= FI\ e_1 \cup FI\ e_2 \\
FI(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) &= FI\ e_1 \cup FI\ e_2 \cup FI\ e_3 \\
FI(\mathit{fn}\, x : t \Rightarrow e) &= FI\ e - \{x\} \\
FI(e_1\, e_2) &= FI\ e_1 \cup FI\ e_2
\end{aligned}
$$

Abbildung 12.3: *Freie Bezeichner*

Aufgabe 12.3 Übersetzen Sie die ersten zwei Gleichungen aus Abbildung 12.3 gemäß den Typdeklarationen in Abbildung 12.1 nach Standard ML. Realisieren Sie Mengen als Listen und achten Sie auf die Verwendung der Konstruktoren *Con* und *Id*.

Aufgabe 12.4 Die durch die abstrakte Grammatik für F eingeführten Phrasen können als mathematische Objekte dargestellt werden. Dabei werden Konstanten, Operatoren, Typen und Ausdrücke ähnlich wie die Werte von Konstruktortypen durch Tupel mit Variantennummern dargestellt (§ 6.1):

$int \to bool$	$\langle 3, \langle 2\rangle, \langle 1\rangle\rangle$
$7 \le 13$	$\langle 3, \langle 4\rangle, \langle 1, \langle 3,7\rangle\rangle, \langle 1, \langle 3,13\rangle\rangle\rangle$
if true then 72 else 33	$\langle 4, \langle 1, \langle 2\rangle\rangle, \langle 1, \langle 3,72\rangle\rangle, \langle 1, \langle 3,33\rangle\rangle\rangle$

Dabei werden die Variantennummern für die mit Alternativen definierten Mengen *Con*, *Opr*, *Ty* und *Exp* jeweils lokal vergeben. Entsprechend ist die Darstellung nur jeweils innerhalb einer syntaktischen Klasse eindeutig. Beispielsweise werden die Konstante *false*, der Operator + und der Typ *bool* alle durch das Tupel $\langle 1\rangle$ dargestellt. Man muss also wissen, ob ein mathematisches Objekt als Konstante, Typ, Operator oder Ausdruck interpretiert werden soll.
Geben Sie die mathematische Darstellung der folgenden Ausdrücke an. Nehmen Sie dabei an, dass der Bezeichner x durch die Zahl 37 dargestellt wird.

a) $x \le 3$

b) $2 * x$

c) *if* $x \le 3$ *then* $2 * x$ *else* 5

12.3 Statische Semantik

Die statische Semantik formuliert Konsistenzbedingungen für die Phrasen der abstrakten Syntax. Dabei geht es in erster Linie um den typgerechten Aufbau von Phrasen und um die Bindung von Bezeichnern (§ 2.6, § 3.8). Eine Phrase heißt semantisch zulässig, wenn sie die Bedingungen der statischen Semantik erfüllt (§ 2.6). Ein Interpreter prüft

die semantische Zulässigkeit einer Phrase im Rahmen einer Verarbeitungsphase, die als semantische Analyse oder Elaboration bezeichnet wird (§ 2.8).

Die semantische Zulässigkeit von Phrasen wird in Bezug auf **Typumgebungen** definiert, die die Typen für frei auftretende Bezeichner vorgeben (sogenannte statische Bindungen, siehe § 3.8.3). Wir wollen unter einer Typumgebung eine Funktion $Id \rightharpoonup Ty$ verstehen, die endlich vielen Bezeichnern je einen Typ zuordnet:

$$T \quad \in \quad TE \quad = \quad Id \xrightarrow{\text{fin}} Ty$$

Beispielsweise handelt es sich bei $\{(x, int),\ (y, bool)\}$ um eine Umgebung, die dem Bezeichner x den Typ *int* und dem Bezeichner y den Typ *bool* zuordnet. Mit der in § 2.4 eingeführten Notation können wir diese Typumgebung auch mit $[x := int,\ y := bool]$ beschreiben.

Wir werden die statische Semantik von F als eine Menge

$$SS \subseteq TE \times Exp \times Ty$$

definieren, die ein Tupel $\langle T, e, t\rangle$ genau dann enthält, wenn der Ausdruck e für die Typumgebung T zulässig ist und den Typ t hat. Beispielsweise soll SS das Tupel $\langle [x := int],\ 2 * x + 3,\ int\rangle$ enthalten.

Wir werden SS mithilfe sogenannter **Inferenzregeln** definieren. Als Beispiel betrachten wir die Inferenzregel für Prozeduranwendungen:

$$\frac{\langle T,\, e_1,\, t' \to t\rangle \in SS \qquad \langle T,\, e_2,\, t'\rangle \in SS}{\langle T,\, e_1\, e_2,\, t\rangle \in SS}$$

Die Regel besagt, dass eine Prozeduranwendung $e_1\, e_2$ für eine Typumgebung T zulässig ist und einen Typ t hat, wenn e_1 für T zulässig ist und einen funktionalen Typ $t' \to t$ hat und e_2 für T zulässig ist und den Typ t' hat.

Abstrakt gesehen besagt die obige Regel, dass die Menge SS das Tupel $\langle T,\, e_1\, e_2,\, t\rangle$ enthält, wenn sie die Tupel $\langle T,\, e_1,\, t' \to t\rangle$ und $\langle T,\, e_2,\, t'\rangle$ enthält. Allgemein hat eine Inferenzregel die Form

$$\frac{P_1 \qquad \cdots \qquad P_n}{P}$$

Die Aussagen $P_1, \ldots, P_n$ über dem Strich ($n \geq 0$) werden als **Prämissen** bezeichnet und die Aussage P unter dem Strich als **Konklusion**. Wir sagen, dass die Aussage P mit der Regel aus den Aussagen $P_1, \ldots, P_n$ **abgeleitet** werden kann.

Um die Lesbarkeit der Regeln zu verbessern, verwenden wir die folgende Notation:

$$T \vdash e : t \qquad :\Longleftrightarrow \qquad \langle T, e, t\rangle \in SS \qquad \text{(lies: } e \text{ hat für } T \text{ den Typ } t\text{)}$$

Abbildung 12.4 zeigt die definierenden Inferenzregeln für die Menge SS, wobei SS genau die Tupel enthalten soll, die mit den Regeln abgeleitet werden können. Für jede Regel

$$\textbf{Sfalse}\ \frac{}{T \vdash \mathit{false} : \mathit{bool}} \qquad \textbf{Strue}\ \frac{}{T \vdash \mathit{true} : \mathit{bool}}$$

$$\textbf{Snum}\ \frac{z \in \mathbb{Z}}{T \vdash z : \mathit{int}} \qquad \textbf{Sid}\ \frac{T\,x = t}{T \vdash x : t}$$

$$\textbf{Soai}\ \frac{o \in \{+,-,*\} \quad T \vdash e_1 : \mathit{int} \quad T \vdash e_2 : \mathit{int}}{T \vdash e_1\, o\, e_2 : \mathit{int}}$$

$$\textbf{Soab}\ \frac{T \vdash e_1 : \mathit{int} \quad T \vdash e_2 : \mathit{int}}{T \vdash e_1 \leq e_2 : \mathit{bool}}$$

$$\textbf{Sif}\ \frac{T \vdash e_1 : \mathit{bool} \quad T \vdash e_2 : t \quad T \vdash e_3 : t}{T \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : t}$$

$$\textbf{Sabs}\ \frac{T[x := t] \vdash e : t'}{T \vdash \mathit{fn}\ x : t \Rightarrow e : t \to t'}$$

$$\textbf{Sapp}\ \frac{T \vdash e_1 : t' \to t \quad T \vdash e_2 : t'}{T \vdash e_1\, e_2 : t}$$

Abbildung 12.4: *Statische Semantik von F*

ist links ein Name angegeben. Überzeugen Sie sich davon, dass für jeden Ausdruck e genau eine Regel existiert, mit der Tupel der Form $\langle T, e, t\rangle$ abgeleitet werden können. Die Notation $T[x := t]$ wurde in § 8.6.4 definiert. Die in den Regeln vorkommenden Metavariablen z, c, x, o, t, e dürfen nur mit Objekten aus den durch die abstrakte Grammatik vereinbarten Wertebereichen instanziiert werden.

Eine Aussage $T \vdash e : t$ ist gemäß Definition genau dann gültig, wenn sie mit den Inferenzregeln aus Abbildung 12.4 abgeleitet werden kann. Abbildung 12.5 zeigt eine **Ableitung** der Aussage $[x := \mathit{int}] \vdash \mathit{fn}\ b : \mathit{bool} \Rightarrow \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ 2 * x : \mathit{bool} \to \mathit{int}$.

Die folgende Proposition stellt fest, dass ein Ausdruck für eine Typumgebung höchstens einen Typ hat.

Proposition 12.1 (Determinismus) *Sei $T \vdash e : t$ und $T \vdash e : t'$. Dann $t = t'$.*

Beweis Durch strukturelle Induktion über $e \in \mathit{Exp}$. Die Behauptung folgt aus der Tatsache, dass Tupel mit dem Ausdruck e jeweils nur mit einer der Regeln abgeleitet werden können und dass die Konklusionen der Regeln die Umgebungen und Ausdrücke der Prämissen eindeutig bestimmen. ∎

Aufgabe 12.5 Geben Sie Typumgebungen an, für die der Ausdruck *if true then x else y* zulässig beziehungsweise unzulässig ist.

(1)	$[x := int,\ b := bool] \vdash b : bool$	mit Sid
(2)	$[x := int,\ b := bool] \vdash x : int$	mit Sid
(3)	$[x := int,\ b := bool] \vdash 2 : int$	mit Snum
(4)	$[x := int,\ b := bool] \vdash 2 * x : int$	mit Soai aus (3), (2)
(5)	$[x := int,\ b := bool] \vdash \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ 2 * x\ : int$	mit Sif aus (1), (2), (4)
(6)	$[x := int] \vdash \mathit{fn}\ b : bool \Rightarrow \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ 2 * x\ : bool \to int$	mit Sabs aus (5)

Abbildung 12.5: *Eine Ableitung*

Aufgabe 12.6 Geben Sie eine Ableitung für die folgende Aussage an: $[x := int] \vdash \mathit{fn}\ f : int \to bool \Rightarrow \mathit{fn}\ y : int \Rightarrow f(2 * x + y)\ : (int \to bool) \to (int \to bool)$.

12.4 Elaborierung

Die die statische Semantik definierenden Inferenzregeln beschreiben einen Algorithmus, der für eine Typumgebung T und einen Ausdruck e entscheidet, ob e für T zulässig ist. Im positiven Fall liefert der Algorithmus zudem den Typ von e für T. Der Algorithmus bestimmt zunächst die für e zuständige Inferenzregel. Wenn e atomar ist, kann die Zulässigkeit von e unmittelbar mit der Regel entschieden werden, und auch der Typ von e ergibt sich direkt aus der Regel. Wenn e aus Teilausdrücken zusammengesetzt ist, liefert die Regel Teilprobleme, die per Rekursion gelöst werden. Beispielsweise liefert die Regel

$$\textbf{Sabs}\quad \frac{T[x := t] \vdash e : t'}{T \vdash \mathit{fn}\ x : t \Rightarrow e : t \to t'}$$

zu T und $\mathit{fn}\ x : t \Rightarrow e$ das Teilproblem $T[x := t]$ und e. Aus der Lösung der Teilprobleme ergeben sich dann die Zulässigkeit und gegebenenfalls der Typ von e. Die Rekursion terminiert, da die Teilprobleme nur echte Teilausdrücke des Gesamtausdrucks enthalten (strukturelle Rekursion).

Wir implementieren den gerade beschriebenen Algorithmus für die Elaboration von Ausdrücken durch die in Abbildung 12.6 deklarierte Prozedur

elab: *ty env* → *exp* → *ty*

die als **Elaborierer** bezeichnet wird. Typumgebungen (engl. type environments) realisieren wir durch Prozeduren, die Ausnahmen werfen, falls für einen Bezeichner keine Bindung vorliegt (§ 6.4.2). Die Prozedur *empty* realisiert die Umgebung, die keinen Bezeichner bindet, und die Prozedur *update* realisiert die Operation $T[x := t]$.

```
val f = update (update empty "x" Int) "y" Bool
val f : ty env
```

```
type 'a env = id -> 'a  (* environments *)
exception Unbound of id
fun empty x = raise Unbound x
fun update env x a y = if y=x then a else env y

exception Error of string

fun elabCon True   = Bool
  | elabCon False  = Bool
  | elabCon (IC _) = Int
fun elabOpr Add Int Int = Int
  | elabOpr Sub Int Int = Int
  | elabOpr Mul Int Int = Int
  | elabOpr Leq Int Int = Bool
  | elabOpr _   _   _   = raise Error "T Opr"
fun elab f (Con c) = elabCon c
  | elab f (Id x) = f x
  | elab f (Opr(opr,e1,e2)) = elabOpr opr (elab f e1) (elab f e2)
  | elab f (If(e1,e2,e3)) =
             (case (elab f e1, elab f e2, elab f e3) of
                (Bool, t2, t3) => if t2=t3 then t2
                                  else raise Error "T If1"
              |  _ => raise Error "T If2")
  | elab f (Abs(x,t,e)) = Arrow(t, elab (update f x t) e)
  | elab f (App(e1,e2)) = (case elab f e1 of
                Arrow(t,t') => if t = elab f e2 then t'
                               else raise Error "T App1"
              | _  => raise Error "T App2")
```

val elab : ty env → exp → ty

Abbildung 12.6: *Implementierung der statischen Semantik von F*

```
f "y"
```
Bool : ty

```
f "z"
```
! Uncaught exception: Unbound "z"

Die Prozedur *elab* hat für jede Ausdrucksvariante eine Regel. Da es für Konstanten und Operatoren mehrere Inferenzregeln gibt, gibt es für ihre Elaboration die Hilfsprozeduren *elabCon* und *elabOpr*.

Es lohnt sich, die Inferenzregeln für die statische Semantik von F und ihre Implementierung durch die Prozedur *elab* genau zu verstehen. Falls Ihnen das noch schwer fällt, soll-

ten Sie zunächst nur die Teilsprache von F betrachten, deren Ausdrücke mit Konstanten, Bezeichnern und Operatoranwendungen gebildet werden. Wenn Sie die Inferenzregeln verstehen, wird es Ihnen leicht fallen, die aus den Regeln abgeleitete Prozedur *elab* zu verstehen. Umgekehrt können Sie aus der Funktionsweise der Prozedur *elab* die Funktionsweise der Inferenzregeln erschließen. Und die Funktionsweise von *elab* können Sie mit dem Interpreter erproben:

```
val f = update (update empty "x" Int) "y" Bool
```
val f : ty env

```
val e = Abs("y", Int, Opr(Leq, Id"x", Id"y"))
```
val e = Abs("y", Int, Opr(Leq, Id "x", Id "y")) : exp

```
elab f e
```
Arrow(Int, Bool) : ty

Ein Umschlag für elab

Die meisten Standard ML-Interpreter können geworfene Ausnahmen mit einstelligen Ausnahmekonstruktoren nur unvollständig darstellen. Bei der Arbeit mit der Prozedur *elab* kann dieses Problem mit einem **Umschlag** *elab'* gelöst werden:

```
datatype elab = T of ty | SE of string

fun elab' f e = T(elab f e) handle
    Unbound s => SE("Unbound "^s)
  | Error s => SE("Error "^s)
```
val elab' : ty env → exp → elab

```
elab' f e
```
T(Arrow(Int, Bool)) : elab

```
elab' f (App(Id"x", Id"x"))
```
SE "Error T App2" : elab

```
elab' empty (Id "x")
```
SE "Unbound x" : elab

Aufgabe 12.7 Deklarieren Sie mithilfe der Prozedur *elab* eine Prozedur *test* : *exp* → *bool*, die testet, ob ein Ausdruck geschlossen und zulässig ist.

Aufgabe 12.8 Geben Sie einen möglichst einfachen Ausdruck *e* an, sodass bei der Anwendung von *elab* auf *empty* und *e* jede der 6 Prozedurregeln zum Einsatz kommt. Erproben Sie *elab* mit einem Interpreter für diesen Ausdruck.

Aufgabe 12.9 Die Prozedur *elab* kann durch Werfen einer Ausnahme *Error s* einen Fehler *s* melden. Geben Sie möglichst einfache Ausdrücke an, für die die Prozedur *elab empty* die Fehler "*T Opr*", "*T If1*", "*T If2*", "*T App1*" und "*T App2*" meldet.

$$
\begin{array}{lcll}
v & \in & Val & = & \mathbb{Z} \cup Pro & \text{Werte} \\
 & & Pro & = & Id \times Exp \times VE & \text{Prozeduren} \\
V & \in & VE & = & Id \xrightarrow{\text{fin}} Val & \text{Wertumgebungen}
\end{array}
$$

Abbildung 12.7: *Werte, Prozeduren und Wertumgebungen für F*

12.5 Dynamische Semantik und Evaluierung

Die dynamische Semantik legt fest, ob und mit welchem Wert die Auswertung eines Ausdrucks terminiert. Sie wird so wie die statische Semantik durch Inferenzregeln definiert.

F hat zwei Arten von Werten, ganze Zahlen und Prozeduren. Die Werte für die Booleschen Konstanten *false* und *true* stellen wir durch die Zahlen 0 und 1 dar.

Eine **Wertumgebung** ist eine Funktion, die endlich vielen Bezeichnern einen Wert zuordnet. Wertumgebungen spielen in der dynamischen Semantik eine ähnliche Rolle wie Typumgebungen in der statischen Semantik.

Prozeduren stellen wir durch Tripel $\langle x, e, V\rangle$ dar, die aus einem Bezeichner x (der Argumentvariablen), einem Ausdruck e (dem Rumpf) und einer Wertumgebung V (den Bindungen für die freien Bezeichner des Codes, siehe § 2.5) bestehen. Wir werden die dynamische Semantik von F so definieren, dass die Auswertung einer Abstraktion $fn\ x : t \Rightarrow e$ in einer Wertumgebung V die Prozedur $\langle x, e, V\rangle$ liefert. Die hier gewählte Prozedurdarstellung unterscheidet sich von der in § 2.5 und § 3.1 beschriebenen in zwei Punkten: Zum einen wird auf den für die dynamische Semantik entbehrlichen Prozedurtyp verzichtet, und zum anderen wird die Wertumgebung aus Gründen der Einfachheit nicht auf die freien Bezeichner des Codes beschränkt (sie kann also zusätzliche Bindungen enthalten).

Abbildung 12.7 zeigt die verschränkt rekursiven Definitionen für Werte, Prozeduren und Wertumgebungen in zusammengefasster Form. Beachten Sie, dass die Wertumgebung einer Prozedur gemäß dieser Definition nicht für alle freien Bezeichner des Codes definiert sein muss. Es ist die Aufgabe der statischen Semantik, zulässige Ausdrücke so zu beschränken, dass die Auswertung von Abstraktionen bindungs- und typkonsistente Prozeduren liefert.

Wir werden die dynamische Semantik von F als eine Menge

$$DS \subseteq VE \times Exp \times Val$$

definieren, die ein Tupel $\langle V, e, v\rangle$ genau dann enthält, wenn die Auswertung des Ausdrucks e in der Wertumgebung V mit dem Wert v terminiert. Beispielsweise soll die Menge DS das Tupel $\langle [x := 5],\ x + 7,\ 12\rangle$ enthalten. Wie bereits erwähnt, werden wir die Menge DS so wie die Menge SS durch Inferenzregeln definieren. Dabei benutzen wir die folgende Notation:

$$V \vdash e \triangleright v \quad :\Longleftrightarrow \quad \langle V, e, v\rangle \in DS \qquad \text{(lies: } e \text{ hat für } V \text{ den Wert } v\text{)}$$

Abbildung 12.8 zeigt die definierenden Inferenzregeln für *DS*. Für Konditionale gibt es diesmal zwei Regeln, da die zweite Prämisse von der Auswertung der Bedingung abhängt. Da die Regeln sinngemäß bereits in § 2.7 erklärt wurden, sind keine weiteren Erklärungen erforderlich. Überzeugen Sie sich davon, dass die formale Beschreibung der dynamischen Semantik durch Inferenzregeln übersichtlicher und lesbarer ist als die informelle sprachliche Beschreibung in § 2.7.

Die folgende Proposition besagt, dass das Ergebnis der Auswertung eines Ausdrucks eindeutig bestimmt ist:

Proposition 12.2 (Determinismus) *Sei* $V \vdash e \triangleright v$ *und* $V \vdash e \triangleright v'$. *Dann* $v = v'$.

Beweis Aufgrund der Regel Dapp kann der Determinismus der dynamischen Semantik nicht wie bei der statischen Semantik durch strukturelle Induktion über e bewiesen werden. Stattdessen führt aber Induktion über die Länge der Ableitung für $V \vdash e \triangleright v$ zum Ziel. Falls e kein Konditional ist, gibt es nur eine Regel, mit der Tupel mit e abgeleitet werden können. Da die Konklusionen der Regeln die Umgebungen und Ausdrücke der Prämissen eindeutig bestimmen, folgt die Behauptung mit der Induktionsannahme. Falls e ein Konditional ist, gibt es zwar zunächst zwei anwendbare Regeln, aber die Auswertung der Bedingung, die gemäß Induktion ein eindeutiges Ergebnis liefert, schließt eine davon aus. ∎

Die statische Semantik von F sorgt dafür, dass es in F weder Divergenz noch Laufzeitfehler gibt:

Satz 12.3 (Auswertbarkeit) *Sei* $\emptyset \vdash e : t$. *Dann existiert genau ein Wert* v *mit* $\emptyset \vdash e \triangleright v$.

Der Beweis des Auswertbarkeitssatzes ist nicht einfach. Ein vergleichbarer Satz wurde erstmals um 1940 von Alan Turing bewiesen, einem Pionier der Informatik.

Sei $\emptyset \vdash e : t$ und $\emptyset \vdash e \triangleright v$. Dann sollte v ein Wert des Typs t sein. Hier zeigt sich eine Schwäche in unserer Prozedurdarstellung: Der Zusammenhang zwischen Prozeduren und Typen lässt sich nicht präzise fassen, da wir die Typen der freien Bezeichner des Codes nicht als Teil der Prozedur darstellen. Immerhin können wir den folgenden Satz formulieren:

Satz 12.4 (Typkorrektheit)

1. *Sei* $\emptyset \vdash e : \mathit{int}$ *und* $\emptyset \vdash e \triangleright v$. *Dann* $v \in \mathbb{Z}$.
2. *Sei* $\emptyset \vdash e : \mathit{bool}$ *und* $\emptyset \vdash e \triangleright v$. *Dann* $v \in \{0,1\}$.

Der Beweis dieses Satzes gelingt allerdings nur dann, wenn wir Prozeduren wie oben erwähnt mit hinreichend Typinformation versehen.

Abbildung 12.9 zeigt eine als **Evaluierer** bezeichnete Prozedur

$$\mathit{eval}\colon \mathit{value\ env} \to \mathit{exp} \to \mathit{value}$$

$$\textbf{Dfalse}\ \frac{}{V \vdash \mathit{false} \rhd 0} \qquad \textbf{Dtrue}\ \frac{}{V \vdash \mathit{true} \rhd 1}$$

$$\textbf{Dnum}\ \frac{z \in \mathbb{Z}}{V \vdash z \rhd z} \qquad \textbf{Did}\ \frac{V\,x = v}{V \vdash x \rhd v}$$

$$\textbf{D+}\ \frac{V \vdash e_1 \rhd z_1 \qquad V \vdash e_2 \rhd z_2 \qquad z = z_1 + z_2}{V \vdash e_1 + e_2 \rhd z}$$

$$\textbf{D–}\ \frac{V \vdash e_1 \rhd z_1 \qquad V \vdash e_2 \rhd z_2 \qquad z = z_1 - z_2}{V \vdash e_1 - e_2 \rhd z}$$

$$\textbf{D*}\ \frac{V \vdash e_1 \rhd z_1 \qquad V \vdash e_2 \rhd z_2 \qquad z = z_1 \cdot z_2}{V \vdash e_1 * e_2 \rhd z}$$

$$\textbf{D}\leq\ \frac{V \vdash e_1 \rhd z_1 \qquad V \vdash e_2 \rhd z_2 \qquad z = \text{if } z_1 \leq z_2 \text{ then } 1 \text{ else } 0}{V \vdash e_1 \leq e_2 \rhd z}$$

$$\textbf{Diftrue}\ \frac{V \vdash e_1 \rhd 1 \qquad V \vdash e_2 \rhd v}{V \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rhd v}$$

$$\textbf{Diffalse}\ \frac{V \vdash e_1 \rhd 0 \qquad V \vdash e_3 \rhd v}{V \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rhd v}$$

$$\textbf{Dabs}\ \frac{}{V \vdash \mathit{fn}\ x : t \Rightarrow e \rhd \langle x, e, V\rangle}$$

$$\textbf{Dapp}\ \frac{V \vdash e_1 \rhd \langle x, e, V'\rangle \qquad V \vdash e_2 \rhd v_2 \qquad V'[x := v_2] \vdash e \rhd v}{V \vdash e_1\,e_2 \rhd v}$$

Abbildung 12.8: *Dynamische Semantik von F*

```
datatype value =
    IV    of int
  | Proc  of id * exp * value env

fun evalCon False  = IV 0
  | evalCon True   = IV 1
  | evalCon (IC x) = IV x
fun evalOpr Add (IV x1) (IV x2) = IV(x1+x2)
  | evalOpr Sub (IV x1) (IV x2) = IV(x1-x2)
  | evalOpr Mul (IV x1) (IV x2) = IV(x1*x2)
  | evalOpr Leq (IV x1) (IV x2) = IV(if x1<=x2 then 1 else 0)
  | evalOpr  _      _   _       = raise Error "R Opr"
fun eval f (Con c) = evalCon c
  | eval f (Id x) = f x
  | eval f (Opr(opr,e1,e2)) = evalOpr opr (eval f e1) (eval f e2)
  | eval f (If(e1,e2,e3)) = (case eval f e1 of
                IV 1 => eval f e2
              | IV 0 => eval f e3
              |  _   => raise Error "R If")
  | eval f (Abs(x,t,e)) = Proc(x, e, f)
  | eval f (App(e1,e2)) = (case (eval f e1, eval f e2) of
                (Proc(x,e,f'), v) =>  eval (update f' x v) e
              | _ => raise Error "R App")
```

val eval : value env → exp → value

Abbildung 12.9: *Implementierung der dynamischen Semantik von F*

die den durch die Inferenzregeln gegebenen Auswertungsalgorithmus implementiert. Dabei kommen ähnliche Ideen wie bei der Prozedur *elab* für die semantische Analyse zur Anwendung.

```
val f = update empty "x" (IV 5)
```
val f : value env

```
val e = Opr(Leq, Id"x", Con(IC 7))
```
val e = Opr(Leq, Id "x", Con(IC 7)) : exp

```
eval f e
```
IV 1 : value

Der Auswertbarkeitssatz 12.3 garantiert, dass *eval* für zulässige Ausdrücke immer regulär terminiert (also ohne eine Ausnahme zu werfen). Der Typerhaltungssatz 12.4 garantiert, dass *eval* für zulässige Ausdrücke des Typs *int* [*bool*] immer ein Ergebnis der Form *IV z* [*IV 0* oder *IV 1*] liefert.

Die Inferenzregeln der dynamischen Semantik und die Prozedur *eval* ignorieren die in

den Abstraktionen vermerkten Typen für die Argumentvariablen. Für die dynamische Semantik spielt es also keine Rolle, welche Typen für die Argumentvariablen einer Abstraktion angegeben sind. Man hat daher die Möglichkeit, die in einem Ausdruck angegebenen Typen vor der Ausführung vollständig zu löschen. Davon machen Programmiersysteme für Standard ML aus Effizienzgründen in der Tat Gebrauch.

Aufgabe 12.10 Schreiben Sie einen Umschlag *eval′* für *eval*. Orientieren Sie sich an dem Umschlag für *elab* in § 12.4.

Aufgabe 12.11 Sei e eine Darstellung des Ausdrucks $fn\,x : int \Rightarrow y$. Überlegen Sie sich, welche Ergebnisse die folgenden Aufrufe von *elab* und *eval* liefern.

a) *elab empty e*

b) *eval empty e*

c) *eval empty* (*App*(*e*, *Con*(*IC* 7)))

Aufgabe 12.12 Geben Sie einen möglichst einfachen Ausdruck e an, sodass bei der Ausführung von *eval empty e* jede der 6 Regeln der Prozedur *eval* zum Einsatz kommt.

Aufgabe 12.13 Die Prozedur *eval* kann durch Werfen einer Ausnahme *Error s* einen Fehler s melden. Geben Sie möglichst einfache Ausdrücke an, für die die Prozedur *eval empty* die Fehler "*R Opr*", "*R If*" und "*R App*" meldet. Überprüfen Sie Ihre Antworten mit einem Interpreter.

Aufgabe 12.14 Die Prozedur *eval empty* liefert für viele Ausdrücke Ergebnisse, für die *elab empty* Fehler meldet. Geben Sie für jeden der von *elab* behandelten Fehler einen entsprechenden Ausdruck an.

Aufgabe 12.15 (Selbstanwendung und Rekursion) Der Auswertungssatz besagt, dass die Prozedur *eval empty* für jeden zulässigen Ausdruck terminiert. Interessanterweise gibt es unzulässige Ausdrücke (im Sinne der statischen Semantik), deren Auswertung divergiert. Diese können mit dem Ausdruck $fn\,g : t \Rightarrow g\,g$ gebildet werden, der eine Prozedur beschreibt, die ihre Argumentprozedur g auf sich selbst anwendet. Dieser Ausdruck ist für jeden Argumenttyp t unzulässig.

a) Geben Sie einen (semantisch unzulässigen) Ausdruck an, für den *eval empty* divergiert.

b) Der Logiker Alonzo Church hat um 1930 entdeckt, dass sich prozedurale Rekursion durch Selbstanwendung von Prozeduren simulieren lässt. Versuchen Sie, einen (semantisch unzulässigen) Ausdruck zu finden, der eine Prozedur beschreibt, die die Fakultäten berechnet. Kniffelig!

Aufgabe 12.16 (Paare) Wir wollen F um Paare erweitern. Die abstrakte Syntax und die Menge der Werte erweitern wir wie folgt:

$$t \in Ty = \cdots \mid t * t$$

$$e \in Exp = \cdots \mid (e, e) \mid fst\,e \mid snd\,e$$

$$v \in Val = \mathbb{Z} \cup Pro \cup (Val \times Val)$$

a) Geben Sie die Inferenzregeln für die statische Semantik von Paaren an.
b) Geben Sie die Inferenzregeln für die dynamische Semantik von Paaren an.
c) Erweitern Sie die Deklarationen der Typen *exp* und *value* um Konstruktoren für Paare.
d) Erweitern Sie die Deklaration der Prozedur *elab* um Regeln für Paare.
e) Erweitern Sie die Deklaration der Prozedur *eval* um Regeln für Paare.

12.6 Rekursive Prozeduren

Wir wollen F jetzt um rekursive Prozeduren erweitern. Dafür benötigen wir zusätzliche Syntax. Eine Möglichkeit besteht in der Einführung von Prozedurdeklarationen wie in Standard ML. Wir wählen jedoch die alternative Möglichkeit, die Sprache F um **rekursive Abstraktionen** der Form

$$rfn\, f(x:t):t' \Rightarrow e$$

zu erweitern. Diese führen neben dem Argumentbezeichner x einen frei wählbaren Bezeichner f ein, über den die beschriebene Prozedur im Rumpf e referiert werden kann.[1] Eine Prozedur zur Berechnung der Fakultäten können wir damit wie folgt beschreiben:

```
rfn fac (n:int):int => if n<=0 then 1 else n*fac(n-1)
```

Die statische Semantik rekursiver Abstraktionen definieren wir durch die Inferenzregel

$$\textbf{Srabs}\quad \frac{(T[f := t \to t'])[x := t] \vdash e : t'}{T \vdash rfn\, f(x:t):t' \Rightarrow e : t \to t'}$$

Für die dynamische Semantik stellen wir rekursive Prozeduren durch Tupel $\langle f, x, e, V\rangle$ dar, die in der ersten Komponente den Bezeichner für die Prozedur tragen:

$$Val = \mathbb{Z} \cup Pro \cup RPro$$

$$RPro = Id \times Id \times Exp \times VE$$

Die dynamische Semantik rekursiver Prozeduren definieren wir mit zwei Inferenzregeln, die die Auswertung rekursiver Abstraktionen und die Anwendung rekursiver Prozeduren beschreiben:

$$\textbf{Drabs}\quad \frac{}{V \vdash rfn\, f(x:t):t' \Rightarrow e \rhd \langle f, x, e, V\rangle}$$

$$\textbf{Drapp}\quad \frac{V \vdash e_1 \rhd v_1 \qquad V \vdash e_2 \rhd v_2 \qquad v_1 = \langle f, x, e, V'\rangle \qquad V'' = (V'[f := v_1])[x := v_2] \qquad V'' \vdash e \rhd v}{V \vdash e_1 e_2 \rhd v}$$

[1] Wir arbeiten ab sofort mit einer Grammatik, die die Buchstaben x und f als gleichberechtigte Metavariablen für Bezeichner einführt. Damit vermeiden wir schwer lesbare Formulierungen wie $rfn\, x_1(x_2:t):t' \Rightarrow e$.

Damit ist die Erweiterung von F um rekursive Prozeduren abgeschlossen.

Mit rekursiven Abstraktionen können wir zulässige Ausdrücke schreiben, deren Auswertung divergiert. Damit ist Satz 12.3 (Auswertbarkeit) nicht mehr gültig. Proposition 12.1 auf S. 246 und 12.2 (Determinismus) sowie Satz 12.4 auf S. 251 (Typkorrektheit) gelten jedoch auch weiterhin.

Aufgabe 12.17 Geben Sie passend zu den Gleichungen in Abbildung 12.3 auf S. 244 eine Gleichung an, die die freien Bezeichner rekursiver Abstraktionen beschreibt.

Aufgabe 12.18 Warum ist die rekursive Abstraktion $\mathit{rfn}\, f\,(f : \mathit{int}) : \mathit{int} \Rightarrow f$ gemäß der Regel Srabs semantisch zulässig?

Aufgabe 12.19 (Erweiterung der Prozeduren elab und eval)

a) Erweitern Sie die Deklaration der Typen *exp* und *value* um Konstruktoren für rekursive Abstraktionen.

b) Erweitern Sie die Prozedur *elab* um eine Regel für rekursive Abstraktionen.

c) Erweitern Sie die Prozedur *eval* um eine Regel für rekursive Abstraktionen. Erweitern Sie zudem die Fallunterscheidung in der Regel für Prozeduranwendungen um die Anwendung rekursiver Prozeduren. Erproben Sie Ihre erweiterte Prozedur *eval* mit einer rekursiven Prozedur, die Fakultäten berechnet.

d) Geben Sie einen zulässigen Ausdruck *e* an, sodass die Auswertung von *eval empty e* divergiert.

Aufgabe 12.20 (Deklarationen) Sie wissen jetzt genug, um F in eigener Regie um Deklarationen, Programme und Let-Ausdrücke zu erweitern:

$d \in \mathit{Dek} = \mathit{val}\, x = e \mid \mathit{fun}\, f\,(x : t) : t' = e$

$p \in \mathit{Prg} = d \ldots d$

$e \in \mathit{Exp} = \cdots \mid \mathit{let}\ p\ \mathit{in}\ e$

a) Geben Sie die Inferenzregeln für die dynamische Semantik der neuen Konstrukte an. Orientieren Sie sich dabei an der informellen Beschreibung in § 2.7 und verwenden Sie Aussagen der Form $V \vdash d \triangleright V'$ und $V \vdash p \triangleright V'$.

b) Geben Sie die Inferenzregeln für die statische Semantik der neuen Konstrukte an. Verwenden Sie dabei Aussagen der Form $T \vdash d : T'$ und $T \vdash p : T'$.

c) Implementieren Sie die abstrakte Syntax mit verschränkt rekursiv deklarierten Konstruktortypen *dek*, *prg* und *exp*. Stellen Sie dabei Programme mithilfe von Listen dar. Benutzen Sie das Schlüsselwort *and* anstelle des Schlüsselworts *datatype*, um die verschränkt rekursive Gruppe von Konstruktortypen zu deklarieren (analog zur Deklaration von verschränkt rekursiven Funktionen, siehe § 7.8).

d) Erweitern Sie die Prozedur *elab* mit verschränkt rekursiv deklarierten Hilfsprozeduren *elabDek* und *elabPrg*.

e) Erweitern Sie die Prozedur *eval* mit verschränkt rekursiv deklarierten Hilfsprozeduren *evalDek* und *evalPrg*.

Bemerkungen

Wir sind jetzt in der Lage, die Semantik einfacher Programmiersprachen präzise zu definieren. Dazu benutzen wir Inferenzregeln und abstrakte Grammatiken als formale Beschreibungswerkzeuge. Die syntaktischen und semantischen Objekte einer Sprache werden dabei als mathematische Objekte modelliert. Die mathematische Beschreibung von Programmiersprachen stellt in Hinblick auf Präzision, Kompaktheit, Lesbarkeit und Handhabbarkeit eine gewaltige Verbesserung gegenüber der informellen Beschreibung wie in Kapitel 2 und 3 dar. Sie liefert die Grundlage für eine programmiersprachliche Theorie, in der Zusammenhänge wie Auswertbarkeit oder Typkorrektheit formuliert und bewiesen werden können. Darüber hinaus ist die mathematische Definition einer Programmiersprache eine wichtige Voraussetzung für ihre korrekte Implementierung durch Programmierwerkzeuge.

Grundlagen für die programmiersprachliche Theorie wurden von mathematischen Logikern erarbeitet, bevor die ersten Programmiersprachen entwickelt wurden (z.B. Fortran, etwa 1955). Von besonderer Bedeutung ist der sogenannte Lambda-Kalkül, der ab 1930 von Alonzo Church entwickelt wurde. Syntaktisch gesehen ist die Sprache des Lambda-Kalküls eine Teilsprache von F. Wenn Sie mehr über die Theorie von Programmiersprachen wissen wollen, können Sie in das Buch von Pierce [5] schauen.

In diesem Kapitel haben wir erstmals etwas größere Programme betrachtet. Zusammengenommen ergeben die Deklarationen für die abstrakte Syntax, die Umgebungen sowie die Prozeduren *elab* und *eval* ein interessantes Programm, das einen Interpreter für F darstellt. Diesen Interpreter werden wir im nächsten Kapitel um eine Komponente erweitern, mit der Phrasen in konkreter Syntax eingegeben werden können.

Versäumen Sie nicht, die Aufgaben zu bearbeiten, in denen es darum geht, den Interpreter um die Behandlung von Paaren, rekursiven Prozeduren und Deklarationen zu erweitern. Sie werden dabei wertvolle Erfahrungen mit dem Debugging von Programmen sammeln (Austesten, Fehlersuche und Fehlerkorrektur).

Verzeichnis

Abstrakte und konkrete Syntax; abstrakte Grammatiken; Metavariablen.

Inferenzregeln; Prämissen und Konklusion; Ableitung von Aussagen.

Statische und dynamische Semantik von F; Typ- und Wertumgebungen; Determinismus, Auswertbarkeit und Typkorrektheit; Elaborierer und Evaluierer.

Rekursive Abstraktionen und Darstellung rekursiver Prozeduren.

Umschläge.

13 Konkrete Syntax

Die konkrete Syntax einer Sprache beschreibt die Darstellung der durch die abstrakte Syntax gegebenen Phrasen durch Wörter und Zeichen. Dabei unterscheiden wir zwei Teilaufgaben:

1. Die Darstellung von Bäumen durch Wortfolgen. Dieser Teil der konkreten Syntax wird als **phrasale Syntax** bezeichnet.
2. Die Darstellung von Wortfolgen durch Zeichenfolgen. Dieser Teil der konkreten Syntax wird als **lexikalische Syntax** bezeichnet.

Am Beispiel der im letzten Kapitel eingeführten Sprache F werden wir zeigen, wie lexikalische und phrasale Syntaxen mithilfe von konkreten Grammatiken beschrieben werden können. Darauf aufbauend werden wir Prozeduren entwickeln, die die Zeichen-, Wort- und Baumdarstellung von Phrasen ineinander übersetzen. Algorithmisch gesehen ist dabei vor allem die Übersetzung von Wortdarstellungen in Baumdarstellungen interessant. Schließlich werden wir in der Lage sein, den im vorherigen Kapitel entwickelten Interpreter so zu erweitern, dass Ausdrücke in Zeichendarstellung eingegeben werden können.

13.1 Lexikalische Syntax für Typen

Wir beginnen mit einer lexikalischen Syntax für die Typen von F. Das ist einfach, da wir nur 5 Wörter benötigen: `"bool"`, `"int"`, `"->"`, `"("` und `")"`. Wir vereinbaren, dass diese Wörter in einer Zeichendarstellung entweder direkt aufeinander folgen oder durch sogenannte **Leerzeichen** getrennt sein dürfen. Als Leerzeichen lassen wir die Zeichen Zwischenraum `" "`, Tabulator `"\t"` und Zeilenwechsel `"\n"` zu. Hier sind Beispiele für lexikalisch zulässige Zeichenfolgen und ihre Übersetzung in Wortfolgen:

```
"(int->int) -> int"    ⇝    "("  "int"  "->"  "int"  ")"  "->"  "int"

 "  intbool->int  "    ⇝    "int"  "bool"  "->"  "int"
```

Die Zeichenfolge `"bit"` ist dagegen lexikalisch unzulässig, da sie nicht in eine Folge zulässiger Wörter zerlegt werden kann.

Eine wohldefinierte lexikalische Syntax hat die Eigenschaft, dass es zu einer Zeichendarstellung immer nur eine Wortdarstellung gibt. Dadurch ist die Übersetzung von Zeichendarstellungen in Wortdarstellungen eindeutig bestimmt. Umgekehrt gibt es zu ei-

```
exception Error of string

datatype token = BOOL | INT | ARROW | LPAR | RPAR

fun lex nil = nil
  | lex (#" ":: cr) = lex cr
  | lex (#"\t":: cr) = lex cr
  | lex (#"\n":: cr) = lex cr
  | lex (#"b":: #"o":: #"o":: #"l":: cr) = BOOL:: lex cr
  | lex (#"i":: #"n":: #"t":: cr) = INT:: lex cr
  | lex (#"-":: #">":: cr) = ARROW:: lex cr
  | lex (#"(":: cr) = LPAR:: lex cr
  | lex (#")":: cr) = RPAR:: lex cr
  | lex _ = raise Error "lex"
```

val lex : char list → token list

Abbildung 13.1: *Ein Lexer für Typen*

ner Wortdarstellung jedoch immer mehrere Zeichendarstellungen, da die Wörter durch unterschiedlich viele Leerzeichen getrennt werden können.

Unter einem **Lexer** versteht man eine Prozedur, die prüft, ob eine Zeichenfolge lexikalisch zulässig ist und sie gegebenenfalls in eine Wortfolge übersetzt. Abbildung 13.1 zeigt einen Lexer, der die beschriebene lexikalische Syntax für Typen realisiert. Die erkannten Wörter werden dabei als Werte des Typs *token* dargestellt.

```
lex (explode "(int->bool)->int")
```
[LPAR, INT, ARROW, BOOL, RPAR, ARROW, INT] : token list

```
lex (explode "  intbool->int  ")
```
[INT, BOOL, ARROW, INT] : token list

Wenn man will, kann man die Deklaration des Lexers in Abbildung 13.1 als eine formale Beschreibung der lexikalischen Syntax für die Typen von F ansehen.

Aufgabe 13.1 Deklarieren Sie eine endrekursive Prozedur

lex' : *token list* → *char list* → *token list*

sodass lex' [] *cs* für jede Zeichenfolge *cs* dasselbe Ergebnis liefert wie *lex cs*.

Aufgabe 13.2 Ändern Sie die Deklaration von *lex* so ab, dass die Wörter *bool* und *int* immer durch mindestens ein Leerzeichen getrennt sein müssen.

13.2 Phrasale Syntax für Typen

Die phrasale Syntax einer Sprache beschreibt, wie die Bäume der abstrakten Syntax durch Wortfolgen darzustellen sind. Als Beispiel wollen wir eine phrasale Syntax für die Typen von F definieren, die einerseits das Setzen überflüssiger Klammern erlaubt und andererseits das Weglassen von Klammern rechts des Pfeils ermöglicht:

$$((\mathit{int} \to \mathit{int})) \quad \rightsquigarrow \quad \mathit{int} \to \mathit{int}$$

$$\mathit{int} \to \mathit{int} \to \mathit{int} \quad \rightsquigarrow \quad \mathit{int} \to (\mathit{int} \to \mathit{int})$$

Das Tolerieren überflüssiger Klammern hat zur Folge, dass jeder Typ mit unendlich vielen verschiedenen Wortdarstellungen beschrieben werden kann.

Die präzise Beschreibung der zulässigen Wortdarstellungen gelingt, indem wir zwei Arten von Wortdarstellungen, *ty* und *pty*, unterscheiden und diese verschränkt rekursiv durch zwei **syntaktische Gleichungen** definieren:[1]

$$\begin{aligned} \mathit{ty} &::= \mathit{pty} \mid \mathit{pty}\ \texttt{"->"}\ \mathit{ty} \\ \mathit{pty} &::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ \mathit{ty}\ \texttt{")"} \end{aligned}$$

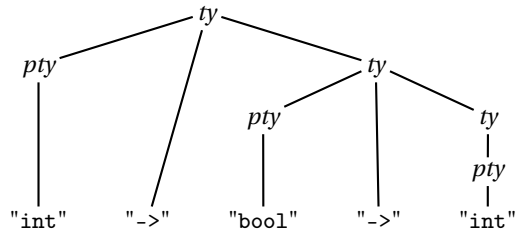
Die erste Gleichung besagt, dass eine Darstellung gemäß *ty* entweder eine Darstellung gemäß *pty* ist, oder eine Darstellung gemäß *pty*, auf die das Wort `"->"` und eine Darstellung gemäß *ty* folgt. Die zweite Gleichung besagt, dass eine Darstellung gemäß *pty* entweder nur aus dem Wort `"bool"` besteht, nur aus dem Wort `"int"` oder aus dem Wort `"("` gefolgt von einer Darstellung gemäß *ty* und dem Wort `")"`.

Beachten Sie, dass jede Darstellung gemäß *pty* auch eine Darstellung gemäß *ty* ist. Die zulässigen Wortdarstellungen für die Typen von F definieren wir als die Darstellungen gemäß *ty*.

Wir bezeichnen *ty* und *pty* als **syntaktische Kategorien**. Das *ty* und *pty* definierende Gleichungssystem bezeichnen wir als **konkrete Grammatik**.[2] Die Wortdarstellungen gemäß einer syntaktischen Kategorie bezeichnen wir als die **Sätze (gemäß) der Kategorie**. Beispielsweise ist $\mathit{int} \to \mathit{int}$ ein Satz gemäß *ty*, aber kein Satz gemäß *pty*. Die **Sätze der Grammatik** sind die Sätze der Kategorien der Grammatik.

Jeder Satz einer konkreten Grammatik kann mithilfe der Gleichungen der Grammatik aus mindestens einer Kategorie der Grammatik **abgeleitet** werden. Die baumartige Darstellung einer solchen Ableitung bezeichnen wir als **Syntaxbaum**. Hier ist ein Syntaxbaum für die Ableitung des Satzes $\mathit{int} \to \mathit{bool} \to \mathit{int}$ aus der Kategorie *ty*:

[1] In diesem Kapitel schreiben wir syntaktische Gleichungen in einem anderen Format als in Kapitel 2.
[2] Konkrete Grammatiken sind eine Spielart der sogenannten *kontextfreien Grammatiken*.



Aufgabe 13.3 Zeichnen Sie Syntaxbäume für *ty* und die folgenden Wortfolgen:

a) $int \to (bool \to int)$

b) $(bool \to bool) \to int$

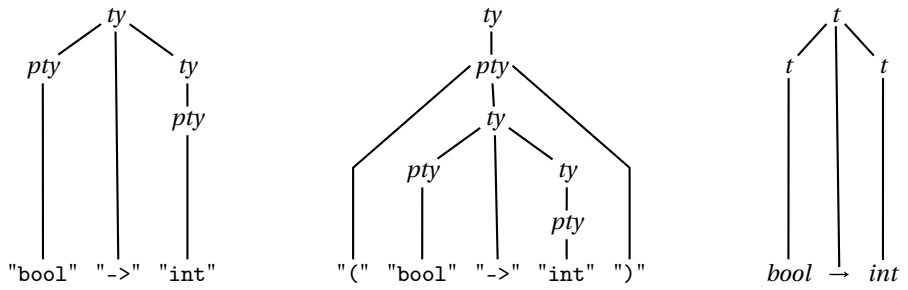
c) $int \to (bool \to bool) \to int$

13.2.1 Affinität

Wir sagen, dass eine konkrete Grammatik **affin** zu einer abstrakten Grammatik ist, wenn jede konkrete Ableitung durch genau eine abstrakte Ableitung und jede abstrakte Ableitung durch mindestens eine konkrete Ableitung simuliert werden kann. Affinität stellt also die notwendige Verbindung zwischen der phrasalen und der abstrakten Syntax einer Sprache her. Erwartungsgemäß ist unsere konkrete Grammatik für die Typen von F affin zu deren abstrakten Grammatik:

$$\begin{array}{rcl} ty & ::= & pty \mid pty \texttt{"->"} ty \\ pty & ::= & \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"} \end{array} \qquad\qquad t = bool \mid int \mid t \to t$$

Machen Sie sich die Affinität an dem folgenden Beispiel klar, das zwei konkrete Ableitungen zeigt, die dieselbe abstrakte Ableitung simulieren:



Der dem Begriff der Affinität zugrundeliegende Simulationsbegriff ist symmetrisch: Eine konkrete Ableitung simuliert eine abstrakte Ableitung genau dann, wenn die abstrakte Ableitung die konkrete Ableitung simuliert. Damit sorgt Affinität dafür, dass jeder Satz der Grammatik mindestens einen abstrakten Typ darstellt, und dass umgekehrt jeder abstrakte Typ durch mindestens einen Satz der konkreten Grammatik dargestellt werden kann.

Da man bei einer von *ty* ausgehenden konkreten Ableitung frei wählen kann, in wie viele redundante Klammerpaare der darzustellende Typ eingeschlossen werden soll, kann jede abstrakte Ableitung durch unendlich viele konkrete Ableitungen simuliert werden.

13.2.2 Eindeutigkeit

Eine konkrete Grammatik heißt **eindeutig**, wenn es zu einem Satz und einer Kategorie höchstens einen Syntaxbaum gibt, der den Satz aus der Kategorie ableitet. Zusammen liefern Affinität und Eindeutigkeit die für Programmiersprachen unverzichtbare Eigenschaft, dass eine zulässige Wortdarstellung immer genau eine Phrase beschreibt.

Unsere Grammatik für die Typen von F ist eindeutig. Das liegt daran, dass zu einer syntaktischen Kategorie A und einer Wortfolge immer nur höchstens eine der Alternativen für A anwendbar ist. Beispielsweise ist für *ty* und `"("` `"bool"` `")"` nur die Alternative *pty* anwendbar.

Wir halten fest, dass man die phrasale Syntax einer Programmiersprache durch eine eindeutige konkrete Grammatik beschreibt, die affin zur abstrakten Grammatik ist.

Aufgabe 13.4 (Mehrdeutige Grammatik) Zeigen Sie mit einem Beispiel, dass die Grammatik *exp* ::= `"x"` | *exp exp* nicht eindeutig ist. Geben Sie eine eindeutige Grammatik an, die dieselben Wortfolgen darstellt.

Aufgabe 13.5 Die beschriebene Grammatik für die Typen von F liefert für jeden Typ unendlich viele Wortdarstellungen. Sie sollen eindeutige Grammatiken angeben, die für jeden Typ genau eine Wortdarstellung liefern. Dabei soll der Typ $(int \to int) \to int \to int$ jeweils wie folgt dargestellt werden:

a) $((int \to int) \to (int \to int))$ (vollständig geklammert)

b) $\to\to int\ int \to int\ int$ (Präfixdarstellung ohne Klammern)

c) $(int \to int) \to int \to int$ (minimal geklammert) kniffelig!

13.3 Parsing durch rekursiven Abstieg

Sei eine konkrete Grammatik und eine syntaktische Kategorie A gegeben. Unter einem **Prüfer für** A verstehen wir eine Prozedur, die für eine Wortfolge entscheidet, ob es sich um einen Satz gemäß A handelt. Unter einem **Parser für** A verstehen wir einen Prüfer für A, der, falls es sich bei der vorgelegten Wortfolge um einen Satz gemäß A handelt, eine Baumdarstellung für die dargestellte Phrase liefert. Ein Parser ist also ein Übersetzer, der Wortfolgen gemäß einer phrasalen Syntax in Bäume gemäß einer abstrakten Syntax übersetzt. Die durch einen Parser geleistete Tätigkeit wird als **Parsing** bezeichnet.

Wenn eine konkrete Grammatik bestimmte Anforderungen erfüllt, können die Gleichungen für ihre Kategorien als abstrakte Algorithmen für die Prüfer für diese Kategorien interpretiert werden. Die so erhaltenen Prüfer können in einem zweiten Schritt zu

Parsern erweitert werden. Im Folgenden beschreiben wir eine algorithmische Interpretation für Grammatiken, die als **rekursiver Abstieg (RA)** bezeichnet wird (engl. recursive descent).

Wir erläutern RA am Beispiel einer besonders einfachen Grammatik, die nur eine syntaktische Kategorie *seq* einführt:

seq ::= `"0"` | `"1"` *seq* | `"2"` *seq seq*

Die Sätze der Grammatik werden mit den Wörtern 0, 1 und 2 gebildet. Beispiele für Sätze gemäß *seq* sind die Folgen 0, 10, 2010 und 2210100; Gegenbeispiele sind die Folgen 00, 21 und 201. Die Grammatik ist eindeutig.

Die algorithmische Interpretation der Gleichung für *seq* gemäß RA liefert den folgenden Prüfer:

```
fun test (0::tr) = tr
  | test (1::tr) = test tr
  | test (2::tr) = test (test tr)
  | test _ = raise Error "test"
```
val test : int list → int list

Die Prozedur *test* prüft, ob die gegebene Liste mit einem Satz gemäß der syntaktischen Kategorie *seq* beginnt. Dabei „liest“ sie die Wörter der Liste Wort für Wort, bis sie entweder einen vollständigen Satz gelesen hat oder ausschließen kann, dass die Liste mit einem Satz beginnt. Im positiven Fall liefert sie die Liste der nicht gelesenen Wörter, im negativen Fall wirft sie eine Ausnahme:

```
test [2,0,1,0]
```
[] : int list

```
test [2,0,1,0,7]
```
[7] : int list

```
test [2,0,1]
```
! Uncaught exception: Error "test"

Die Prozedur *test* geht genau gemäß der Gleichung

seq ::= `"0"` | `"1"` *seq* | `"2"` *seq seq*

für die syntaktische Kategorie *seq* vor. Sie entscheidet zunächst aufgrund des Kopfs der Liste, welche der drei Alternativen vorliegt. Für die Alternative 0 ist keine weitere Aktion erforderlich. Für die Alternative 1 muss der Rest der Liste wieder mit einem Satz gemäß *seq* beginnen, was per Rekursion entschieden wird. Für die Alternative 2 muss die Liste mit zwei aufeinander folgenden Sätzen gemäß *seq* beginnen, was mit zweifacher Rekursion entschieden wird. Die Prozedur *test* terminiert, da die Argumentliste bei jedem Rekursionsschritt um ein Wort kürzer wird.

Eine konkrete Grammatik heißt **RA-tauglich**, wenn die algorithmische Interpretation ihrer Gleichungen die folgenden Eigenschaften erfüllt:

1. Falls es zu einer Rekursion kommt, wurde die Argumentliste um mindestens ein Wort verkürzt.
2. Falls zwischen verschiedenen Alternativen gewählt werden muss, kann die Wahl immer aufgrund des ersten Wortes der Argumentliste erfolgen.

Aus den Gleichungen einer RA-tauglichen Grammatik können wir routinemäßig Prüfer für die durch die Gleichungen definierten Kategorien ableiten, so wie wir das am Beispiel der Gleichung für *seq* gesehen haben.

Wir wollen nun den Prüfer für *seq* zu einem Parser erweitern, der Sätze gemäß *seq* in Bäume des Typs

```
datatype tree = A | B of tree | C of tree * tree
```

übersetzt. Die Darstellung von Bäumen durch Sätze definieren wir wie folgt:

```
fun rep A = [0]
  | rep (B t) = 1 :: rep t
  | rep (C(t,t')) = 2 :: rep t @ rep t'
```
rep : tree → int list

```
rep (C(B A, C(A, A)))
```
[2, 1, 0, 2, 0, 0] : int list

Den dieser Darstellung entsprechenden Parser erhalten wir, indem wir den Prüfer für *seq* so erweitern, dass er die Baumdarstellung des gelesenen Satzes und den Rest der Wortfolge liefert:

```
fun parse (0::tr) = (A, tr)
  | parse (1::tr) = let val (s,ts) = parse tr in (B s, ts) end
  | parse (2::tr) = let val (s,ts) = parse tr
                        val (s', ts') = parse ts
                    in  (C(s,s'), ts') end
  | parse _ = raise Error "parse"
```

*val parse : int list → tree * int list*

```
parse [2,0,1,0,7]
```
*(C(A, B A), [7]) : tree * int list*

```
parse (rep (C(B A, C(A, A))))
```
*(C(B A, C(A, A)), []) : tree * int list*

Aufgabe 13.6 Sei die folgende phrasale Syntax für die Typen von *F* gegeben:

ty ::= `"bool"` | `"int"` | `"->"` *ty ty*

Die Baumdarstellungen der Typen sollen in Standard ML durch Werte des Typs

```
datatype ty  = Bool | Int | Arrow of ty * ty
```

dargestellt werden, die erforderlichen Wörter durch Werte des Typs *token* aus § 13.1.

a) Deklarieren Sie einen Prüfer *test* : *token list* → *token list*.

b) Deklarieren Sie einen Parser *parse* : *token list* → *ty* ∗ *token list*.

c) Deklarieren Sie eine Prozedur *rep* : *ty* → *token list*, die Typen als Wortfolgen darstellt.

d) Deklarieren Sie eine Prozedur *str* : *ty* → *string*, die Typen als Zeichenfolgen darstellt.

Aufgabe 13.7 (Baumrekonstruktion aus der Prälinearisierung) Wir können die Prälinearisierung eines Baums (§ 7.6.3) als eine Wortdarstellung des Baums auffassen.

a) Schreiben Sie eine Prozedur *rep* : *tree* → *int list*, die die Prälinearisierung eines Baums liefert.

b) Schreiben Sie eine Prozedur *parse* : *int list* → *tree* ∗ *int list*, die einen Baum aus seiner Prälinearisierung rekonstruiert. Für jeden Baum *t* soll gelten: *parse*(*rep t*) = (*t*, *nil*).

13.4 Parser für Typen

Wir wollen jetzt einen Parser für die phrasale Syntax der Typen von F schreiben:

ty ::= *pty* | *pty* `"->"` *ty*
pty ::= `"bool"` | `"int"` | `"("` *ty* `")"`

Zunächst stellen wir fest, dass die Gleichung für *ty* nicht RA-tauglich ist, da das erste Wort nicht ausreicht, um sich für eine der Alternativen zu entscheiden. Diese Schwierigkeit lässt sich aber leicht beheben, indem wir die Gleichung mit sogenannten **Optionalklammern** formulieren:

ty ::= *pty* [`"->"` *ty*]
pty ::= `"bool"` | `"int"` | `"("` *ty* `")"`

Jetzt muss ein Prüfer für *ty* die Entscheidung zwischen den beiden Alternativen erst fällen, nachdem er einen Satz gemäß *pty* gelesen hat. Falls dann das erste Wort `"->"` ist, muss noch ein Satz gemäß *ty* gelesen werden, ansonsten sind keine weiteren Wörter erforderlich. Nun ist es einfach, die Prüfer für *ty* und *pty* (wie in Abbildung 13.2 gezeigt) zu deklarieren.

```
ty [INT, ARROW, BOOL, RPAR]
[RPAR] : token list
```

```
fun ty ts = case pty ts of ARROW::tr => ty tr | tr => tr
and pty (BOOL::tr) = tr
  | pty (INT::tr) = tr
  | pty (LPAR::tr) = (case ty tr of RPAR::tr => tr
                       | _ => raise Error "RPAR")
  | pty _ = raise Error "pty"
```

val ty : token list → token list
val pty : token list → token list

Abbildung 13.2: *Prüfer für ty und pty*

```
ty [INT, ARROW, BOOL, ARROW]
```
! Uncaught exception: Error "pty"

```
ty [LPAR, BOOL]
```
! Uncaught exception: Error "RPAR"

Die in Abbildung 13.3 gezeigten Parser für *ty* und *pty* ergeben sich routinemäßig durch Erweiterung der Prüfer. Sie lassen sich übersichtlicher schreiben, wenn man die in Abbildung 13.4 auf S. 268 deklarierten Hilfsprozeduren *match* und *extend* verwendet:

- Mit *extend* können wir bei *ty* den Let-Ausdruck durch die Anwendung *extend* (t, tr) *ty Arrow* ersetzen. Diese setzt den Parser *ty* auf die Wortfolge *tr* an und erhält so einen zweiten Typen t'. Sie liefert dann den Typen $Arrow(t, t')$ und den Rest der Wortfolge als Ergebnis.
- Mit *match* können wir den Case-Ausdruck in der dritten Regel von *pty* durch die Anwendung *match* (*ty tr*) *RPAR* ersetzen. Diese prüft, ob *RPAR* als erstes Wort in der von *ty tr* gelieferten Wortfolge erscheint. Wenn das der Fall ist, liefert sie das Paar aus dem von *ty tr* gelieferten Typen und der Wortfolge, die sich durch Streichen von *RPAR* ergibt. Ansonsten wirft sie die Ausnahme *Error* "*match*".

Abbildung 13.5 zeigt die mit *match* und *extend* formulierten Parser.

Die in Abbildung 13.4 gezeigte Prozedur *parse* konvertiert Parser des Typs *token list* $\to \alpha *$ *token list* in Parser des Typs *token list* $\to \alpha$, indem sie testet, ob alle Wörter gelesen wurden.

```
ty [INT, ARROW, BOOL, RPAR]
```
*(Arrow(Int, Bool), [RPAR]) : ty * token list*

```
parse ty [INT, ARROW, BOOL, RPAR]
```
! Uncaught exception: Error "parse"

```
parse ty [INT, ARROW, BOOL, ARROW, INT]
```
Arrow(Int, Arrow(Bool, Int)) : ty

```
parse ty (lex (explode "int->bool->int"))
```
Arrow(Int, Arrow(Bool, Int)) : ty

```
datatype ty = Bool | Int | Arrow of ty * ty

fun ty ts = (case pty ts of
               (t, ARROW::tr) => let val (t',tr') = ty tr
                                 in  (Arrow(t,t'), tr') end
             | s => s)
and pty (BOOL::tr) = (Bool,tr)
  | pty (INT::tr) = (Int,tr)
  | pty (LPAR::tr) = (case ty tr of
                        (t,RPAR::tr') => (t,tr')
                      | _ => raise Error "pty")
  | pty _ = raise Error "pty"
```

*val ty : token list → ty * token list*
*val pty : token list → ty * token list*

Abbildung 13.3: *Parser für ty und pty*

```
fun match (a,ts) t = if null ts orelse hd ts <> t
                     then raise Error "match"
                     else (a, tl ts)

fun extend (a,ts) p f = let val (a',tr) = p ts in (f(a,a'),tr) end

fun parse p ts = case p ts of
                   (a,nil) => a
                 | _ => raise Error "parse"
```

*val match : α * token list → token → α * token list*
*val extend : α * token list → (token list → β * token list) → (α * β → γ) → γ * token list*
*val parse : (token list → α * token list) → token list → α*

Abbildung 13.4: *Hilfsprozeduren match, extend und parse*

```
fun ty ts = case pty ts of
              (t, ARROW::tr) => extend (t,tr) ty Arrow
            | s => s
and pty (BOOL::tr) = (Bool,tr)
  | pty (INT::tr) = (Int,tr)
  | pty (LPAR::tr) = match (ty tr) RPAR
  | pty _ = raise Error "pty"
```

Abbildung 13.5: *Parser für ty und pty mit match und extend*

Aufgabe 13.8 Deklarieren Sie eine Prozedur $ty : ty \to string$, die die Typen von F durch Zeichenfolgen darstellt. Die Darstellung soll gemäß der in § 13.1 und § 13.2 definierten lexikalischen und phrasalen Syntax erfolgen und nur die unbedingt notwendigen Klammern und Leerzeichen enthalten.

13.5 Arithmetische Ausdrücke

Als Nächstes wollen wir eine konkrete Syntax für arithmetische Ausdrücke entwickeln, die mit Zahlen und Bezeichnern sowie Addition und Multiplikation gebildet werden. Dabei sind vor allem die folgenden Aspekte von Interesse:

1. Für Zahlkonstanten und Bezeichner benötigen wir unendlich viele Wörter.
2. Wir müssen die Klammersparregel Punkt-vor-Strich für $*$ und $+$ realisieren: $x * y + z \rightsquigarrow (x * y) + z$.
3. Wir müssen die Klammersparregel Klammere-links für $+$ und $*$ realisieren: $x + y + z \rightsquigarrow (x + y) + z$ (gerade umgekehrt wie bei Typen).

Abbildung 13.6 zeigt drei Grammatiken, die eine abstrakte, eine phrasale und eine lexikalische Syntax für arithmetische Ausdrücke beschreiben. Die Grammatik für die phrasale Syntax realisiert die Klammersparregeln Punkt-vor-Strich und Klammere-links. Sie ist eindeutig und affin zur abstrakten Grammatik. Die Grammatik für die lexikalische Syntax beschreibt die möglichen Wörter und ihre Zeichendarstellung. Sie lässt offen, welche Wortkombinationen durch Leerzeichen zu trennen sind.

13.5.1 Lexer

Abbildung 13.7 zeigt einen Lexer, der die lexikalische Syntax für arithmetische Ausdrücke implementiert. Die noch offene Worttrennungsfrage wird auf natürliche Weise dadurch gelöst, dass der Lexer jeweils möglichst lange Zahlkonstanten und Bezeichner liest, eine Konvention, die als **maximal munch rule** bezeichnet wird. Damit sind Leerzeichen nur zwischen Bezeichnern und zwischen Zahlkonstanten erforderlich.

```
lex (explode "x1")
```
[ID "x", ICON 1] : token list

```
lex (explode "one two")
```
[ID "one", ID "two"] : token list

```
lex (explode "onetwo")
```
[ID "onetwo"] : token list

Das erste Argument der Prozedur *lexId* ist die reversierte Liste der bereits gelesenen Buchstaben des Bezeichners.

Abstrakte Syntax

$z \in \mathbb{Z}$

$x \in Id$

$e \in Exp = z \mid x \mid e + e \mid e * e$

Phrasale Syntax

$$\begin{aligned} exp &::= [\,exp\ \texttt{"+"}\,]\ mexp \\ mexp &::= [\,mexp\ \texttt{"*"}\,]\ pexp \\ pexp &::= num \mid id \mid \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$

Lexikalische Syntax

$$\begin{aligned} word &::= \texttt{"+"} \mid \texttt{"*"} \mid \texttt{"("} \mid \texttt{")"} \mid num \mid id \\ num &::= [\,\texttt{"\~{}"}\,]\ pnum \\ pnum &::= digit\ [\,pnum\,] \\ digit &::= \texttt{"0"} \mid \cdots \mid \texttt{"9"} \\ id &::= letter\ [\,id\,] \\ letter &::= \texttt{"a"} \mid \cdots \mid \texttt{"z"} \mid \texttt{"A"} \mid \cdots \mid \texttt{"Z"} \end{aligned}$$

Abbildung 13.6: *Syntax für arithmetische Ausdrücke*

```
lexId [] (explode "Aufgabe 5")
```
[ID "Aufgabe", ICON 5] : token list

```
lexId [#"f", #"u", #"A"] (explode "gabe 5")
```
[ID "Aufgabe", ICON 5] : token list

Das erste Argument der Prozedur *lexInt* ist 1 oder ~1, wobei ~1 anzeigt, dass ein negatives Vorzeichen gelesen wurde. Das zweite Argument von *lexInt* ist die Zahl, die sich aus den bisher gelesenen Ziffern ergibt.

```
lex (explode "~3472 Katzen")
```
[ICON ~3472, ID "Katzen"] : token list

```
lexInt ~1 34 (explode "72 Katzen")
```
[ICON ~3472, ID "Katzen"] : token list

Aufgabe 13.9 Schreiben Sie den Lexer für arithmetische Ausdrücke so um, dass er verschränkt endrekursiv wird. Erweitern Sie dazu die Prozeduren *lex*, *lexInt* und *lexId* jeweils um ein zusätzliches Argument des Typs *token list*, das die bereits gelesenen Wörter in reversierter Ordnung enthält.

```
datatype token = ADD | MUL | LPAR | RPAR | ICON of int | ID of string

fun lex nil = nil
  | lex (#" ":: cr) = lex cr
  | lex (#"\t":: cr) = lex cr
  | lex (#"\n":: cr) = lex cr
  | lex (#"+":: cr) = ADD:: lex cr
  | lex (#"*":: cr) = MUL:: lex cr
  | lex (#"(":: cr) = LPAR:: lex cr
  | lex (#")":: cr) = RPAR:: lex cr
  | lex (#"~":: c:: cr) = if Char.isDigit c then lexInt ~1 0 (c::cr)
                          else raise Error "~"
  | lex (c::cr) = if Char.isDigit c then lexInt 1 0 (c::cr)
                  else if Char.isAlpha c then lexId [c] cr
                  else raise Error "lex"
and lexInt s v cs = if null cs orelse not(Char.isDigit (hd cs))
                    then ICON(s*v) :: lex cs
                    else lexInt s (10*v+(ord(hd cs)-ord#"0")) (tl cs)
and lexId cs cs' = if null cs' orelse not(Char.isAlpha (hd cs'))
                   then ID(implode(rev cs)) :: lex cs'
                   else lexId (hd cs' ::cs) (tl cs')
```

val lex : char list → token list
val lexInt : int → int → char list → token list
val lexId : char list → char list → token list

Abbildung 13.7: *Lexer für arithmetische Ausdrücke*

Aufgabe 13.10 (Bezeichner mit Ziffern) Schreiben Sie die Prozedur *lexId* so um, dass sie die folgenden Bezeichner erkennt:

$$\begin{aligned} id &::= letter\ [\,id'\,] \\ id' &::= (\,digit \mid letter\,)\ [\,id'\,] \end{aligned}$$

Bezeichner gemäß dieser Grammatik müssen mit einem Buchstaben begonnen werden und können mit Buchstaben und Ziffern fortgesetzt werden.

13.5.2 Parser

Die Grammatik für die phrasale Syntax arithmetischer Ausdrücke

$$\begin{aligned} exp &::= [\,exp\ \texttt{"+"}\,]\ mexp \\ mexp &::= [\,mexp\ \texttt{"*"}\,]\ pexp \\ pexp &::= num \mid id \mid \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$

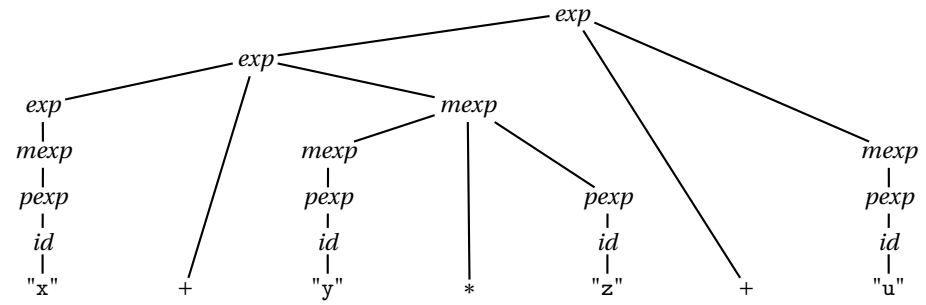


Abbildung 13.8: *Punkt-vor-Strich und Klammere-links im Syntaxbaum*

ist eindeutig und affin zur abstrakten Grammatik (Abbildung 13.6 auf S. 270). Sie realisiert die Klammersparregeln Punkt-vor-Strich und Klammere-links. Der Syntaxbaum in Abbildung 13.8 soll Ihnen helfen, die Struktur der Grammatik zu verstehen.

Die Gleichungen für *exp* und *mexp* sind nicht RA-tauglich, da das erste Wort nicht ausreicht, um zu entscheiden, welche Alternative gewählt werden muss. Wir lösen das Problem dadurch, dass wir die linksrekursiven Gleichungen durch rechtsrekursive ersetzen:

$$\begin{aligned} exp &::= mexp\ [\texttt{"+"}\ exp] \\ mexp &::= pexp\ [\texttt{"*"}\ mexp] \end{aligned}$$

Die neuen Gleichungen sind RA-tauglich. Sie liefern dieselben Sätze wie die ursprünglichen Gleichungen, lassen die arithmetischen Operatoren aber rechts klammern. Diesen Strukturfehler werden wir bei der Synthese der abstrakten Baumdarstellung korrigieren.

Da die neuen Gleichungen RA-tauglich sind, ist die Realisierung von Prüfern eine Routineangelegenheit. Bei der Realisierung der Parser für *exp* und *mexp* müssen wir jetzt aber dafür sorgen, dass Rechtsklammerungen in den abstrakten Bäumen als Linksklammerungen dargestellt werden. Damit diese Umklammerung leicht zu bewerkstelligen ist, führen wir zwei rechtsrekursive **Hilfskategorien** exp' und $mexp'$ ein:

$$\begin{aligned} exp &::= mexp\ exp' \\ exp' &::= [\texttt{"+"}\ mexp\ exp'] \\ mexp &::= pexp\ mexp' \\ mexp' &::= [\texttt{"*"}\ pexp\ mexp'] \end{aligned}$$

Die Einführung der Hilfskategorien ändert nichts an den Sätzen für *exp* und *mexp*. Die Rekursion für die Klammerung der mit "+" oder "*" gebildeten Ausdrucksfolgen läuft jetzt über exp' und $mexp'$. Da die entsprechenden Prozeduren erst aufgerufen werden, nachdem der erste Ausdruck der Folge gelesen wurde, gelingt der linksrekursive Aufbau der abstrakten Baumdarstellung dadurch, dass diese Prozeduren die abstrakte Darstellung der bereits gelesenen Teilfolge als zusätzliches Argument übergeben bekommen. Abbildung 13.9 zeigt die auf diesem Schema basierenden Parser.

```
datatype exp = Con of int | Id of string | Sum of exp * exp
             | Pro of exp * exp

fun exp ts = exp' (mexp ts)
and exp' (e, ADD::tr) = exp' (extend (e,tr) mexp Sum)
  | exp' s = s
and mexp ts = mexp' (pexp ts)
and mexp' (e, MUL::tr) = mexp' (extend (e,tr) pexp Pro)
  | mexp' s = s
and pexp (ICON z :: tr) = (Con z, tr)
  | pexp (ID x :: tr) = (Id x, tr)
  | pexp (LPAR :: tr) = match (exp tr) RPAR
  | pexp _ = raise Error "pexp"
```

*val exp : token list → exp * token list*
*val exp' : exp * token list → exp * token list*
*val mexp : token list → exp * token list*
*val mexp' : exp * token list → exp * token list*
*val pexp : token list → exp * token list*

Abbildung 13.9: *Parser für arithmetische Ausdrücke*

```
mexp' (Id "x", lex (explode "*y*z+u"))
```
*(Pro(Pro(Id "x", Id "y"), Id "z"), [ADD, ID "u"]) : exp * token list*

```
parse exp (lex (explode "2*x+y+(z+u)"))
```
Sum(Sum(Pro(Con 2, Id "x"), Id "y"), Sum(Id "z", Id "u")) : exp

Aufgabe 13.11 (Listen) Wir betrachten Ausdrücke, die mit Bezeichnern und den Operatoren :: und @ gebildet werden. Die phrasale Syntax sei durch die Grammatik

$$\begin{aligned} \mathit{exp} &::= \mathit{pexp}\,[(\texttt{"::"} \mid \texttt{"@"})\,\mathit{exp}] \\ \mathit{pexp} &::= \mathit{id} \mid \texttt{"("}\,\mathit{exp}\,\texttt{")"} \end{aligned}$$

gegeben. Die Operatoren :: und @ klammern also so wie in Standard ML gleichberechtigt rechts: $x :: y @ z :: u @ v \rightsquigarrow x :: (y @ (z :: (u @ v)))$. Wörter und Baumdarstellungen seien wie folgt dargestellt:

```
datatype token = ID of string | CONS | APPEND | LPAR | RPAR

datatype exp = Id of string | Cons of exp * exp | Append of exp * exp
```

a) Schreiben Sie einen Lexer *lex* : *char list* → *token list*.
b) Schreiben Sie einen Parser für *exp*.
c) Schreiben Sie eine Prozedur *exp* : *exp* → *string*, die Ausdrücke gemäß der obigen Grammatik mit minimaler Klammerung darstellt.

Aufgabe 13.12 (Tupel) Wir betrachten Ausdrücke, die mit Bezeichnern und Tupelkonstruktion gebildet werden. Die phrasale Syntax sei durch die Grammatik

$$exp ::= id \mid \texttt{"("}\,[row]\,\texttt{")"}$$
$$row ::= exp\,[\texttt{","}\,row]$$

gegeben. Wörter und Baumdarstellungen stellen wir wie folgt dar:

```
datatype token = ID of string | COMMA | LPAR | RPAR

datatype exp = Id of string | Tup of exp list
```

a) Schreiben Sie einen Lexer $lex : char\ list \to token\ list$.

b) Erweitern Sie die Grammatik um eine Kategorie row', damit sie sich als Grundlage für Parser eignet. Durch die zusätzliche Kategorie soll erreicht werden, dass ein Parser korrekt zwischen redundanter Klammerung und Tupelkonstruktion unterscheiden kann: $((x,(y))) \rightsquigarrow Tup[Id\texttt{"}x\texttt{"},\ Id\texttt{"}y\texttt{"}]$.

c) Schreiben Sie einen Parser für *exp*.

Aufgabe 13.13 (Typen) Wir betrachten Typen, die mit *bool*, *int*, $*$ und $\to$ gebildet werden. Die konkrete Syntax soll der von Standard ML entsprechen. Dabei klammert $*$ vor $\to$ und $\to$ klammert rechts: $int * int * int \to int \to int \rightsquigarrow (int * int * int) \to (int \to int)$. Die mit $*$ dargestellten Produkte können zwei- oder höherstellig sein. Wörter und Baumdarstellungen seien wie folgt dargestellt:

```
datatype token = BOOL | INT | STAR | ARROW | LPAR | RPAR

datatype ty = Bool | Int | Star of ty list | Arrow of ty * ty
```

a) Schreiben Sie einen Lexer $lex : char\ list \to token\ list$.

b) Beschreiben Sie die phrasale Syntax durch eine eindeutige Grammatik, die affin zu der durch den Typ *ty* gegebenen abstrakten Grammatik ist. Verwenden Sie die Kategorien *ty*, *sty* und *pty*.

c) Erweitern Sie die Grammatik um eine Hilfskategorie sty', damit sie sich als Grundlage für Parser eignet. Durch die zusätzliche Kategorie soll erreicht werden, dass die Parser Produkte korrekt übersetzen: $[INT] \rightsquigarrow Int$ und $[INT, STAR, BOOL, STAR, INT] \rightsquigarrow Star[Int, Bool, Int]$.

d) Schreiben Sie Parser für die erweiterte Grammatik.

13.6 Konkrete Syntax für F

Es ist jetzt nicht mehr schwer, eine konkrete Syntax für F zu definieren. Die phrasale Syntax für F definieren wir durch die Grammatik in Abbildung 13.10. Die Grammatik ist eindeutig und affin zu der in Abbildung 12.2 auf S. 243 definierten abstrakten Grammatik für F.

$$
\begin{array}{rcl}
\mathit{ty} & ::= & \mathit{pty}\ [\texttt{"->"}\ \mathit{ty}] \\
\mathit{pty} & ::= & \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ \mathit{ty}\ \texttt{")"} \\
\mathit{exp} & ::= & \texttt{"if"}\ \mathit{exp}\ \texttt{"then"}\ \mathit{exp}\ \texttt{"else"}\ \mathit{exp} \\
 & \mid & \texttt{"fn"}\ \mathit{id}\ \texttt{":"}\ \mathit{ty}\ \texttt{"=>"}\ \mathit{exp} \\
 & \mid & \mathit{aexp}\ [\texttt{"<="}\ \mathit{aexp}] \\
\mathit{aexp} & ::= & [\mathit{aexp}\ (\texttt{"+"} \mid \texttt{"-"})]\ \mathit{mexp} \\
\mathit{mexp} & ::= & [\mathit{mexp}\ \texttt{"*"}]\ \mathit{sexp} \\
\mathit{sexp} & ::= & [\mathit{sexp}]\ \mathit{pexp} \\
\mathit{pexp} & ::= & \texttt{"false"} \mid \texttt{"true"} \mid \mathit{num} \mid \mathit{id} \mid \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{array}
$$

Abbildung 13.10: *Phrasale Syntax für F*

Die lexikalische Syntax für F definieren wir wie folgt. Ein Wort ist entweder eine Zahlkonstante oder ein Bezeichner gemäß der Definition in Abbildung 13.6 auf S. 270, oder eines der folgenden Schlüsselwörter: `"->"`, `"("`, `")"`, `":"`, `"=>"`, `"<="`, `"+"`, `"-"`, `"*"`. Die fehlende Unterscheidung zwischen echten Bezeichnern und den bezeichnerartigen Schlüsselwörtern `"bool"`, `"int"`, `"if"`, `"then"`, `"else"`, `"fn"`, `"false"`, `"true"` holen wir beim Übergang zur abstrakten Wortdarstellung nach:

```
datatype token = ARROW | LPAR | RPAR | COLON (* : *)
               | DARROW (* => *) | LEQ | ADD | SUB | MUL
               | BOOL | INT | IF | THEN | ELSE | FN | FALSE | TRUE
               | ICON of int | ID of string
```

Damit die Grammatik für die phrasale Syntax von F RA-tauglich wird, ersetzen wir die linksrekursiven Gleichungen für *aexp*, *mexp* und *sexp* durch rechtsrekursive. Dabei gehen wir so wie in § 13.5.2 beschrieben vor und korrigieren die resultierenden Strukturfehler bei der Synthese der abstrakten Baumdarstellung. Für *sexp* bekommen wir die Gleichungen

$$
\begin{array}{rcl}
\mathit{sexp} & ::= & \mathit{pexp}\ \mathit{sexp}' \\
\mathit{sexp}' & ::= & [\mathit{pexp}\ \mathit{sexp}']
\end{array}
$$

Die Entscheidung, welche Alternative für sexp' vorliegt, wird wie immer gemäß des Kopfs der Wortfolge gefällt. Wenn es sich dabei um ein Wort handelt, mit dem ein Satz gemäß *pexp* beginnen kann, kann sich der Parser korrekterweise für die Alternative $\mathit{pexp}\ \mathit{sexp}'$ entscheiden. Ein Blick auf die Gleichung für *pexp* zeigt, dass ein Satz gemäß *pexp* nur mit einer Zahlkonstanten, einem Bezeichner, oder einem der Wörter `"false"`, `"true"` oder `"("` anfangen kann.

Aufgabe 13.14 Betrachten Sie applikative Ausdrücke mit der folgenden abstrakten Syntax:

```
datatype exp = Id  of string          (* Identifier  *)
             | App of exp * exp       (* Application *)
```

a) Geben Sie eine eindeutige Grammatik an, die eine phrasale Syntax für diese Ausdrücke beschreibt. Wie in Standard ML soll redundante Klammerung erlaubt sein, und Applikationen sollen bei fehlender Klammerung links geklammert werden. Verwenden Sie die Kategorien *exp*, *pexp* und *id*.

b) Schreiben Sie eine Prozedur $exp : exp \to string$, die applikative Ausdrücke gemäß Ihrer Grammatik mit minimaler Klammerung darstellt.

c) Geben Sie eine rechtsrekursive Grammatik mit einer Hilfskategorie exp' an, aus der sich die Parsingprozeduren ableiten lassen.

d) Schreiben Sie eine Prozedur $test : token\ list \to bool$, die testet, ob eine Liste von Wörtern einen applikativen Ausdruck gemäß Ihrer Grammatik darstellt. Wörter sollen wie folgt dargestellt werden:

```
datatype token = ID of string | LPAR | RPAR
```

Aufgabe 13.15 Schreiben Sie einen Lexer für F.

Aufgabe 13.16 Schreiben Sie Parser für die syntaktischen Kategorien von F.

Aufgabe 13.17 Zu guter Letzt sollen Sie mit den syntaktischen und semantischen Komponenten der letzten zwei Kapitel einen Interpreter für F schreiben. Dieser soll durch eine Prozedur $inter : string \to string$ realisiert werden, die prüft, ob es sich bei ihrem Argument um die Zeichendarstellung eines semantisch zulässigen Ausdrucks handelt und eine Zeichendarstellung des Werts und des Typs des Ausdrucks liefert. Beispielweise soll *inter* die folgenden Ergebnisse liefern:

```
          inter "4+5"          ⇝  "9 : int"
inter "fn x:int => 2*x"        ⇝  "fn: int->int"
      inter "true+5"           ⇝  "! Error: T Opr"
```

Achten Sie darauf, dass auch ganzzahlige Überlauffehler (*Overflow*) korrekt gemeldet werden. Dabei soll erkennbar sein, ob der Fehler bei der lexikalischen Analyse der Zeichendarstellung oder bei der Auswertung des Ausdrucks aufgetreten ist.

Bemerkungen

Sie wissen jetzt, wie man die konkrete Syntax einfacher Programmiersprachen mit Grammatiken beschreiben und mit Lexern und Parsern implementieren kann. Besonders interessant ist dabei die phrasale Syntax, die festlegt, wie die Phrasen der abstrakten

Syntax durch Wortfolgen darzustellen sind. Ihre Beschreibung erfolgt mithilfe einer konkreten Grammatik, die eindeutig und affin zur abstrakten Grammatik sein muss. Durch diese Eigenschaften erreicht man, dass jeder zulässigen Wortfolge genau eine Phrase zugeordnet wird und dass umgekehrt jede Phrase mindestens eine Wortdarstellung hat.

Bei RA-tauglichen Grammatiken kann man die für die Übersetzung von Wortdarstellungen in Baumdarstellungen erforderlichen Parser direkt aus den Gleichungen der Grammatik ableiten (Parsing durch rekursiven Abstieg). Die Klammersparregel Klammere-links führt allerdings zu linksrekursiven Gleichungen, die nicht RA-tauglich sind. Dieses Problem lässt sich aber dadurch lösen, dass man die konkrete Grammatik rechts statt links klammern lässt und diesen Fehler bei der Synthese der abstrakten Baumdarstellung korrigiert.

In der Praxis realisiert man Lexer und Parser meistens mit Werkzeugen, die die entsprechenden Prozeduren automatisch aus annotierten Grammatiken generieren. Diese Werkzeuge prüfen zudem automatisch, ob die für die Eindeutigkeit der Wortdarstellung erforderlichen Eigenschaften erfüllt sind.

Schließlich sei noch erwähnt, dass verschränkte Rekursion bei der Beschreibung und der Verarbeitung konkreter Syntaxen eine wichtige Rolle spielt. Typische Beispiele sind der Lexer und die Parser für arithmetische Ausdrücke, sowie die Grammatik für deren phrasale Syntax.

Verzeichnis

Phrasale und lexikalische Syntax.

Konkrete Grammatiken: syntaktische Kategorien, Alternativen, Sätze, Ableitungen, Syntaxbäume, Affinität, Eindeutigkeit, Optionalklammern.

Lexer und maximal munch rule.

Prüfer und Parser; Parsing durch rekursiven Abstieg.

RA-Tauglichkeit; linksklammernder Aufbau der abstrakten Syntax mit rechtsrekursiven Hilfskategorien.

Baumrekonstruktion aus der Prälinearisierung.

14 Datenstrukturen

In diesem Kapitel geht es um die Realisierung von Datenstrukturen. Dabei unterscheiden wir zwischen der Benutzung und der Implementierung von Datenstrukturen. Die Benutzung einer Datenstruktur erfolgt über eine Schnittstelle, die Operationen bereitstellt, mit denen die Objekte der Datenstruktur gebildet und verwendet werden können. Die Implementierung einer Datenstruktur realisiert die Schnittstelle mit programmiersprachlichen Konstrukten, die als Strukturen, Signaturen und Funktoren bezeichnet werden.

14.1 Strukturen

Zunächst benötigen wir das programmiersprachliche Konstrukt der **Struktur**, mit dem wir verschiedene Objekte zusammenfassen können. Eine Struktur wird mithilfe einer **Strukturdeklaration** eingeführt:

```
structure S = struct
   val a = 4
   fun f x = x+1
end
```
structure S : {val a : int, val f : int → int}

Die gezeigte Strukturdeklaration fasst die Deklarationen für die Bezeichner a und f zusammen. Sie bindet den Bezeichner S an eine Struktur mit zwei **Feldern** a und f, auf die mit den **zusammengesetzten Bezeichnern** $S.a$ und $S.f$ zugegriffen werden kann:

```
S.a
```
4 : int

```
S.f(2*S.a + S.a)
```
13 : int

Erwartungsgemäß sind die Bindungen für die Bezeichner a und f außerhalb der Strukturdeklaration nicht direkt verfügbar.

Jede Folge von Deklarationen kann mit einer Strukturdeklaration zusammengefasst werden:

structure ⟨*Bezeichner*⟩ = struct ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩ end

Bei der Ausführung einer Strukturdeklaration werden die durch sie zusammengefassten Deklarationen wie üblich ausgeführt. Für jeden der durch die zusammengefassten Deklarationen gebundenen Bezeichner bekommt die Struktur ein entsprechendes Feld.

Hier ist ein zweites Beispiel:

```
structure S = struct
    val a = 4
    exception E
    datatype t = A | B
    structure T = struct val a = a+3 end
end
```

Diesmal bekommt die Struktur *S* die Felder *a*, *E*, *t*, *A*, *B* und *T*. Das Feld *T* ist eine Struktur, auf deren Feld *a* mit dem zusammengesetzten Bezeichner *S.T.a* zugegriffen werden kann:

```
S.T.a
7 : int
```

Die Felder *A* und *B* der Struktur *S* sind Konstruktoren und können dementsprechend in Mustern verwendet werden:

```
fun switch S.A = S.B
  | switch S.B = S.A
val switch : S.t → S.t
```

Wir ziehen noch eine hilfreiche Parallele: Strukturdeklarationen realisieren für Deklarationen ein Ordnungsprinzip, das sich in ähnlicher Form bei der Organisation von Dateien mit Dateiordnern wiederfindet.

14.2 Implementierung von Datenstrukturen

In Standard ML können Datenstrukturen mithilfe von Strukturen realisiert werden. Statt von der Realisierung spricht man üblicherweise von der **Implementierung** einer Datenstruktur. Als Beispiel konzipieren wir eine Struktur, die endliche Mengen über ganzen Zahlen bereitstellt. Die Struktur soll die folgenden Felder haben:

type set
val set : int list → set
val union : set → set → set
val subset : set → set → bool

Die Prozedur *set* soll zu einer Liste $[x_1, \ldots, x_n]$ die Menge $\{x_1, \ldots, x_n\}$ liefern, die Prozedur *union* soll die Vereinigung zweier Mengen liefern und die Prozedur *subset* soll die

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end

structure ISet :> ISET = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

Abbildung 14.1: *Eine Struktur für endliche Mengen ganzer Zahlen*

Teilmengenbeziehung testen. Das Feld *type set* (verschieden vom Feld *val set*) soll einen neuen Typ für die durch die Struktur realisierten Mengen bereitstellen.

Wir kommen jetzt zur Implementierung der Struktur. Dabei werden wir Mengen durch Listen darstellen, wobei wir der Einfachheit halber jede Liste als Darstellung einer Menge akzeptieren:

```
type set = int list
```

Damit ist klar, wie die Prozeduren *set*, *union* und *subset* zu deklarieren sind.

Es gibt aber eine Schwierigkeit. Gemäß dieser Deklaration ist der Typ *set* mit dem Gleichheitstest für Listen ausgestattet. Das ist ein Konflikt, da die **Darstellungsgleichheit** (Gleichheit von Listen) nicht mit der **abstrakten Gleichheit** (Gleichheit von Mengen) übereinstimmt. Beispielsweise sind $[1,2]$ und $[2,1]$ zwei verschiedene Listen, die dieselbe Menge darstellen. Daher wäre es wünschenswert, dass bei der Benutzung der Struktur die Implementierung des Typs *set* durch Listen nicht mehr sichtbar ist und auch kein Gleichheitstest für *set* verfügbar ist. Dieses **Verbergen von Implementierungsinformation** lässt sich dadurch bewerkstelligen, dass wir die Struktur mit einem sogenannten **Signaturconstraint** deklarieren.

Abbildung 14.1 zeigt die Deklaration einer Struktur *ISet*, die Mengen wie gewünscht bereitstellt. Zunächst wird eine **Signatur** *ISET* deklariert, die die Felder der Struktur genauso spezifiziert, wie sie bei der Benutzung sichtbar sein sollen. Danach wird die Struktur *ISet* deklariert, wobei das gewünschte Verbergen von Implementierungsinformation durch den mit der Signatur *ISET* formulierten Signaturconstraint :> *ISET* erzielt wird. Auch die Hilfsprozedur *elem* wird durch den Signaturconstraint verborgen. Hier sind Beispiele für die Benutzung von *ISet*:

```
val s = ISet.set [3,2]
val s : set

val s' = ISet.set [5,9,4]
val s' : set

ISet.subset s s'
false : bool
```

Wenn Sie die zusammengesetzten Bezeichner für die Felder von *ISet* stören, können Sie die Felder mit der speziellen Deklaration *open ISet* direkt sichtbar machen. Man spricht vom **Öffnen einer Struktur**.

```
open ISet
type set
val set : int list → set
val subset : set → set → bool
val union : set → set → set

val s = union (set[1,2,3]) (set[3,4])
val s : set

subset (set[1,4]) s
true : bool
```

Allgemein hat eine Strukturdeklaration mit einem Signaturconstraint die folgende Form:

structure ⟨*Bezeichner*⟩ :> ⟨*Signaturausdruck*⟩ =
    struct ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩ *end*

Bei dem Signaturausdruck kann es sich entweder um einen an eine Signatur gebundenen Bezeichner handeln (wie bei der Deklaration von *ISet*) oder um die direkte Angabe einer Signatur mit *sig* … *end*. Wenn eine Struktur mit einem Signaturconstraint deklariert wird, sehen die Benutzer der Struktur die Felder der Struktur genau so, wie sie von der Signatur spezifiziert werden (das gilt auch beim Öffnen von Strukturen). Im Einzelnen kann ein Signaturconstraint die folgenden Funktionen erfüllen:

- Einführung von **abstrakten Typen**, deren Implementierung bei der Benutzung der Struktur nicht sichtbar ist. Beispiel: der Typ *set* der Struktur *Iset*. Die Einführung eines abstrakten Typen *t* geschieht mit der Spezifikation *type t* oder *eqtype t*. Bei der Spezifikation mit *type* wird der Gleichheitstest des implementierenden Typs verborgen, bei der Spezifikation mit *eqtype* wird er sichtbar gemacht.
- Verbergen von Objekten, die nur für die Implementierung der Struktur von Bedeutung sind. Beispiel: Die Prozedur *elem* der Struktur *Iset*.
- Überprüfung von Typannahmen. Für jedes der von der Signatur spezifizierten Felder prüft der Interpreter, ob es von der Struktur gemäß seiner Spezifikation implementiert wird. Dabei darf die Struktur ein Feld allgemeiner implementieren als die Signatur es fordert. Beispiel: Die Struktur *ISet* implementiert das Feld *set* polymorph mit

dem Typschema $\forall\alpha.\ \alpha \to \alpha$, die Signatur *ISET* spezifiziert es monomorph mit dem Typ *int list* → *set*.

Wir halten noch die folgenden Dinge über Strukturen und Signaturen fest:

- Eine Signatur beschreibt die Schnittstelle zwischen der Benutzung und der Implementierung einer Struktur.
- Signaturen verhalten sich zu Strukturen so, wie sich Typen zu Werten verhalten. Wenn eine Struktur ohne Signaturconstraint eingeführt wird, wird sie der Benutzer gemäß einer automatisch abgeleiteten Signatur sehen, die keine Implementierungsdetails verbirgt.
- Per Konvention schreibt man Bezeichner für Strukturen und Signaturen groß. Bezeichner für Signaturen schreibt man darüber hinaus nur mit Großbuchstaben.
- In let-Ausdrücken ist die Deklaration von Strukturen und Signaturen unzulässig.

Aufgabe 14.1 Sei die Struktur *ISet* gegeben.

a) Schreiben Sie eine Prozedur *member* : *int* → *ISet.set* → *bool*, die testet, ob eine Zahl in einer Menge enthalten ist.

b) Schreiben Sie eine Prozedur *empty* : *ISet.set* → *bool*, die testet, ob eine Menge leer ist.

c) Schreiben Sie eine Prozedur *equal* : *ISet.set* → *ISet.set* → *bool*, die testet, ob zwei Mengen gleich sind.

Aufgabe 14.2 Warum ist die folgende Deklaration unzulässig?

```
structure S :> sig eqtype t end = struct
    type t = int -> int
end
```

14.3 Abstrakte Datenstrukturen

Wir unterscheiden zwischen der Spezifikation und der Implementierung einer Datenstruktur. Die **Spezifikation einer Datenstruktur** beschreibt die Typen und Operationen der Datenstruktur aus Benutzersicht, ohne sich auf eine bestimmte Implementierung festzulegen. Wenn die Spezifikation einer Datenstruktur auch darauf verzichtet, die Laufzeiten der Operationen festzulegen, spricht man von einer **abstrakten Datenstruktur** oder einem **abstrakten Datentyp** (kurz **ADT**).

Abstrakte Datenstrukturen können durch eine Signatur und eine **Modellimplementierung** beschrieben werden. Dabei legt die Modellimplementierung die Funktion der Operationen fest (man spricht von der **Semantik der Operationen**), nicht aber ihre Laufzeiten. Die Modellimplementierung kann informell mithilfe mathematischer Datenstrukturen formuliert werden. Ein typisches Beispiel ist die Spezifikation einer abstrakten Datenstrukur ISet, die endliche Mengen über ganzen Zahlen bereitstellt:

type set
val set : int list → set
val union : set → set → set
val subset : set → set → bool

Die Signatur spezifiziert einen abstrakten Typ und drei Operationen, mit denen Mengen gebildet und verglichen werden können. Die Semantik der Operationen können wir, wie im letzten Abschnitt getan, durch die Angabe einer mathematischen Modellimplementierung spezifizieren.

Die in Abbildung 14.1 auf S. 281 gezeigte Strukturdeklaration implementiert die abstrakte Datenstruktur ISet mithilfe von Listen. Sie realisiert die Operation *set* mit konstanter, die Operation *union* mit linearer und die Operation *subset* mit quadratischer Laufzeit (jeweils bezogen auf die Länge der darstellenden Liste).

Wenn man Mengen spezieller durch strikt sortierte Listen darstellt, kann man die Operation *subset* mit linearer Laufzeit realisieren, allerdings auf Kosten der Operation *set*, die dann linear-logarithmische Laufzeit benötigt (siehe Aufgabe 14.3). Das hierbei auftretende Phänomen, eine Operation auf Kosten anderer Operationen schneller realisieren zu können, wird als **Trade-off** bezeichnet.

Die folgenden Bemerkungen sollen Ihnen helfen, den Begriff der abstrakten Datenstruktur besser zu verstehen.

- Eine abstrakte Datenstruktur besteht aus einer Signatur und einer Modellimplementierung.
- Eine Struktur, die eine abstrakte Datenstruktur implementiert, muss gemäß der Signatur der abstrakten Struktur getypt sein und die Semantik der Operationen gemäß der Modellimplementierung realisieren.
- Eine abstrakte Datenstruktur ist im Wesentlichen durch ihre Operationen bestimmt.
- Bei der Benutzung sollen zwei Implementierungen einer abstrakten Datenstruktur bis auf Effizienz nicht unterscheidbar sein.
- Eine abstrakte Datenstruktur stellt das Bindeglied zwischen der Benutzung und der Implementierung einer Struktur dar. Gleichzeitig entkoppelt sie die Benutzung der Struktur von ihrer Implementierung: Um die Struktur zu benutzen, muss man nicht wissen, wie sie implementiert ist, und um die Struktur zu implementieren, muss man nicht wissen, wie sie benutzt wird.
- Bei der Implementierung einer abstrakten Datenstruktur ist es oft sinnvoll, bestimmte Operationen auf Kosten anderer Operationen schneller zu machen (die sogenannten Trade-offs).

Aufgabe 14.3 Die Struktur *ISet* aus Abbildung 14.1 auf S. 281 implementiert die abstrakte Datenstruktur ISet korrekt aber ineffizient. Zum einen kann die eine Menge darstel-

lende Liste Elemente mehrfach enthalten (z.B. bei *ISet.set*[*1*, *1*]). Zum anderen hat die Operation *subset* quadratische Laufzeit (in der Summe der Länge der darstellenden Listen).

a) Schreiben Sie eine Struktur *ISSet*, die Mengen mit strikt sortierten Listen implementiert (§ 5.5). Achten Sie darauf, dass Sie die Operation *set* mit linear-logarithmischer und die Operationen *union* und *subset* mit linearer Laufzeit realisieren.

b) Da die Darstellungsgleicheit für *ISSet.set* mit der abstrakten Gleichheit übereinstimmt, kann sie für die Benutzer der Struktur sichtbar gemacht werden. Deklarieren Sie eine entsprechende Struktur *ISSet'*. Verwenden Sie dabei einen Signaturconstraint, der die gewünschte Signatur direkt angibt (d.h. ohne Rückgriff auf eine Signaturdeklaration).

Aufgabe 14.4 In § 7.8 haben Sie gelernt, dass sich finitäre Mengen eindeutig durch gerichtete Bäume darstellen lassen. Deklarieren Sie eine Struktur *FSet*, die finitäre Mengen mit gerichteten Bäumen gemäß der folgenden Signatur implementiert:

eqtype fset
val fset : fset list → fset
val union : fset → fset → fset
val subset : fset → fset → bool
*val compare : fset * fset → order*

Die Operation *compare* soll Mengen gemäß der lexikalischen Ordnung für reine Bäume vergleichen.

Aufgabe 14.5 (Umgebungen) Unter einer Umgebung wollen wir wie in § 6.4.2 eine Funktion verstehen, die durch Strings realisierte Bezeichner auf bestimmte Werte abbildet. Deklarieren Sie eine Struktur *Env*, die Umgebungen wie folgt implementiert:

type α env
exception Unbound
*val env : (string * α) list → α env*
val lookup : α env → string → α
val update : α env → string → α → α env

- *env* liefert zu einer Liste von Paaren die entsprechende Umgebung. Wenn die Liste zu einem String mehr als ein Paar enthält, soll nur eines der Paare verwendet werden.
- *lookup* liefert zu einer Umgebung f und einem String s den Wert $f\,s$. Falls f auf s nicht definiert ist, wirft *lookup* die Ausnahme *Unbound*.
- *update* liefert zu einer Umgebung f, einem String s und einem Wert x die Umgebung $f[s{:=}x]$.

Deklarieren Sie die Struktur *Env* mit einem Signaturconstraint, der die Signatur direkt angibt (d.h. ohne Rückgriff auf eine Signaturdeklaration).

Aufgabe 14.6 (Listen) Deklarieren Sie eine Struktur *MyList*, die Listen mit der folgenden Signatur bereitstellt:

eqtype α list
val empty : α list
val cons : α → α list → α list
val null : α list → bool
val hd : α list → α
val tl : α list → α list

Achten Sie darauf, dass die Struktur alle Operationen mit konstanter Laufzeit implementiert. Machen Sie sich klar, dass alle anderen Listenoperationen mit den Operationen der Struktur programmiert werden können. Zeigen Sie das am Beispiel von *foldl*.

Aufgabe 14.7 (Puzzle) Für eine abstrakte Datenstruktur stellt sich oft die Frage, ob eine gewünschte Operation mit den Operationen der Struktur programmiert werden kann. Dazu wollen wir eine abstrakte Datenstruktur Puzzle betrachten, die endliche Mengen ganzer Zahlen gemäß der folgenden Signatur bereitstellt:

type set
val set : int list → set
val find : (int → bool) → set → int option

Die Operation *set* soll zu einer Liste die entsprechende Menge liefern, und die Operation *find* soll zu einer Eigenschaft und einer Menge ein Element der Menge liefern, das die Eigenschaft erfüllt.

a) Deklarieren Sie eine Struktur *Puzzle*, die die abstrakte Datenstruktur Puzzle realisiert.
b) Schreiben Sie mithilfe der Operationen von *Puzzle* eine Prozedur *set → int list*, die die Elemente einer Menge als Liste liefert.

14.4 Vektoren

Bei der effizienten Implementierung von Datenstrukturen muss man die Laufzeit aller Operationen im Auge behalten. Manchmal können bestimmte Operationen auf Kosten anderer Operationen schneller gemacht werden. Welches Laufzeitprofil insgesamt am effizientesten ist, wird davon abhängen, welche Anwendungshäufigkeiten man für die Operationen annimmt.

Bei Listen handelt es sich um eine Datenstruktur, die endliche Folgen implementiert. Erwartungsgemäß können mit Listen nicht alle Operationen für Folgen mit konstanter Laufzeit realisiert werden. Dazu gehört unter anderem der Positionszugriff, der zu einer Position den an dieser Position stehenden Wert liefert. Diese über *List.nth* verfügbare Operation hat für Listen lineare Laufzeit.

Man kann Folgen aber so realisieren, dass der Positionszugriff nur konstante Zeit erfordert. Dieser schnelle Zugriff ist die Grundlage für viele effiziente Algorithmen. Stan-

```
eqtype α vector
val fromList : α list → α vector
val tabulate : int * (int → α) → α vector
val sub      : α vector * int → α      (* Subscript *)
val length   : α vector → int
val map      : (α → β) → α vector → β vector
val foldl    : (α * β → β) → β → α vector → β
val foldr    : (α * β → β) → β → α vector → β
val concat   : α vector list → α vector
```

Abbildung 14.2: *Ausschnitt aus der Signatur der Standardstruktur Vector*

dard ML wird dieser Tatsache durch die Standardstruktur *Vector* gerecht, die Folgen mit einem schnellen Positionszugriff als sogenannte **Vektoren** bereitstellt. Abbildung 14.2 zeigt einen Ausschnitt aus der Signatur von *Vector*. Die Operation *fromList* liefert zu einer Liste den entsprechenden Vektor. Die Operationen *tabulate*, *length*, *map*, *foldl*, *foldr* und *concat* verhalten sich wie die entsprechenden Operationen für Listen. Die Operation *sub* realisiert den Positionszugriff. Wenn die angefragte Position nicht existiert, wirft *sub* die Ausnahme *Subscript*.

Es ist üblich, von den **Komponenten** eines Vektors zu sprechen, statt von seinen Elementen. Wie bei Listen sind die **Positionen** eines Vektors mit 0 beginnend nummeriert. Der Positionszugriff *sub* liefert für einen Vektor und eine Zahl k den Wert, der an der k-ten Position des Vektors steht. Dieser Wert wird als die k-**te Komponente** des Vektors bezeichnet.

Wir kommen jetzt zu den Laufzeiten der Vektor-Operationen. Der Positionszugriff *sub* hat, wie bereits gesagt, konstante Laufzeit. Auch *length* hat konstante Laufzeit. Die Operationen *tabulate*, *map*, *foldl* und *foldr* haben die gleichen Laufzeiten wie bei Listen. Die Laufzeiten der Operationen *fromList* und *concat* sind linear (bei *concat* in der Summe der Längen der zu konkatenierenden Vektoren).

Listen haben gegenüber Vektoren den Vorteil, dass die Cons-Operation konstante Laufzeit hat. Eine Cons-Operation für Vektoren hätte dagegen lineare Laufzeit in der Länge des zweiten Arguments (Realisierung beispielsweise mit *concat* möglich). Der Tradeoff zwischen Vektoren und Listen besteht also in der Wahl zwischen einem schnellen Positionszugriff und einer schnellen Cons-Operation. Eine schnelle Cons-Operation bringt Vorteile, wenn Folgen Schritt für Schritt konstruiert werden müssen. Wenn dagegen wiederholt auf die Elemente einer einmal konstruierten Folge zugegriffen werden soll, ist ein schneller Positionszugriff vorteilhaft.

Wir halten fest, dass Vektoren in Bezug auf Effizienz einen neuen Aspekt in die Sprache einbringen, der nicht auf die bisher eingeführten Sprachkonstrukte zurückgeführt werden kann.

Die folgenden Bindungen für Vektoren sind vordeklariert:

```
type 'a vector = 'a Vector.vector
val vector = Vector.fromList
```

Aufgabe 14.8 Schreiben Sie eine Prozedur *vector2list* : α *vector* $\to$ α *list*, die zu einem Vektor die entsprechende Liste liefert.

Aufgabe 14.9 Schreiben Sie eine Prozedur *cons* : $\alpha \to \alpha$ *vector* $\to \alpha$ *vector*, die zu x und $[x_1, \ldots, x_n]$ den Vektor $[x, x_1, \ldots, x_n]$ liefert. Verwenden Sie dabei *Vector.concat*.

Aufgabe 14.10 Deklarieren Sie eine Struktur *MyVector*, die Vektoren gemäß der Signatur in Abbildung 14.2 als Listen implementiert.

Aufgabe 14.11 Mit Vektoren lassen sich Funktionen, deren Definitionsbereich ein endliches Intervall der ganzen Zahlen ist, mit konstanter Laufzeit berechnen. Schreiben Sie eine Prozedur *vectorize*: $(int \to \alpha) \to int \to int \to \alpha$, die zu einer Prozedur p und zwei Zahlen m, n eine Prozedur mit konstanter Laufzeit liefert, die für alle Zahlen $\{x \in \mathbb{Z} \mid m \leq x \leq n\}$ dasselbe Ergebnis wie p liefert und ansonsten die Ausnahme *Subscript* wirft. Nehmen Sie dabei an, dass p auf allen Zahlen zwischen m und n terminiert. Hinweis: Schreiben Sie die Prozedur *vectorize* so, dass sie zunächst den Vektor $[p\,m, \ldots, p\,n]$ bestimmt.

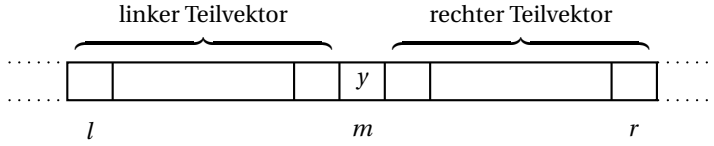
14.5 Binäre Suche

Nehmen Sie an, dass Sie in einem Telefonbuch den Eintrag für einen Namen suchen. Eine Möglichkeit besteht darin, die Einträge beginnend mit Seite 1 der Reihe nach durchzuarbeiten. Dieses zeitaufwendige Verfahren bezeichnet man als **lineare Suche**. Alternativ kann man das Telefonbuch etwa in der Mitte aufschlagen, prüfen ob der gesuchte Name gemäß der alphabetischen Sortierung weiter vorne oder weiter hinten stehen muss, und in der richtigen Hälfte mit derselben Technik weitersuchen. Dieses schnell zum Erfolg führende Verfahren wird als **binäre Suche** bezeichnet. Lineare Suche hat lineare und binäre Suche hat logarithmische Komplexität in der Anzahl der Einträge. Wenn man die Anzahl der Einträge verdoppelt, verdoppelt sich bei linearer Suche der Arbeitsaufwand, bei binärer Suche ist dagegen nur ein Halbierungsschritt mehr erforderlich.

Beim Programmieren bietet es sich an, das Telefonbuch durch mehrere gleich lange Vektoren darzustellen, wobei die Komponenten einer Position zusammen einen Eintrag darstellen. Beispielsweise könnten wir je einen Vektor für Namen, Telefonnummern und Adressen haben. Das Suchproblem besteht dann darin, für einen Namen die erste Position zu finden, an der der Name und die ihm entsprechende Telefonnummer und Adresse abgelegt sind.

Gemäß dieser Motivation können wir jetzt eine polymorphe Prozedur

$$position : (\alpha * \alpha \to order) \to \alpha\ vector \to \alpha \to int\ option$$




```
fun position compare v x = let
    fun position' l r =
        if l>r then NONE
        else let val m = (l+r) div 2
                 val y = Vector.sub(v,m)
             in case compare(x,y) of
                 EQUAL => SOME m
               | LESS => position' l (m-1)
               | GREATER => position' (m+1) r
             end
in
    position' 0 (Vector.length v - 1)
end
```

*position : (α * α → order) → α vector → α → int option*

Abbildung 14.3: *Binäre Suche*

schreiben, die zu einer Vergleichsprozedur, einem Vektor und einem Wert mit binärer Suche eine Position ermittelt, an der der Wert im Vektor vorkommt. Dabei wollen wir wie folgt vorgehen. Zunächst wird die Länge des Vektors ermittelt. Wenn die Länge 0 ist, ist x keine Komponente des Vektors und die Suche ist beendet. Ansonsten werden die mittlere Position m des Vektors und die Komponente y an dieser Position bestimmt. Dann werden 3 Fälle unterschieden:

1. Wenn $x = y$, dann ist m die gesuchte Position und die Suche ist beendet.
2. Wenn $x < y$, dann wird die Suche im linken Teilvektor (bis Position $m-1$) fortgesetzt, da alle Komponenten ab Position m größer als x sind.
3. Wenn $y < x$, dann wird die Suche im rechten Teilvektor (ab Position $m+1$) fortgesetzt, da alle Komponenten bis Position m kleiner als x sind.

Abbildung 14.3 zeigt die Deklaration einer diesem Vorgehen entsprechenden Prozedur. Die eigentliche Suche erfolgt mit der Hilfsprozedur *position'*, die die linkeste und rechteste Position des zu durchsuchenden Teilvektors als Argumente erhält. Die Laufzeit der Prozedur ist logarithmisch in der Länge des Vektors, da binäre Suche zum Einsatz kommt und keine Nebenkosten anfallen. Dabei nehmen wir an, dass die Vergleichsprozedur konstante Laufzeit hat.

Beachten Sie, dass sich die logarithmische Laufzeit der binären Suche nur mit Vektoren

erreichen lässt, da bei Listen der Positionszugriff zu teuer ist. Damit sind wir erstmals auf das weit verbreitete Phänomen gestoßen, dass die Existenz eines effizienten Algorithmus von der Wahl der richtigen Datenstruktur abhängt.

Aufgabe 14.12 Deklarieren Sie eine Struktur *IVSet*, die Mengen ganzer Zahlen mit strikt sortierten Vektoren gemäß der folgenden Signatur implementiert:

eqtype set
val set : int list → set
val subset : set → set → bool
val elem : int → set → bool

Dabei soll *set* linear-logarithmische, *subset* lineare und der Elementtest *elem* logarithmische Komplexität haben.

14.6 Schlangen und Darstellungsinvarianten

Wir betrachten eine weitere Datenstruktur für Folgen, bei der Folgen als Schlangen bezeichnet werden. Für Schlangen geben wir zwei Implementierungen an, eine ineffiziente und eine effiziente. Im Zusammenhang mit der effizienten Implementierung lernen wir den wichtigen Begriff der Darstellungsinvariante kennen.

Abbildung 14.4 zeigt die Signatur für **Schlangen** und eine Modellimplementierung, die Schlangen durch Listen realisiert. Die Operation *snoc* liefert zu einer Schlange und einem Wert die Schlange, die sich durch Anfügen des Wertes an das Ende der Schlange ergibt. Der Name *snoc* ergibt durch Umdrehen von *cons*. Die Operation *head* liefert das erste Element einer Schlange, und die Operation *tail* liefert zu einer Schlange die Schlange, die sich durch Entfernen des ersten Elements ergibt. Wenn die Operationen *head* und *tail* auf die leere Schlange angewendet werden, werfen sie die Ausnahme *Empty*.

Bei der Modellimplementierung fällt für eine Folge von n Erweiterungsschritten mit der Operation *snoc*, die nacheinander auf die leere Schlange angewendet werden, die quadratische Laufzeit $O(n^2)$ an. Das liegt daran, dass die Laufzeit jedes Erweiterungsschritts linear von der Länge der zu erweiternden Schlange abhängt. Interessanterweise existiert jedoch eine effizientere Implementierung für Schlangen, die die quadratische Laufzeit der Modellimplementierung vermeidet und bei der eine Folge von n Operationen (egal welche) ausgehend von der leeren Schlange immer lineare Laufzeit $O(n)$ hat.

Bei der effizienten Implementierung wird eine Schlange durch ein Paar (q,r) aus zwei Listen dargestellt, für das zwei Bedingungen gelten müssen, die zusammengenommen als die **Darstellungsinvariante** der Implementierung bezeichnet werden:

1. Die Liste $q \mathbin{@} \mathit{rev}\, r$ stellt die Schlange dar.
2. Wenn q leer ist, dann ist auch r leer.

```
signature QUEUE = sig
    type 'a queue
    val empty  : 'a queue
    val snoc : 'a queue -> 'a -> 'a queue
    val head   : 'a queue -> 'a              (* Empty *)
    val tail   : 'a queue -> 'a queue        (* Empty *)
end

structure Queue :> QUEUE = struct
    type 'a queue = 'a list
    val empty = nil
    fun snoc q x = q@[x]
    val head = hd
    val tail = tl
end
```

Abbildung 14.4: *Schlangen*

```
structure FQueue :> QUEUE = struct
    type 'a queue = 'a list * 'a list
    val empty = ([],[])
    fun snoc ([],_) x = ([x],[])
      | snoc (q, r) x = (q, x::r)
    fun head (q, r) = hd q
    fun tail ([x], r) = (rev r, [])
      | tail ( q , r) = (tl q, r)
end
```

Abbildung 14.5: *Effiziente Implementierung von Schlangen*

Aus der Darstellungsinvariante ergibt sich die in Abbildung 14.5 gezeigte Implementierung von Schlangen. Jede Operation kann davon ausgehen, dass sie nur auf Darstellungen von Schlangen angewendet wird, für die die Darstellungsinvariante erfüllt ist. Umgekehrt müssen die Operationen *empty*, *snoc* und *tail* aber auch sicherstellen, dass sie Darstellungen liefern, die die Invariante erfüllen. Überzeugen Sie sich davon, dass dies in der Tat der Fall ist.

Wir kommen jetzt zur Laufzeit der Operationen. Bis auf *tail* haben alle Operationen konstante Laufzeit, *tail* hat nur dann keine konstante Laufzeit, wenn ein Reversionsschritt erforderlich ist. Wenn *tail* eine lange Liste zu reversieren hat, profitieren die nachfolgenden *tail*-Operationen davon, da sie dann nicht reversieren müssen. Da jeder Erweiterungsschritt höchstens ein Element zur rechten Liste hinzufügt, sind die Reversionskosten durch die Anzahl der Erweiterungsschritte beschränkt. Wir können also sagen,

dass alle Operationen der effizienten Implementierung von Schlangen **akkumuliert betrachtet** konstante Laufzeit haben.

Die Korrektheit der effizienten Implementierung von Schlangen steht und fällt mit der Einhaltung der Darstellungsinvariante. Beispielsweise liefern die Operationen *snoc*, *head* und *tail* für die unzulässige Darstellung $([], [2,1])$ falsche Ergebnisse. In diesem Zusammenhang ist es von entscheidender Bedeutung, dass die Typdisziplin aufgrund der abstrakten Typen für Schlangen sicherstellt, dass Benutzer der Struktur *FQueue* keine unzulässigen Darstellungen von Schlangen fabrizieren können. Da es sich bei *Queue.queue* und *FQueue.queue* um zwei verschiedene abstrakte Typkonstruktoren handelt, ist übrigens auch sichergestellt, dass die Darstellungen der beiden Implementierungen nicht vertauscht werden können.

In Aufgabe 14.3 auf S. 284 haben Sie Mengen durch strikt sortierte Listen dargestellt. Auch für diese Darstellung gibt es eine Invariante, die wir für eine darstellende Liste *xs* wie folgt formulieren können:

1. Die Elemente der Liste *xs* sind die Elemente der dargestellten Menge.
2. Die Menge *xs* ist strikt sortiert.

Aufgabe 14.13 Erweitern Sie die effiziente Implementierung von Schlangen um die folgenden Operationen:

a) Eine Operation $cons : \alpha\ queue \to \alpha \to \alpha\ queue$, die zu einer Schlange und einem Wert die Schlange liefert, die sich durch Anfügen des Wertes am Anfang der Schlange ergibt.

b) Eine Operation $length : \alpha\ queue \to int$, die die Länge einer Schlange mit konstanter Laufzeit liefert. Stellen Sie Schlangen dabei durch Tripel (q, r, n) dar und formulieren Sie die Invariante für die neue Darstellung.

Aufgabe 14.14 (Priorisierte Schlangen) Bei den Einträgen einer priorisierten Schlange handelt es sich um Paare (p, x), die aus einer Priorität $p \in \mathbb{Z}$ und einem Wert x bestehen. Bei der Erweiterung einer priorisierten Schlange um einen Eintrag (p, x) wird dieser so in die Schlange eingetragen, dass alle bisherigen Einträge mit einer Priorität $\geq p$ vor ihm und alle bisherigen Einträge mit einer Priorität $< p$ nach ihm stehen.

a) Geben Sie eine Signatur für priorisierte Schlangen an.

b) Geben Sie eine Implementierung für priorisierte Schlangen an.

c) Geben Sie die Darstellungsinvariante Ihrer Implementierung an.

14.7 Funktoren

Um Mengen effizient darstellen zu können, benötigt man eine Ordnung für den Elementtyp (siehe Aufgabe 14.3 auf S. 284). Wenn wir Mengen nicht für jeden Elementtyp getrennt implementieren wollen, können wir Mengen mit einem sogenannten **Funktor** implementieren, der den Elementtyp und die Ordnung als Argumente bekommt und

```
functor Set
(
    type t
    val compare : t * t -> order
)
:>
sig
    type set
    val set : t list -> set
    val union : set -> set -> set
    val subset : set -> set -> bool
end
=
struct
    type set = t list
    val set = ssort compare
    val union = smerge compare
    val subset = ssublist compare
end
```

Abbildung 14.6: *Ein Funktor für endliche Mengen*

eine entsprechende Struktur als Ergebnis liefert. Wir können den Funktor dann auf verschiedene Elementtypen und verschiedene Ordnungen anwenden, um entsprechende Strukturen zu konstruieren.

Abbildung 14.6 zeigt die Deklaration eines solchen Funktors *Set*. Dabei nehmen wir an, dass die polymorphen Prozeduren *ssort*, *smerge* und *ssublist* bereits deklariert sind und die erforderlichen Operationen bereitstellen. Um eine Struktur für Mengen über Strings zu bekommen, können wir den Funktor wie folgt anwenden:

```
structure StringSet = Set(type t = string
                          val compare = String.compare)
```

Durch diese Deklaration wird der Strukturbezeichner *StringSet* mit der folgenden Signatur getypt:

type set
val set : string list → set
val union : set → set → set
val subset : set → set → bool

Auf den ersten Blick mag die Syntax für Funktordeklarationen verwirrend erscheinen.

Sie ergibt sich aber quasi von selbst, wenn man mit einer Strukturdeklaration

structure ⟨*Bezeichner*⟩
:> *sig* ⟨*Spezifikation*⟩ … ⟨*Spezifikation*⟩ *end*
= *struct* ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩ *end*

beginnt, das Schlüsselwort *structure* durch *functor* ersetzt und die Parameterspezifikationen nach dem Strukturbezeichner einfügt:

functor ⟨*Bezeichner*⟩
(⟨*Spezifikation*⟩ … ⟨*Spezifikation*⟩)
:> *sig* ⟨*Spezifikation*⟩ … ⟨*Spezifikation*⟩ *end*
= *struct* ⟨*Deklaration*⟩ … ⟨*Deklaration*⟩ *end*

Diese Herangehensweise entspricht der Tatsache, dass ein Funktor schließlich nur eine parametrisierte Struktur ist. Die Anwendung eines Funktors auf konkrete Parameter liefert eine Struktur und kann im Rahmen einer Strukturdeklaration geschehen:

structure ⟨*Bezeichner*⟩ = ⟨*Bezeichner*⟩ (⟨*Deklaration*⟩ … ⟨*Deklaration*⟩)

Aufgabe 14.15 Schreiben Sie polymorphe Prozeduren *ssort*, *smerge* und *ssublist*, wie sie für die Deklaration des Funktors *Set* in Abbildung 14.6 erforderlich sind.

Aufgabe 14.16 Deklarieren Sie mithilfe des Funktors *Set* eine Struktur *IntSet*, die Mengen über ganzen Zahlen bereitstellt.

Aufgabe 14.17 Deklarieren Sie einen Funktor *VSet*, der Mengen über einem geordneten Elementtyp *t* durch strikt sortierte Vektoren gemäß der folgenden Signatur implementiert:

type set
val set : t list → set
val subset : set → set → bool
val elem : t → set → bool

Dabei soll *set* linear-logarithmische, *subset* lineare und *elem* logarithmische Komplexität haben. In Aufgabe 14.12 auf S. 290 haben Sie den algorithmischen Teil der Aufgabe bereits für den Elementtyp *int* gelöst.

Bemerkungen

In diesem Kapitel haben wir den Begriff der Datenstruktur mithilfe von Beispielen präzisiert. Eine Datenstruktur ist im Wesentlichen durch einen Satz von Operationen gegeben, mit denen die Objekte der Struktur gebildet und benutzt werden können. Unter einer abstrakten Datenstruktur verstehen wir eine modellhafte Datenstruktur, die auf keine bestimmte Implementierung festgelegt ist. Bei der Implementierung einer abstrakten

Datenstruktur geht es darum, die Operationen der Struktur mit der richtigen Laufzeit zu realisieren. Da die Beschleunigung bestimmter Operationen oft die Verlangsamung anderer Operationen nach sich zieht, muss diese Aufgabe im Rahmen eines Anforderungsprofils gelöst werden. Als Beispiele für Datenstrukturen mit solchen Trade-offs haben wir Vektoren und Schlangen betrachtet.

In Standard ML können Datenstrukturen mithilfe von Strukturen, Signaturen und Funktoren realisiert werden. Diese Konstrukte werden zusammenfassend als **Module** bezeichnet. Der Begriff rührt daher, dass Module eigentlich für die Strukturierung großer Programme entwickelt wurden.

Wir haben erwähnt, dass Moduldeklarationen in Standard ML nicht lokal im Rahmen eines Let-Ausdrucks erfolgen dürfen. Diese Einschränkung hat mit dem Design der Sprache zu tun, das die Modulsprache als eine konservative Erweiterung der Kernsprache einführt. Das Design der Modulsprache von Standard ML ist neueren Datums (etwa 1990) und hat eine Leitfunktion für andere Programmiersprachen.

Verzeichnis

Strukturen: Felder und zusammengesetzte Bezeichner, Öffnen von Strukturen.

Signaturen: Strukturdeklarationen mit Signaturconstraints, Verbergen von Implementierungsinformation, abstrakte Typen mit und ohne Gleichheitstest.

Abstrakte Datenstrukturen: Operationen, Modellimplementierungen, abstrakte Gleichheit versus Darstellungsgleichheit, Benutzung versus Implementierung, Trade-offs bei den Laufzeiten der Operationen.

Vektoren: Komponenten und Positionen, lineare und binäre Suche.

Darstellung von endlichen Mengen mit strikt sortierten Listen und Vektoren.

Schlangen; Akkumulierte Laufzeit.

Darstellungsinvarianten.

Funktoren.

15 Speicher und veränderliche Objekte

Wir erweitern unser programmiersprachliches Repertoire jetzt um Operationen, mit denen der Zustand eines Speichers abgefragt und verändert werden kann. Die Speicheroperationen versetzen uns in die Lage, veränderliche Objekte darzustellen. Damit ergeben sich neue Datenstrukturen und neue Algorithmen.

Mathematische Objekte sind gedankliche Objekte, die per Konstruktion unveränderlich sind. Denken Sie etwa an die Zahl 12. Mathematische Objekte existieren unabhängig vom Begriff der Zeit. Daher macht es keinen Sinn, von der Lebensdauer eines mathematischen Objektes zu sprechen.

Physikalische Objekte sind veränderliche Objekte. Sie ändern ihren Zustand im Laufe der Zeit. Stellen Sie sich eine Uhr vor. Sie muss ihren Zustand mit der Zeit verändern, sonst kann sie nicht die aktuelle Uhrzeit anzeigen. Sie können den Zustand der Uhr auch von außen ändern, indem Sie eine andere Uhrzeit einstellen.

Computer sind physikalische Objekte, mit denen gedankliche Objekte für eine gewisse Zeit realisiert werden können. Stellen Sie sich einen geöffneten Interpreter auf dem Desktop Ihres Computers vor. Wenn Sie eine Deklaration eingeben, realisiert der Interpreter das deklarierte Objekt, sodass Sie damit arbeiten können.

Computer können veränderliche und unveränderliche Objekte darstellen. Ein Beispiel für ein auf einem Computer realisierbares veränderliches Objekt ist ein Interpreter. Zunächst können Sie einen Interpreter erzeugen (starten), indem Sie das den Interpreter beschreibende Programm zur Ausführung bringen. Danach können Sie den Zustand des Interpreters durch die Eingabe von Deklarationen verändern. Schließlich können Sie den Interpreter beseitigen (beenden). Der Interpreter hat dann für die Zeitspanne zwischen seiner Erzeugung und seiner Beseitigung existiert. Jedes Mal, wenn Sie das den Interpreter beschreibende Programm zur Ausführung bringen, erzeugen Sie einen neuen Interpreter. Wenn Sie wollen, können Sie mehrere Interpreter erzeugen, die gleichzeitig auf demselben Computer existieren.

15.1 Zellen und Referenzen

Wir erweitern das Ausführungsmodell unserer Programmiersprache jetzt um ein veränderliches Objekt, das wir als **Speicher** bezeichnen. Wir stellen uns den Speicher als ein physikalisches Objekt vor, das in sogenannte **Zellen** unterteilt ist. In jeder Zelle wird ein Wert dargestellt. Die Zellen des Speichers sind durchnummeriert und die Nummern der Zellen werden als **Referenzen** bezeichnet. Für Programme ist der Speicher als abstrakte Datenstruktur sichtbar:

eqtype α ref	**Referenztypen**
ref : $\alpha \rightarrow \alpha$ *ref*	**Allokation**
! : α *ref* $\rightarrow \alpha$	**Dereferenzierung**
:= : α *ref* $* \alpha \rightarrow$ *unit*	**Zuweisung**

Die Allokationsoperation *ref* wählt eine bisher nicht benutzte Zelle aus, schreibt die Darstellung eines Werts in die Zelle und liefert die Referenz der Zelle. Wir sagen, dass die Allokationsoperation eine Zelle **alloziert**. Die Dereferenzierungsoperation ! liefert zu einer Referenz den in der entsprechenden Zelle dargestellten Wert. Die Zuweisungsoperation := schreibt die Darstellung eines Werts in eine bereits allozierte Zelle. Dabei wird die bisher in der Zelle befindliche Darstellung überschrieben. Man sagt, dass man **einen Wert in eine Zelle legt**, wenn man seine Darstellung in die Zelle schreibt. Zusammenfassend bezeichnen wir die Operationen *ref*, ! und := als **Speicheroperationen**. Alle drei Speicheroperationen haben konstante Laufzeit.

Der Typkonstruktor *ref* liefert die sogenannten Referenztypen. Eine Referenz des Typs *t ref* identifiziert eine Zelle, in der Werte des Typs *t* liegen können. Die abstrakten Referenztypen stellen sicher, dass nur solche Referenzen im Umlauf sind, deren Zellen durch die Allokationsoperation alloziert wurden.

Sie wissen jetzt genug, um die folgende Interaktion zu verstehen:

```
val r = ref 0
val r : int ref

(!r, r:=25, !r)
(0, (), 25) : int * unit * int

!r+6
31 : int
```

Offensichtlich spielt die Reihenfolge, in der die Teilausdrücke eines zusammengesetzten Ausdrucks ausgeführt werden, bei der Verwendung von Speicheroperationen eine wichtige Rolle. Wie Sie bereits wissen, werden Teilausdrücke in Standard ML immer von links nach rechts ausgeführt.

Es ist wichtig, zwischen einer Referenz und der durch die Referenz identifizierten Speicherzelle zu unterscheiden. Die Referenz ist ein Wert und damit ein unveränderliches

Objekt. Die Speicherzelle ist kein Wert, sondern ein veränderliches Teilobjekt des Interpreters. Der Verbindung zwischen der Referenz und der durch sie bezeichneten Speicherzelle wird durch den Interpreter hergestellt.

Jede Allokation liefert eine neue Referenz. Der Vergleich

```
ref 0 = ref 0
```

liefert daher den Wert *false*, da der linke und der rechte Unterausdruck verschiedene Referenzen liefern. Der Vergleich

```
!(ref 0) = !(ref 0)
```

liefert dagegen den Wert *true*, da diesmal die Werte in den durch die Referenzen identifizierten Zellen verglichen werden.

Syntaktisch gesehen handelt es sich bei dem Wort := um einen Operator, dem alle anderen Operatoren untergeordnet sind. Das sorgt dafür, dass $f\,x := y + 4$ wie $(f\,x) := (y + 4)$ geklammert wird. Die Wörter ! und *ref* werden wie Bezeichner behandelt. Entsprechend muss man auf das Setzen von Klammern achten. Beispielsweise kann bei $!(!(\mathit{ref}\ (\mathit{ref}\ 1)))$ kein Klammerpaar weggelassen werden. Hier ist ein weiteres Beispiel:

```
map ! (map ref [1, 2, 3])
```
[1, 2, 3] : int list

Hier sind einige gebräuchliche Sprechweisen für Referenzen:

- Unter dem **Wert einer Referenz** verstehen wir den Wert in der durch die Referenz identifizierten Zelle.
- Wir sagen, dass wir eine Referenz auf einen Wert **setzen**, wenn wir den Wert mithilfe der Zuweisungsoperation in die durch die Referenz identifizierte Zelle legen. Alternativ sagen wir, dass wir der Referenz den Wert **zuweisen**.
- Dereferenzierung und Zuweisung bezeichnen wir auch als **Lesen** und **Schreiben**.

Wir unterscheiden zwischen funktionalen und imperativen Werten. Ein **funktionaler** Wert beinhaltet keine Referenzen, während ein **imperativer** Wert Referenzen beinhaltet. Referenzen sind besonders einfache imperative Werte. Ein Paar aus zwei Referenzen ist ein Beispiel für einen zusammengesetzten imperativen Wert.

Funktionale Werte stellen unveränderliche Objekte dar, und imperative Werte stellen veränderliche Objekte dar. Genauer gesprochen können wir sagen, dass ein imperativer Wert zusammen mit einem Speicherzustand den Zustand eines veränderlichen Objekts darstellt.

Wir werden unveränderliche Objekte auch als **funktionale Objekte** und veränderliche Objekte als **imperative Objekte** bezeichnen.

Aufgabe 15.1 Sind alle Werte des Typs *int ref list* imperativ?

Aufgabe 15.2 Welchen Wert liefert der Ausdruck $\mathit{ref}\,(3+2) = \mathit{ref}\,(7-2)$?

Ambige Deklarationen

Wir können jetzt eine Frage beantworten, die wir bisher offen lassen mussten. In § 3.5.2 haben wir gelernt, dass Standard ML zwischen polymorphen und ambigen Deklarationen unterscheidet. Der Grund für diese Unterscheidung liegt in den Speicheroperationen. Betrachten Sie dazu den folgenden Ausdruck:

```
let
    val r = ref (fn x => x)
in
    r:= (fn () => ()) ;
    1 + (!r 4)
end
```

Dieser Ausdruck ist aus den folgenden Gründen unzulässig:

1. Die Deklaration von *r* ist ambig, da ihre rechte Seite eine Applikation ist. Also muss *r* mit einem Typen $(t \to t)\ ref$ typisiert werden.
2. Das erste benutzende Auftreten von *r* verlangt den Typ $(unit \to unit)\ ref$.
3. Das zweite benutzende Auftreten von *r* verlangt einen Typ $(int \to int)\ ref$.

Nehmen Sie jetzt probehalber an, dass der Bezeichner *r* im obigen Let-Ausdruck polymorph mit dem Typschema $\forall\alpha.\ (\alpha \to \alpha)\ ref$ getypt wird. Dann ist der Ausdruck gemäß der statischen Semantik zulässig. Bei seiner Ausführung kommt es jedoch bei der Ausführung des Teilausdrucks *!r 4* zu einem Typfehler, da eine Prozedur des Typs $unit \to unit$ auf ein Argument des Typs *int* angewendet wird.

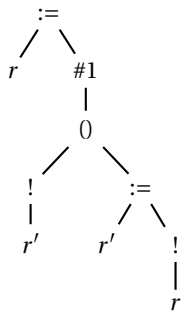
15.2 Speichereffekte

Wenn die Ausführung einer Phrase den Zustand des Speichers ändert, sagen wir, dass die Ausführung der Phrase einen **Speichereffekt** bewirkt. Zu einer Änderung des Speicherzustands kommt es, wenn durch die Allokationsoperation eine bisher nicht allozierte Zelle alloziert oder durch die Zuweisungsoperation eine bereits allozierte Zelle mit einem anderen Wert belegt wird.

Prozeduren, deren Zweck darin besteht, einen Speichereffekt zu bewirken, haben oft den Ergebnistyp *unit*. Ein typisches Beispiel ist die Prozedur *swap*, die die Werte zweier Referenzen vertauscht:

```
fun swap r r' = r:= #1(!r', r':= !r)
val swap : α ref→ α ref→ unit
```

Es ist wichtig, die Funktionsweise dieser auf den ersten Blick undurchsichtigen Prozedur im Detail zu verstehen. Dafür ist es hilfreich, den Rumpf der Prozedur als Baum darzustellen:



Die wichtigsten Schritte bei der Ausführung des Prozedurrumpfs sind:

1. Beschaffe den Wert der Referenz r'.
2. Beschaffe den Wert der Referenz r.
3. Weise r' den Wert aus (2) zu.
4. Weise r den Wert aus (1) zu.

Machen Sie sich klar, dass der Prozedurrumpf diese Ausführungsschritte genau in der angegebenen Reihenfolge vorgibt und dass diese Schritte in der Tat die Werte der Referenzen r und r' vertauschen. Hier ist ein Beispiel:

```
let val (r,r') = (ref 2, ref 3) in swap r r' ; (!r, !r') end
```
*(3,2) : int * int*

Die vordeklarierte Prozedur *app* : $(\alpha \to unit) \to \alpha\ list \to unit$ dient dazu, eine Prozedur p zwecks Erzielung von Speichereffekten auf jedes Element einer Liste anzuwenden (von links nach rechts):

```
fun app p nil = ()
  | app p (x::xr) = (p x : unit ; app p xr)
```
val app : (α → unit) → α list → unit

```
val xs = map ref [2, 3, 6]
```
val xs = [ref, ref, ref] : int ref list

```
map ! xs
```
[2, 3, 6] : int list

```
app (fn r => r:= !r+7) xs
```
() : unit

```
map ! xs
```
[9, 10, 13] : int list

Aufgabe 15.3 Die zweite Regel der oben deklarierten Prozedur *app* ist mit einer Typangabe versehen. Wozu dient diese Typangabe?

15.3 Imperative Prozeduren

Mit Referenzen sind wir in der Lage, eine Prozedur *counter* : *unit* → *int* zu schreiben, die in einer Speicherzelle mitzählt, wie oft sie aufgerufen wird und dementsprechend beim n-ten Aufruf die Zahl n liefert:

```
val r = ref 0
```
val r : int ref

```
fun counter () = (r:= !r+1 ; !r)
```
val counter : unit → int

```
counter()
```
1 : int

```
(counter(),  counter(), counter())
```
*(2, 3, 4) : int * int * int*

```
counter() + counter()
```
11 : int

Jeder Aufruf der Prozedur *counter* erhöht den Wert der Referenz r um eins. Als Ergebnis liefert die Prozedur jeweils den erhöhten Wert. Da die Referenz initial den Wert 0 hat, liefert der erste Aufruf der Prozedur den Wert 1.

Bei der Prozedur *counter* handelt es sich um einen imperativen Wert, der die Referenz r beinhaltet. Die Prozedur *counter* stellt also ein veränderliches Objekt dar. Wir sagen, dass *counter* eine **imperative Prozedur** ist.

Die vorliegende Realisierung der Prozedur *counter* ist unschön, da die von *counter* benutzte Zelle von außen über den Bezeichner r zugänglich ist. Mit einem Let-Ausdruck und einer Abstraktion können wir erreichen, dass nur die Prozedur *counter* Zugriff auf die Zelle hat:

```
val counter =
    let
        val r = ref 0
    in
        fn () => (r:= !r+1 ; !r)
    end
```

Wir sagen, dass diese Deklaration die Zelle **einkapselt**. Beachten Sie, dass der die Zelle allozierende Let-Ausdruck nur einmal ausgeführt wird, und zwar bei der Ausführung der Deklaration für *counter* (siehe § 2.7.3 und § 2.7.1 (9)). Der Let-Ausdruck liefert die Prozedur, an die der Bezeichner *counter* gebunden wird. Die Tripeldarstellung (§ 2.5, § 3.2) dieser Prozedur ist

$$(fn\,() \Rightarrow (r := !r + 1\,;\, !r),\ \ unit \to int,\ \ [r := \rho])$$

wobei ρ (sprich rho) die Referenz der durch den Let-Ausdruck allozierten Zelle bezeichnet. Die Einkapselung der Zelle ist dadurch gegeben, dass ihre Referenz ρ ausschließlich über die Umgebung der Prozedur *counter* zugänglich ist.

Hier ist eine Prozedur *newCounter* : $int \to unit \to int$, die bei jedem Aufruf eine neue Zählprozedur mit einer neuen eingekapselten Zelle erzeugt:

```
fun newCounter i =
    let
        val r = ref i
    in
        fn () => (r:= !r+1 ; !r)
    end
```

Der Anfangswert der zu erzeugenden Zählprozedur kann beliebig vorgegeben werden. Wir bekommen die folgenden Interaktionen:

```
val c = newCounter 0
```
val c : unit → int

```
c()
```
1 : int

```
c() + c()
```
5 : int

```
val c' = newCounter ~3
```
val c' : unit → int

```
c'()
```
~2 : int

```
c'() + c()
```
3 : int

Aufgabe 15.4 (Generatoren) Ein Generator für eine Folge $x_1, x_2, \ldots$ von Werten eines Typs t ist eine Prozedur $unit \to t$, die beim n-ten Aufruf das Folgenglied x_n liefert.

a) Schreiben Sie einen Generator *square* für die Folge $1, 4, 9, \ldots$ der Quadratzahlen.

b) Schreiben Sie eine Prozedur *newSquare* : $unit \to unit \to int$, die bei jedem Aufruf einen neuen Generator für die Folge der Quadratzahlen liefert.

c) Schreiben Sie eine Prozedur *newGenerator* : $(int \to \alpha) \to unit \to \alpha$, die zu einer Prozedur f einen Generator für die Folge $f\,1, f\,2, f\,3, \ldots$ liefert.

d) Schreiben Sie mithilfe der Prozedur *newGenerator* einen Generator *cube* für die Folge $1, 8, 27, \ldots$ der Kubikzahlen.

e) Schreiben Sie mithilfe der Prozedur *newGenerator* eine Prozedur *newCube*, die bei jedem Aufruf einen neuen Generator für die Folge der Kubikzahlen liefert.

Aufgabe 15.5 (Generator für Fakultäten) Schreiben Sie einen Generator *fac* für die Folge 1!, 2!, 3!, ... der Fakultäten. Verwenden Sie keine Rekursion. Die Laufzeit für einen Aufruf von *fac* soll konstant sein.

Aufgabe 15.6 (Zähler mit Rücksetzen) Vervollständigen Sie die Deklaration

```
val (count, inc, reset) =
```

so, dass sie einen eingekapselten Zähler mit dem Anfangswert 0 und drei Prozeduren wie folgt liefert:

- *count* : *unit* → *int* liefert den Wert des Zählers.
- *inc* : *unit* → *unit* erhöht den Wert des Zählers um 1.
- *reset* : *unit* → *unit* setzt den Zähler auf 0 zurück.

15.4 Reihungen

Unter einer **imperativen Datenstruktur** verstehen wir eine Datenstruktur, deren Objekte veränderlich sind. Datenstrukturen, deren Objekte unveränderlich sind, bezeichnen wir dagegen als **funktional**.[1]

Die grundlegende imperative Datenstruktur in Standard ML ist die mit Referenzen realisierte Datenstruktur der Speicherzellen. Mit Speicherzellen können wir weitere imperative Datenstrukturen realisieren. Zunächst betrachten wir die Datenstruktur der Reihungen.

Reihungen gehören zur Standardausstattung von Programmiersprachen und spielen bei vielen effizienten Algorithmen eine Rolle. Eine **Reihung** können wir uns als einen Vektor vorstellen, dessen Komponenten mittels einer Zuweisungsoperation verändert werden können. Diese Idee können wir einfach dadurch realisieren, dass wir Reihungen als Vektoren von Referenzen implementieren. Abbildung 15.1 zeigt eine entsprechende Implementierung von Reihungen. Die Operation *array* liefert zu einer Zahl n und einem Wert x eine neue Reihung der Länge n, die an allen Positionen den Wert x trägt. Die Operationen *fromList*, *sub*, *length*, *foldl*, *foldr* entsprechen den gleichnamigen Operationen für Vektoren. Mit der Operation *app*, die es auch für Vektoren und Listen gibt, kann eine Prozedur von links nach rechts auf jede Komponente einer Reihung angewendet werden, um Speichereffekte zu erzielen. Mit den Operationen *update* und *modify* können Reihungen verändert werden. Die Operation *update* weist einer Position einen Wert zu. Die Operation *modify* wendet eine Prozedur auf alle Komponenten einer Reihung an und kann als eine imperative Version der Operation *map* für Vektoren aufgefasst werden.

Standard ML stellt Reihungen über eine Standardstruktur *Array* zur Verfügung, die eine Obermenge der in Abbildung 15.1 gezeigten Operationen implementiert. Hier sind

[1] Statt von funktionalen und imperativen Datenstrukturen spricht man auch von *persistenten* und *ephemeren* Datenstrukturen.

```
signature ARRAY = sig
    eqtype 'a array
    val array    : int * 'a -> 'a array
    val fromList : 'a list -> 'a array
    val sub      : 'a array * int -> 'a  (* Subscript *)
    val length   : 'a array -> int
    val foldl    : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val foldr    : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val app      : ('a -> unit) -> 'a array -> unit
    val update   : 'a array * int * 'a -> unit  (* Subscript *)
    val modify   : ('a -> 'a) -> 'a array -> unit
end

structure Array :> ARRAY = struct
    type 'a array = 'a ref vector
    fun array (n,x) = Vector.tabulate(n, fn _ => ref x)
    fun fromList xs = Vector.fromList (map ref xs)
    fun sub (v,i) = !(Vector.sub(v,i))
    fun length v = Vector.length v
    fun foldl f s v = Vector.foldl (fn (x,a) => f(!x,a)) s v
    fun foldr f s v = Vector.foldr (fn (x,a) => f(!x,a)) s v
    fun app p v = Vector.app (fn x => p(!x)) v
    fun update (v,i,x) = Vector.sub(v,i) := x
    fun modify f a = iterup 0 (length a - 1) ()
        (fn (i,_) => update(a, i, f(sub(a,i))))
end
```

Abbildung 15.1: *Reihungen*

einige Interaktionen, die die Funktionsweise der Reihungsoperationen verdeutlichen:

```
val a = Array.fromList [1, 2, 3, 4]
```
val a : int array

```
fun array2list a = Array.foldr (op::) nil a
```
val array2list: α array → α list

```
array2list a
```
[1, 2, 3, 4] : int list

```
(Array.modify (fn x => x*x) a ; array2list a)
```
[1, 4, 9, 16] : int list

```
(Array.modify (fn x => x*x) a ; array2list a)
```
[1, 16, 81, 256] : int list

```
fun interval xs s g = let
    val log = Array.array(g-s+1, false)
    fun test x = if s<=x andalso x<=g
                 then Array.update(log, x-s, true)
                 else ()
in
    app test xs ;
    Array.foldl (fn (b,b') => b andalso b') true log
end
```

val interval : int list → int → int → bool

Abbildung 15.2: *Intervalltest mit Logbuch*

Reihungen sind insbesondere für solche Anwendungen interessant, die von der konstanten Laufzeit der Operationen *sub* und *update* profitieren können. Als Beispiel betrachten wir eine Prozedur *interval*, die testet, ob eine Liste alle Zahlen enthält, die zwischen zwei Zahlen $s \leq g$ liegen (ein sogenannter **Intervalltest**). Wir realisieren die Prozedur *interval* mit einer als **Logbuch** bezeichneten Reihung, die für jedes Element des Intervalls s bis g eine Position hat. Zu Beginn setzen wir alle Positionen des Logbuchs auf *false*. Dann testen wir für jedes Element der Liste, ob es im Intervall liegt. Wenn ein Element im Intervall liegt, setzen wir die ihm entsprechende Position des Logbuchs auf *true*. Nachdem wir alle Elemente der Liste getestet haben, sagt das Logbuch für jedes Element des Intervalls, ob es in der Liste vorkommt oder nicht. Abbildung 15.2 zeigt die Realisierung der beschriebenen Prozedur.

Der beschriebene Intervalltest ist vor allem für lange Listen und kurze Intervalle interessant, da seine Laufzeit dann linear in der Länge der Liste ist.

Aufgabe 15.7 Schreiben Sie eine Prozedur $sum : int\ array \to int$, die zu einer Reihung die Summe ihrer Komponenten liefert. Verwenden Sie *Array.foldl*.

Aufgabe 15.8 Schreiben Sie eine Prozedur $copy : \alpha\ array \to \alpha\ array$, die zu einer Reihung eine Kopie liefert (d.h. eine neue Reihung, die dieselbe Komponentenfolge wie die gegebene Reihung hat).

Aufgabe 15.9 Der beschriebene Intervalltest kann weiter verbessert werden, indem man in einer Zelle mitzählt, wie viele Positionen des Logbuchs noch nicht auf *true* gesetzt wurden. Sobald alle Positionen des Logbuchs auf *true* gesetzt sind, kann der Test beendet werden. Schreiben Sie eine Prozedur *interval'*, die einen entsprechend optimierten Intervalltest implementiert.

Aufgabe 15.10 (Histogramme) Ein Histogramm ist eine grafische Darstellung von Häufigkeiten in Form von Säulen. Schreiben Sie eine Prozedur $histo : int\ list \to int \to int\ list$, die zu xs und n eine Liste der Länge $n+1$ liefert, deren i-te Position angibt, wie oft die Zahl i in der Liste xs vorkommt. Beispielsweise soll $histo\ [2,0,3,2,2,0]\ 2 = [2,0,3]$ gelten.

```
fun reverse a = let
    fun swap i j =
        Array.update(a, i, #1(Array.sub(a,j),
                              Array.update(a, j, Array.sub(a,i))))
    fun reverse' l r =
        if l>=r then ()
        else (swap l r; reverse' (l+1) (r-1))
in
    reverse' 0 (Array.length a - 1)
end
```

val reverse : α array → unit

Abbildung 15.3: *Reversieren von Reihungen*

15.5 Reversieren und Sortieren von Reihungen

Um die Komponentenfolge einer Reihung zu reversieren oder zu sortieren, ist es nicht notwendig, eine neue Reihung zu allozieren. Stattdessen kann die bestehende Reihung durch Umordnung ihrer Komponenten so geändert werden, dass ihre Komponentenfolge der Reversion oder der Sortierung der ursprünglichen Komponentenfolge entspricht.

Um eine Reihung der Länge n zu reversieren, können wir wie folgt vorgehen: Zuerst vertauschen wir die Komponenten an den Positionen 0 und $n-1$, dann die Komponenten an den Positionen 1 und $n-2$. Dieses Vertauschen von Komponenten setzen wir so lange fort, bis sich die linke und die rechte Position treffen. Hier ist ein Beispiel, das die jeweilige Komponentenfolge der Reihung durch eine Liste darstellt:

	[1, 2, 3, 4, 5, 6, 7]	Vertausche Positionen 0 und 6
→	[7, 2, 3, 4, 5, 6, 1]	Vertausche Positionen 1 und 5
→	[7, 6, 3, 4, 5, 2, 1]	Vertausche Positionen 2 und 4
→	[7, 6, 5, 4, 3, 2, 1]	

Abbildung 15.3 zeigt eine Prozedur *reverse*, die den geschilderten Algorithmus realisiert. Die Hilfsprozedur *swap* vertauscht die Komponenten an zwei gegebenen Positionen der zu reversierenden Reihung. Sie entspricht der in § 15.2 besprochenen Vertauschungsprozedur für Referenzen.

Aufgabe 15.11 Schreiben Sie eine Prozedur *min*: *int array* → *int*, die zu einer nicht leeren Reihung eine Position liefert, deren Komponente minimale Größe hat. Beispielsweise soll *min* zu einer Reihung mit der Komponentenfolge $[3,2,1,2,1,4]$ eine der Positionen 2 oder 4 liefern.

Aufgabe 15.12 (Rotieren) Schreiben Sie eine Prozedur *rotate*: *α array* → *unit*, die die Komponenten einer Reihung um eine Position nach rechts verschiebt und die letzte

Komponente an die Stelle der ersten Komponente rückt. Beispielsweise soll die Komponentenfolge $[1,2,3,4]$ zu $[4,1,2,3]$ geändert werden. Verwenden Sie eine Hilfsprozedur $rotate' : int \to \alpha \to unit$, die für l und x alle Positionen ab l um eins nach rechts verschiebt und die Position l auf x setzt. Die letzte Komponente soll dabei verloren gehen.

Aufgabe 15.13 (Sortieren durch Auswählen) Ein einfacher, als Sortieren durch Auswählen bezeichneter Algorithmus geht beim Sortieren einer durch ihre linkeste und rechteste Position gegebenen nicht leeren Teilreihung (l,r) wie folgt vor:

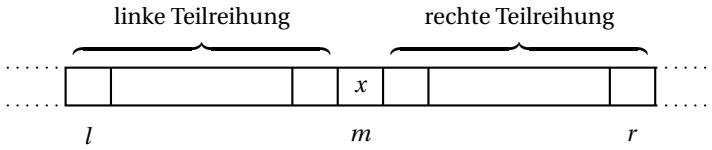
a) Bestimme die Position m einer Komponente minimaler Größe.

b) Vertausche die Komponenten an den Positionen l und m.

c) Sortiere die Teilreihung $(l+1,r)$.

Schreiben Sie eine Prozedur $ssort : int\ array \to unit$, die Reihungen durch Auswählen sortiert. Bestimmen Sie die Laufzeit Ihrer Prozedur.

Aufgabe 15.14 (Quicksort) Der in der Praxis am häufigsten verwendete Algorithmus zum Sortieren von Reihungen heißt Quicksort. Die Idee hinter Quicksort haben wir bereits in Aufgabe 5.13 auf S. 105 kennengelernt, allerdings in vereinfachter Form für Listen. Bei Reihungen ergibt sich der wichtige Vorteil, dass die Teillisten durch Komponentenumordnung innerhalb der Reihung dargestellt werden können (man sagt **in place** oder **in situ**). Das Herzstück von Quicksort ist eine Prozedur *part*, die eine Teilreihung (l,r) für eine gegebene Komponente x von (l,r) so umordnet, dass sie eine Position m zwischen l und r wie folgt liefert:



- Die Komponente an der Position m ist x.
- Alle Komponenten der linken Teilreihung $(l,\ m-1)$ sind kleiner gleich x.
- Alle Komponenten der rechten Teilreihung $(m+1,\ r)$ sind größer gleich x.

Beispielsweise kann *part* eine Teilreihung $[4,3,6,1,5]$ für 4 zu $[1,3,4,6,5]$ umordnen und die Position $l+2$ liefern.

a) Schreiben Sie eine endrekursive Prozedur $part : int \to int \to int \to int$, die zu x, l und r eine Position m gemäß der obigen Spezifikation liefert. Nehmen Sie dabei an, dass die zugrunde liegende Reihung über den Bezeichner a verfügbar ist.

b) Schreiben Sie eine binär-rekursive Prozedur $quick' : int \to int \to unit$, die eine gegebene Teilreihung (l,r) einer Reihung a mithilfe der Prozedur *part* sortiert.

c) Schreiben Sie eine Prozedur $quick : int\ array \to unit$, die eine Reihung gemäß dem Quicksort-Algorithmus sortiert.

Die Laufzeit von Quicksort ist quadratisch in der Länge der Reihung, wobei der Worst-Case-Fall in der Praxis nur sehr selten auftritt. Für Listen ist Quicksort in der Praxis uninteressant, da sich keine In-Place-Vorteile ergeben und Sortieren durch Mischen generell schneller ist.

15.6 Agenden

Unter einer **Agenda** wollen wir ein veränderliches Objekt verstehen, in das Werte eingetragen werden können, und aus dem sie auch wieder gelöscht werden können. Die Einträge einer Agenda sollen wie die Positionen einer Liste linear angeordnet sein, sodass wir vom ersten und vom letzten Eintrag sprechen können. Für Agenden soll es die folgenden Operationen geben:

```
eqtype α agenda
val agenda : unit → α agenda
val insert : α agenda → α → unit
val remove : α agenda → unit      (* Empty *)
val first  : α agenda → α         (* Empty *)
val empty  : α agenda → bool
```

- *agenda* liefert eine neue Agenda, die zunächst leer ist.
- *insert* trägt einen Wert in eine Agenda ein.
- *remove* nimmt den ersten Eintrag aus einer Agenda.
- *first* liefert den Wert des ersten Eintrags einer Agenda.
- *empty* testet, ob eine Agenda leer ist.

Die Operationen *remove* und *first* werfen die Ausnahme *Empty*, wenn sie auf eine leere Agenda angewendet werden.

Agenden, bei denen der erste Eintrag immer der am längsten in der Agenda stehende Eintrag ist, nennen wir (imperative) **Schlangen**. Agenden, bei denen der erste Eintrag immer der am kürzesten in der Agenda stehende Eintrag ist, nennen wir **Stapel**. Man sagt prägnant, dass Schlangen eine **first-in, first-out Strategie** (FIFO) realisieren, und Stapel eine **last-in, first-out Stategie** (LIFO).

Einen Stapel können wir uns bildhaft als einen Kartenstapel vorstellen. Die Operation *insert* legt eine neue Karte auf den Stapel, die Operation *remove* nimmt die oberste Karte vom Stapel. Die Operation *first* liefert den auf der obersten Karte des Stapels vermerkten Wert. Dieser bildhaften Vorstellung entsprechend werden die Operationen *insert*, *remove* und *first* bei Stapeln als *push*, *pop* und *top* bezeichnet.

Auch Schlangen können wir uns als Kartenstapel vorstellen, allerdings mit der wichtigen Änderung, dass *insert* eine neue Karte unter den Stapel schiebt.

Die einfachste Art, Agenden zu implementieren, besteht darin, sie durch Zellen darzustellen, die die Liste der Einträge enthalten. Die erste Position der Liste entspricht dabei

dem ersten Eintrag der Agenda. Bei Schlangen fügt *insert* neue Werte am Ende der Liste ein, bei Stapeln am Anfang der Liste. Für Stapel ist diese Implementierung effizient, da alle Operationen konstante Laufzeit haben, für Schlangen ist sie weniger effizient, da *insert* lineare Laufzeit hat.

Aufgabe 15.15 Implementieren Sie eine Datenstruktur für Stapel mit der folgenden Signatur:

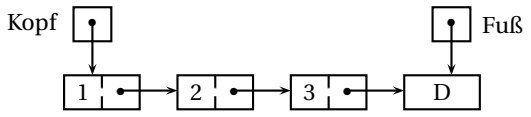
```
eqtype α stack
val stack : unit → α stack
val push  : α stack → α → unit
val pop   : α stack → unit      (* Empty *)
val top   : α stack → α         (* Empty *)
val empty : α stack → bool
```

Achten Sie darauf, dass alle Operation konstante Laufzeit haben.

Aufgabe 15.16 Lösen Sie Aufgabe 14.7 auf S. 286 so, dass die Operation *find* nur einmal aufgerufen werden muss. Hilfe: Verwenden Sie eine Testprozedur, die die vorgelegten Werte in einen Stapel ablegt und stets den Wert *false* liefert.

15.7 Effiziente imperative Schlangen

Wir geben jetzt eine Implementierung für imperative Schlangen an, bei der alle Operationen konstante Laufzeit haben (auch einzeln betrachtet). Dabei stellen wir Schlangen durch **verzeigerte Zellen** dar. Für eine Schlange mit den Einträgen 1, 2, 3 ergibt sich die folgende **Kastendarstellung**:



Jeder Kasten stellt eine Zelle dar, und jeder Pfeil eine Referenz. Ein Pfeil stellt dabei die Referenz der Zelle dar, auf die er zeigt. Die Pfeile werden als **Zeiger** bezeichnet. Für jeden Eintrag der Schlange gibt es eine Zelle. Zusätzlich gibt es eine Zelle für den sogenannten **Dummy-Eintrag**, die den Wert D trägt. Die Einträge der Schlange sind gemäß ihrer Ordnung verkettet, wobei der Dummy-Eintrag als letztes Glied der Kette erscheint. Der Dummy-Eintrag existiert auch dann, wenn die Schlange leer ist. Die Zellen für die Einträge enthalten entweder den Wert D (wenn es sich um den Dummy-Eintrag handelt) oder einen Wert $E(x,r)$, wobei x der Wert des Eintrags und r die Referenz der Nachfolgerzelle ist:[2]

[2] Das Typsynonym *α entry* wird wegen der verschränkten Rekursion mit dem Konstruktortyp *α state* mit dem Schlüsselwort *withtype* deklariert.

```
datatype 'a state = D | E of 'a * 'a entry
withtype 'a entry = 'a state ref
```

Neben den Zellen für die Einträge gibt es noch zwei Zellen, die als **Kopf** und **Fuß** der Schlange bezeichnet werden. Der Kopf zeigt stets auf den ersten Eintrag der Schlange, und der Fuß stets auf den letzten Eintrag. Wenn die Schlange leer ist, zeigen Kopf und Fuß auf dieselbe Zelle, die dann als Dummy-Eintrag fungiert und den Wert *D* trägt.

Die Operation *remove* setzt den Zeiger im Kopf der Schlange auf den zweiten Eintrag der Schlange. Die Operation *insert* macht den bisherigen Dummy-Eintrag zu einem regulären Eintrag, der auf einen neuen Dummy-Eintrag zeigt. Der Fuß der Schlange ermöglicht den schnellen Zugriff auf den bisherigen Dummy-Eintrag. Sein Zeiger wird durch die Operation *insert* auf den neuen Dummy-Eintrag gesetzt.

Abbildung 15.4 zeigt eine auf den obigen Ausführungen beruhende Implementierung imperativer Schlangen. Eine Schlange wird dabei durch das Paar dargestellt, das aus den Referenzen der Kopf- und der Fußzelle besteht.

Aufgabe 15.17 Zeichnen Sie die Kastendarstellungen der Zustände, die die Schlange *q* gemäß den folgenden Deklarationen durchläuft.

```
open Queue
val q = queue ()
val _ = (insert q 7 ; insert q 13 ; remove q)
```

Aufgabe 15.18 Deklarieren Sie eine Struktur für imperative Schlangen, die neben *insert* eine Operation *insertFirst* hat, mit der ein Wert am Anfang einer Schlange eingetragen werden kann.

15.8 Schleifen

Bei einer Schleife handelt es sich um ein programmiersprachliches Konstrukt, mit dem iterative Berechnungen beschrieben werden können, die den Speicherzustand verändern. In Standard ML ist eine Schleife ein Ausdruck der Form *while* e_1 *do* e_2, der mit zwei Ausdrücken e_1 und e_2 gebildet ist, die als **Bedingung** und **Rumpf** der Schleife bezeichnet werden. Die Semantik einer Schleife kann durch die Gleichung

$$\textit{while}\ e_1\ \textit{do}\ e_2\ =\ \textit{if}\ e_1\ \textit{then}\ (e_2\ ;\ \textit{while}\ e_1\ \textit{do}\ e_2)\ \textit{else}\ ()$$

beschrieben werden. Bei der Ausführung einer Schleife wird also zunächst die Bedingung ausgewertet. Wenn die Bedingung zu *false* auswertet, ist die Ausführung der Schleife beendet und liefert den Wert (). Wenn die Bedingung dagegen zu *true* auswertet, wird der Rumpf der Schleife ausgeführt. Danach wird die Schleife erneut ausgeführt. Anschaulich gesprochen können wir sagen, dass das Ziel einer Schleife darin besteht, durch wiederholte Ausführung des Rumpfes den Speicherzustand so zu ändern, dass die Bedingung zu *false* auswertet.

```
signature QUEUE = sig
    eqtype 'a queue
    val queue  : unit -> 'a queue
    val insert : 'a queue -> 'a -> unit
    val remove : 'a queue -> unit  (* Empty *)
    val first  : 'a queue -> 'a  (* Empty *)
    val empty  : 'a queue -> bool
end

structure Queue :> QUEUE = struct
    datatype 'a state = D | E of 'a * 'a entry
    withtype 'a entry = 'a state ref
    type 'a queue = 'a entry ref * 'a entry ref
    fun queue () = let
        val dummy = ref D
    in
        (ref dummy, ref dummy)
    end
    fun insert (_,f) x = let
        val dummy = ref D
    in
        !f:= E(x,dummy) ;
        f:= dummy
    end
    fun remove (h,_) = case !(!h) of
        D => raise Empty
      | E(_,n) => h:= n
    fun first (h,_) = case !(!h) of D => raise Empty | E(x,_) => x
    fun empty (h,f) = !h = !f
end
```

Abbildung 15.4: *Effiziente imperative Schlangen*

```
fun poweri x n = let
    val a = ref 1
    val i = ref 1
in
    while !i <= n do (
        a:= !a*x ;
        i:= !i+1
    );
    !a
end
```

```
fun power x n = let
    fun power' a i =
        if i<=n
        then power' (a*x) (i+1)
        else a
in
    power' 1 1
end
```

Abbildung 15.5: *Potenzberechnung mit Schleife (links) und mit Endrekursion (rechts)*

Abbildung 15.5 zeigt links eine Prozedur *poweri*, die mithilfe einer Schleife zu x und n die Potenz x^n bestimmt. Die Prozedur alloziert zunächst zwei Zellen a und i, die wir als **Akku** und **Zähler** bezeichnen wollen. Der Zweck der Schleife besteht darin, den Akku, der initial den Wert 1 hat, n-mal mit x zu multiplizieren, sodass er schließlich den Wert x^n hat. Die Schleife erhöht den Wert des Zählers bei jedem Durchlauf um eins. Da der Zähler initial den Wert 1 hat und die Schleife terminiert, sobald der Zähler größer als n ist, wird die Schleife genau n-mal durchlaufen (für $i = 1, \ldots, n$).

In Standard ML sind Schleifen eine abgeleitete Form, die sich auf endrekursive Prozeduren zurückführen lässt:

$$\text{while } e_1 \text{ do } e_2 \quad \rightsquigarrow \quad \begin{array}{l}\text{let}\\ \quad \text{fun } \mathit{loop}\,() = \text{if } e_1 \text{ then } (e_2 \,;\, \mathit{loop}\,()) \text{ else } ()\\ \text{in}\\ \quad \text{loop } ()\\ \text{end}\end{array}$$

Umgekehrt gilt, dass sich die Anwendung einer endrekursiven Prozedur auch durch eine Schleife realisieren lässt. Betrachten Sie dazu die in Abbildung 15.5 rechts angegebene Prozedur *power*, die Potenzen mithilfe einer endrekursiven Hilfsprozedur *power′* bestimmt. Machen Sie sich klar, dass die Anwendung *power′ 1 1* mechanisch in den Rumpf der Prozedur *poweri* überführt werden kann, die Potenzen mit einer Schleife berechnet.

Das Programmieren mit Schleifen erfordert Übung. Wegen der Entsprechung von Schleifen und Endrekursion bietet es sich an, eine Berechnungsaufgabe zunächst mit einer endrekursiven Prozedur zu lösen und danach die endrekursive Lösung in eine Schleife zu übersetzen. Als Beispiel für dieses Vorgehen können die Prozeduren *power* (endrekursive Lösung) und *poweri* (Lösung mit Schleife) in Abbildung 15.5 dienen.

In vielen gebräuchlichen Programmiersprachen spielen Schleifen, anders als in Standard ML, eine wichtige Rolle. Das liegt vor allem daran, dass in diesen Sprachen Speicheroperationen eine grundlegende Rolle spielen. In § 16.4 werden wir dazu mehr sagen.

Aufgabe 15.19 Schreiben Sie eine Prozedur $\mathit{fac} : \mathit{int} \to \mathit{int}$, die mit einer Schleife zu $n \in \mathbb{N}$ die n-te Fakultät $n!$ bestimmt.

Aufgabe 15.20 Schreiben Sie eine Prozedur $\mathit{gcd} : \mathit{int} \to \mathit{int} \to \mathit{int}$, die mit einer Schleife zu zwei positiven Zahlen den größten gemeinsamen Teiler bestimmt.

Aufgabe 15.21 Schreiben Sie eine Prozedur $\mathit{length} : \alpha\ \mathit{list} \to \mathit{int}$, die mit einer Schleife die Länge einer Liste bestimmt. Verwenden Sie die vordeklarierten Prozeduren *null*, *tl* und *not*.

Aufgabe 15.22 Schreiben Sie eine Prozedur $\mathit{min} : \mathit{int\ array} \to \mathit{int}$, die mit einer Schleife die kleinste Komponente einer nicht leeren Reihung bestimmt.

Aufgabe 15.23 Schreiben Sie eine Prozedur $\mathit{reverse} : \mathit{int\ array} \to \mathit{unit}$, die eine Reihung mithilfe einer Schleife durch Umordnung ihrer Komponenten reversiert.

15.9 Lineare Speicher

Unter einem **linearen Speicher** wollen wir eine Reihung über *int* verstehen. Wie Sie bereits aus § 1.13.1 wissen, enthält *int* nur die Zahlen aus einem endlichen Intervall.

Lineare Speicher sind ein gutes Modell für die von der Hardware eines Computers realisierten Speicher. Wir werden zeigen, wie Tupel, Listen und mit Konstruktoren gebildete Werte in linearen Speichern dargestellt werden können. Die dabei verwendeten Techniken spielen bei der Realisierung von Programmiersystemen eine wichtige Rolle. Außerdem liefern uns lineare Speicher ein Modell, mit dem wir den Speicherplatzbedarf von baumartigen Werten abschätzen können.

Wir werden mit einem als **Halde** bezeichneten linearen Speicher arbeiten, mit dem wir Zellen blockweise allozieren können. Wir implementieren die Halde wie in Abbildung 15.6 gezeigt mit einer Struktur *Heap*. Das Kernstück der Halde ist eine Reihung über *int*. Die Positionen der Reihung bezeichnen wir als **Adressen**. Wir unterscheiden zwischen **freien** und **allozierten** Zellen der Halde. Zunächst sind alle Zellen frei. Auf die Halde kann mit den folgenden Operationen zugegriffen werden:

- *new n* **alloziert** einen **Block** der **Länge** $n \geq 1$ und liefert die **Adresse des Blocks**. Ein Block der Länge n besteht aus n aufeinander folgenden Zellen. Seine Adresse ist die Adresse der ersten Zelle. Die Blöcke werden, beginnend mit der Adresse 0, aufeinander folgend in der Halde alloziert. Die relativen Positionen der Zellen eines Blocks werden als **Indizes** bezeichnet. Ein Block der Länge n hat die Indizes 0 bis $n-1$.
- *sub a i* liefert den Wert in der Zelle mit der Adresse $a+i$. Dabei sollte a die Adresse eines Blocks und i ein Index dieses Blocks sein.
- *update a i x* legt den Wert x in die Zelle mit der Adresse $a+i$. Dabei sollte a die Adresse eines Blocks und i ein Index dieses Blocks sein.
- *release a* dealloziert alle Zellen der Halde, deren Adresse größer gleich a ist. Neue Blöcke werden danach ab der Adresse a alloziert.
- *show* liefert die Werte der allozierten Zellen als Liste. Damit können Sie beim Experimentieren nachsehen, welche Werte sich im allozierten Teil der Halde befinden.

Im Folgenden gehen wir davon aus, dass die Struktur *Heap* mit *open Heap* geöffnet ist. Zunächst deklarieren wir eine Prozedur *new'*, die eine Folge von Zahlen in einem neu allozierten Block ablegt:

```
fun new' xs = let
    val a = new (length xs)
in
    foldl (fn (x,i) => (update a i x ; i+1)) 0 xs ;
    a
end
val new' : int list → address
```

```
signature HEAP = sig
    exception Address
    exception OutOfMemory
    type address = int
    type index = int
    val new     : int -> address  (* Address, OutOfMemory *)
    val sub     : address -> index -> int  (* Address *)
    val update  : address -> index -> int -> unit  (* Address *)
    val release : address -> unit
    val show    : unit -> (address * int) list
end

structure Heap :> HEAP = struct
    val size = 1000
    val array = Array.array(size,~1)
    val lar = ref ~1  (* last allocated address *)
    exception Address
    exception OutOfMemory
    type address = int
    type index = int
    fun new n = if n<1 then raise Address
                else if !lar+n >= size then raise OutOfMemory
                     else #1(!lar+1, lar:= !lar+n)
    fun check a = if a<0 orelse a > !lar then raise Address else a
    fun sub a i = Array.sub(array, check(a+i))
    fun update a i x = Array.update(array, check(a+i), x)
    fun release a = lar:= (if a=0 then a else check a) - 1
    fun show () = iterdn (!lar) 0 nil
                  (fn (a,es) => (a, Array.sub(array,a)) :: es)
end
```

Abbildung 15.6: *Implementierung einer Halde*

```
new' [4, 7, 8]
```
0 : address

```
new' [~1, 2]
```
3 : address

```
show()
```
*[(0, 4), (1, 7), (2, 8), (3, ~1), (4, 2)] : (address * int) list*

Machen Sie sich klar, dass eine Halde mit den Operationen *new* und *release* stapelartig verwaltet werden kann.

Eigentlich wäre es sinnvoll, den Typ *address* abstrakt zu halten. Das scheitert jedoch daran, dass in den Zellen der Blöcke sowohl Zahlen als auch Adressen abgelegt werden sollen.

Aufgabe 15.24 Deklarieren Sie eine Prozedur *counter* : *unit* → *int*, die beim n-ten Aufruf die Zahl n liefert. Die zum Zählen benötigte Zelle soll dabei in der Halde alloziert werden.

15.10 Lineare Darstellung von Listen

Hier ist eine Prozedur *putList*, die eine Liste ganzer Zahlen in der Halde darstellt und eine Zahl liefert, die die Liste darstellt:

```
fun putList nil = ~1
  | putList (x::xr) = new' [x, putList xr]
```
val putList : int list → int

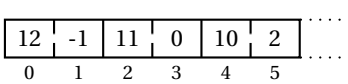
Die leere Liste wird durch die Zahl -1 dargestellt. Eine nicht leere Liste wird durch die Adresse eines Blocks dargestellt, dessen erste Zelle den Kopf und dessen zweite Zelle die Darstellung des Rumpfs der Liste enthält. Für die Darstellung der leeren Liste haben wir eine negative Zahl gewählt, damit klar ist, dass es sich um keine Adresse handelt. Hier ist eine Interaktion, die die Liste $[10,11,12]$ ab der Adresse 0 in der Halde darstellt:

```
(release 0 ; putList [10, 11, 12])
```
4 : int

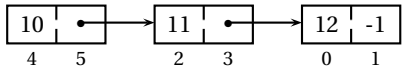
Die für die Darstellung der Liste allozierten Zellen schauen wir uns mit *show* an:

```
show()
```
*[(0, 12), (1, ~1), (2, 11), (3, 0), (4, 10), (5, 2)] : (int * int) list*

Grafisch können wir die allozierten Zellen wie folgt darstellen:



Die Liste [10,11,12] wird also durch drei Blöcke mit den Adressen 0, 2 und 4 dargestellt, wobei der rechteste Block das erste Element der Liste enthält. Wir können die Darstellung der allozierten Zellen wesentlich übersichtlicher gestalten, wenn wir die Blöcke unabhängig voneinander zeichnen und die Adressen in den Zellen durch sogenannte **Zeiger** grafisch darstellen:



Diese Darstellung der allozierten Haldenzellen wird als **verzeigerte Blockdarstellung** bezeichnet. Sie enthält etwas mehr Information als in der Halde steht. Um die Blöcke und Adressen wie gezeigt zu identifizieren, muss nämlich zusätzlich bekannt sein, dass die Adresse 4 die Darstellung einer Liste ist (gemäß der durch *putList* formulierten Vorschrift). Dass diese Zusatzinformation genügt, belegen wir mit einer zu *putList* inversen Prozedur *getList*, die zu einer Darstellung die dargestellte Liste liefert:

```
fun getList a = if a = ~1 then nil
                else sub a 0 :: getList (sub a 1)
```
val getList : int → int list

```
getList 4
```
[10, 11, 12] : int list

Die Haldendarstellung einer Liste kann mit der Operation *update* modifiziert werden. Hier ist eine Prozedur, mit der das n-te Element einer in der Halde dargestellten Liste modifiziert werden kann:

```
fun updateList a n x = if a = ~1 orelse n<0 then raise Subscript
                       else if n=0 then update a 0 x
                            else updateList (sub a 1) (n-1) x
```
val updateList : int → int → int → unit

```
(updateList 4 1 17 ; getList 4)
```
[10, 17, 12] : int list

Aufgabe 15.25 Betrachten Sie den Haldenzustand, der sich durch die Ausführung des folgenden Ausdrucks ergibt:

```
(release 0 ; putList [13,17] ; putList [19,23] ; update 0 1 6)
```

a) Zeichnen Sie die verzeigerte Blockdarstellung der durch die Adresse 2 dargestellten Liste.

b) Welche Liste liefert *getList* für die Adresse 2?

Aufgabe 15.26 Deklarieren Sie Prozeduren, die mit der durch *putList* formulierten Haldendarstellung von Listen arbeiten:

a) Eine Prozedur *null* : *int* → *bool*, die testet, ob eine Liste leer ist.

b) Eine Prozedur *head* : *int* → *int*, die den Kopf einer Liste liefert. Falls die Liste leer ist, soll die Ausnahme *Empty* geworfen werden.

c) Eine Prozedur *tail* : *int* → *int*, die den Rumpf einer Liste liefert. Falls die Liste leer ist, soll die Ausnahme *Empty* geworfen werden.

d) Eine Prozedur *cons* : *int* → *int* → *int*, die zu einer Zahl x und zu einer Darstellung einer Liste xr eine Darstellung der Liste $x :: xr$ liefert.

e) Eine Prozedur *append* : *int* → *int* → *int*, die die Konkatenation zweier Listen liefert.

f) Zeichnen Sie die verzeigerten Blockdarstellungen der durch die Ausführung der folgenden Deklarationen in der Halde dargestellten Listen. Nehmen Sie dabei an, dass die Halde zu Beginn leer ist.

```
val a0 = putList[]
val a1 = putList [1,2,3]
val a2 = append a1 a0
val a3 = append a1 a1
```

Aufgabe 15.27 Deklarieren Sie eine Prozedur *extend* : *int* → *int* → *int*, die zu zwei Listendarstellungen a und b eine Darstellung der Konkatenation der von a und b dargestellten Listen liefert. Dabei sollen keine neuen Zellen alloziert werden. Stattdessen darf die Darstellung von a in der Halde modifiziert werden. Hinweis: Schreiben Sie zuerst eine Prozedur *extend*′, die annimmt, dass die durch a dargestellte Liste nicht leer ist.

Aufgabe 15.28 Überlegen Sie sich, wie Optionen über *int* in der Halde dargestellt werden können. Schreiben Sie entsprechende Prozeduren *putOption* und *getOption*.

15.11 Lineare Darstellung von Bäumen

In § 7.1 haben wir reine Bäume wie folgt dargestellt:

```
datatype tree = T of tree list
```

Bei der Darstellung reiner Bäume in der Halde folgen wir diesem Schema:

```
fun putTree (T ts) = putList (map putTree ts)
```
val putTree : tree → int

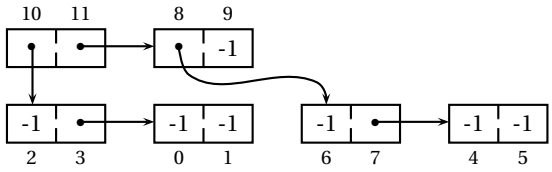
```
fun getTree a = T(map getTree (getList a))
```
val getTree : int → tree

```
val t1 = T[T[], T[]]
val t2 = T[t1, t1]
```

```
(release 0 ; putTree t2)
10 : int

getTree 10
T [T[T[],T[]], T[T[],T[]]] : tree
```

Die Haldendarstellung des Baums *t2* besteht aus 12 Zellen:



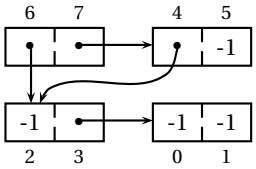
Der Teilbaum *t1* ist dabei zweifach dargestellt, und zwar durch die Adressen 2 und 6. Wenn man *t1* nur einmal darstellt, kann man 4 Zellen einsparen:

```
val _  = release 0
val a1 = putTree t1
val a2 = putList [a1,a1]
val a1 = 2 : int
val a2 = 6 : int
```

Dabei stellt die Adresse *a2* den Baum *t2* dar.

```
getTree a2
T [T[T[],T[]], T[T[],T[]]] : tree
```

Die Haldendarstellung von *t2* sieht diesmal wie folgt aus:



Wir merken uns, dass man bei der Haldendarstellung von Bäumen Platz sparen kann, wenn man mehrfach auftretende Teilbäume nur einmal darstellt. Dieses Vorgehen bezeichnet man als **Strukturzusammenlegung** (engl. structure sharing).

Aufgabe 15.29 Deklarieren Sie eine Prozedur $tree : int \to int$, die für $n \geq 0$ den balancierten Binärbaum der Tiefe n in der Halde darstellt. Dabei sollen mehrfach vorkommende Teilbäume nur einmal in der Halde dargestellt werden, sodass für die Darstellung eines balancierten Binärbaums der Tiefe n genau $4n$ Zellen alloziert werden.

Aufgabe 15.30 Überlegen Sie sich, wie die markierten Bäume aus § 7.9 in der Halde dargestellt werden können. Schreiben Sie entsprechende Prozeduren *putLtr* und *getLtr*.

15.12 Lineare Darstellung von Ausdrücken

Mit Konstruktoren können wir arithmetische Ausdrücke, die mit Konstanten, Variablen und Summen gebildet sind, wie folgt darstellen (siehe § 6.4):

```
type var = int
datatype exp = C of int | V of var | S of exp * exp
```

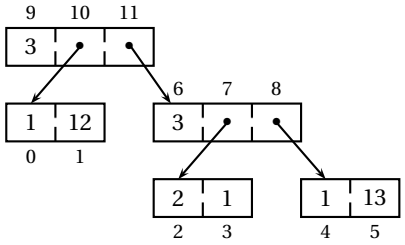
Die folgende Prozedur stellt die mit Konstruktoren dargestellten Ausdrücke in der Halde dar:

```
fun putExp (C n) = new' [1, n]
  | putExp (V x) = new' [2, x]
  | putExp (S(e,e')) = new' [3, putExp e, putExp e']
```
putExp : exp → int

Die verschiedenen Ausdrucksvarianten werden dabei durch Blöcke dargestellt, deren erste Zelle die entsprechende Variantennummer enthält (siehe § 6.1). Hier ist ein Beispiel:

```
(release 0 ; putExp (S(C 12, S(V 1, C 13))))
```
9 : int

Die verzeigerte Blockdarstellung der durch diese Interaktion allozierten Haldenzellen sieht wie folgt aus:



Eine zu *putExp* inverse Prozedur, die zu einer Darstellung den dargestellten Ausdruck liefert, können wir wie folgt deklarieren:

```
fun getExp a = case sub a 0 of
    1 => C(sub a 1)
  | 2 => V(sub a 1)
  | 3 => S(getExp(sub a 1), getExp(sub a 2))
  | _ => raise Address
```
val getExp : int → exp

```
getExp 9
```
S(C 12, S(V 1, C 13)) : exp

Aufgabe 15.31 Deklarieren Sie eine Prozedur $eval : (var \to int) \to int \to int$, die einen in der Halde dargestellten Ausdruck gemäß einer Umgebung evaluiert.

Aufgabe 15.32 Schreiben Sie eine Prozedur $instantiate : (var \to int) \to int \to unit$, die die Variablen eines in der Halde dargestellten Ausdrucks gemäß einer Umgebung in Konstanten umwandelt. Dabei sollen keine neuen Zellen alloziert werden.

15.13 Speicherplatzbedarf bei der Programmausführung

Während der Ausführung eines Programms müssen die für die Ausführung benötigten Werte im Speicher des ausführenden Computers dargestellt werden. Die in diesem Kapitel angegebenen linearen Darstellungen für Tupel, Listen und mit Konstruktoren gebildete Werte geben uns ein Modell in die Hand, mit dem wir den Speicherplatzbedarf bei der Programmausführung abschätzen können.

Als Beispiel betrachten wir die Konkatenation von Listen. Wir nehmen an, dass Listen so wie in § 15.10 beschrieben im Speicher dargestellt werden. Da Listen gemäß den Gleichungen

$$\begin{aligned} nil @ ys &= ys \\ (x :: xr) @ ys &= x :: (xr @ ys) \end{aligned}$$

konkateniert werden, müssen für die Berechnung von $xs @ ys$ so viele Zellen neu alloziert werden, wie für die Darstellung von xs benötigt werden ($2|xs|$ viele).

Es kommt oft vor, dass die Darstellung eines Werts bei der Programmausführung nur für kurze Zeit benötigt wird und danach nicht mehr zugänglich ist. Moderne Programmiersysteme verfügen daher über einen **Speicherbereiniger** (engl. garbage collector), der nicht mehr zugängliche Blöcke automatisch erkennt und die von ihnen belegten Zellen für die Allokation neuer Blöcke freigibt.

Als Beispiel betrachten wir die naive Reversionsprozedur für Listen:

```
fun rev nil = nil
  | rev (x::xr) = rev xr @ [x]
```

Die Anzahl der von *rev* allozierten Zellen hängt nur von der Länge der zu reversierenden Liste ab. Sei sn die Anzahl der Zellen, die *rev* für eine Liste der Länge n alloziert. Dann gilt:

$$\begin{aligned} s0 &= 0 \\ sn &= s(n-1) + 2 + 2(n-1) = s(n-1) + 2n \qquad \text{für } n > 0 \end{aligned}$$

Die zweite Gleichung betrifft nichtleere Listen und ergibt sich wie folgt. Sei $x :: xr$ eine Liste der Länge n. Dann werden zunächst $s(n-1)$ Zellen bei der Reversion von xr alloziert. Weiter werden 2 Zellen für die Darstellung der Liste $[x]$ alloziert. Schließlich werden $2(n-1)$ Zellen für die Konkatenation von *rev xr* und $[x]$ alloziert (die Liste *rev xr* hat die Länge $n-1$).

Aus den Gleichungen für s folgt $sn = 0 + 2 \cdot 1 + \cdots + 2 \cdot n = n(n+1)$. Also werden für die Reversion einer Liste der Länge 10 insgesamt 110 Zellen alloziert. Da für die Darstellung der Ergebnisliste aber nur 20 Zellen erforderlich sind, sind nach der Reversion 90 der allozierten Zellen nicht mehr zugänglich. Diese Zellen können vom Speicherbereiniger freigegeben werden.

Auf den Platzbedarf von Prozeduren werden wir in § 16.10 eingehen. Wir können aber bereits verraten, dass der Platz für die Darstellung eine Prozedur linear in der Anzahl der freien Bezeichner des Codes der Prozedur ist. Den Platz für den Code zählen wir dabei nicht mit, da er nur einmal als Teil des Programms dargestellt wird. Auch den Platz für die Werte der freien Bezeichner zählen wir nicht mit, da deren Darstellung bereits existiert, wenn die Darstellung der Prozedur in den Speicher geschrieben wird.

Aufgabe 15.33 Wieviele Zellen allozieren die folgenden Prozeduren für Listen der Länge n?

a) *length*

b) *map* (*fn* $x \Rightarrow x$)

c) *foldl op* :: *nil*

d) *exists* (*fn* $x \Rightarrow x > 5$)

Aufgabe 15.34 Schreiben Sie eine naive Reversionsprozedur *reverse* : *int* → *int*, die mit der durch *putlist* formulierten Haldendarstellung von Listen arbeitet. Verwenden Sie eine Prozedur *append*, die eine Darstellung der Konkatenation zweier Listen liefert. Zeichnen Sie die verzeigerte Blockdarstellung der durch die folgenden Deklarationen allozierten Haldenzellen:

```
val _ = release 0
val a = putList [3,7]
val b = reverse a
```

Markieren Sie die Zellen, die weder von *a* noch von *b* aus erreichbar sind.

Bemerkungen

In diesem Kapitel haben wir unser Programmiermodell durch die Hinzunahme eines Speichers erweitert. Damit eröffnen sich völlig neue Programmiertechniken. Wir können jetzt veränderliche Objekte darstellen, endrekursive Berechnungen mit Schleifen formulieren, und Listen und Bäume nur mit Werten des Typs *int* darstellen.

Die Hardware eines Computers realisiert ein sehr primitives Programmiermodell, in dem es nur Werte des Typs *int* gibt und lineare Speicher eine wichtige Rolle spielen. Viele gebräuchliche Programmiersprachen orientieren sich an diesem Modell und stellen Speicheroperationen und Schleifen als primäres Ausdrucksmittel zur Verfügung. Solche Sprachen werden als **imperativ** bezeichnet. Dagegen werden Sprachen, in denen so wie in Standard ML Prozeduren und unveränderliche Objekte die Hauptrolle spielen, als **funktional** bezeichnet. Das Paradebeispiel für imperative Sprachen ist die für die Entwicklung des Betriebssystems Unix entworfene Sprache C. Die moderne Form imperativer Sprachen wird als **objektorientiert** bezeichnet und räumt abstrakten Datenstrukturen eine wichtige Rolle ein. Zwei prominente Vertreter der objektorientierten Sprachen sind C++ und Java.

Verzeichnis

Speicher: Zellen und Referenzen; Speicheroperation und Referenztypen; Allokation, Dereferenzierung (Lesen), Zuweisung (Schreiben); Speicherzustand und Speichereffekte.

Sprechweisen für Referenzen: Wert einer Referenz; eine Referenz auf einen Wert setzen, einer Referenz einen Wert zuweisen.

Funktionale und imperative Werte, unveränderliche (funktionale) und veränderliche (imperative) Objekte, Zustand von veränderlichen Objekten.

Imperative Prozeduren und Einkapselung von Referenzen.

Imperative und funktionale Datenstrukturen.

Reihungen: Intervalltest mit Logbuch; Reversieren und Sortieren durch Umordnung der Komponenten; Sortieren durch Auswählen und Quicksort.

Agenden: Stapel, Schlangen, LIFO, FIFO.

Effiziente Realisierung imperativer Schlangen mit verzeigerten Zellen; Kastendarstellung.

Schleifen: Bedingung, Rumpf; Zusammenhang mit Endrekursion.

Lineare Darstellung von Listen, Bäumen und Ausdrücken: lineare Speicher; Halde, Adressen, Blöcke, Indizes; verzeigerte Blockdarstellung; Strukturzusammenlegung.

Speicherplatzbedarf bei der Programmausführung, Speicherbereiniger.

Funktionale, imperative und objektorientierte Sprachen.

16 Stapelmaschinen und Übersetzer

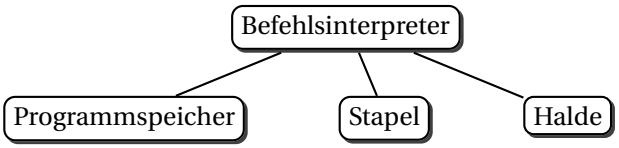
An der *Software-Hardware-Schnittstelle* von Computern findet sich eine primitive Programmiersprache, deren Befehle direkt durch die Hardware des Computers ausgeführt werden. Programmiersprachen, die sich an der Struktur der Software-Hardware-Schnittstelle orientieren, werden als *maschinennah* bezeichnet.

In diesem Kapitel betrachten wir zwei maschinennahe Programmiersprachen W und M. W verfügt über imperative Variablen und Schleifen und findet sich als Teilsprache in gängigen Programmiersprachen wie C und Java wieder. Mit M lernen wir das maschinennahe Ausführungsmodell für Programmiersprachen mit rekursiven Prozeduren kennen. Die Programme von M sind einfache Befehlsfolgen, die Konditionale und Schleifen mit Sprüngen realisieren. Wie bei der Maschinensprache eines Computers beschreiben die Befehle von M in sich abgeschlossene Aktionen, die den Speicherzustand der ausführenden Maschine verändern.

In diesem Kapitel entwickeln wir auch einen Übersetzer, der W nach M übersetzt. Damit erklären wir die Rückführung einer rekursiven Programmstruktur mit Bezeichnern auf eine lineare Befehlsstruktur ohne Bezeichner.

16.1 Eine Stapelmaschine

Die Sprache M beschreiben wir durch eine Maschine, die in M geschriebene Programme ausführt. Wir bezeichnen sowohl die Sprache als auch die Maschine mit M. Die Maschine M besteht aus einem Befehlsinterpreter und drei linearen Speichern:



Die Maschine kennt nur Werte des Typs *int*. Der **Stapel** und die **Halde** sind lineare Speicher, die gemäß § 15.6 und § 15.9 verwaltet werden. Das die Maschine steuernde **Programm** ist eine Folge von **Befehlen**, die im **Programmspeicher** beginnend mit der Adresse 0 abgelegt sind. Jeder Befehl belegt eine Zelle des Programmspeichers. Der Programmspeicher verfügt über eine spezielle, als **Programmzähler** bezeichnete Zelle, in der die Adresse des jeweils als Nächstes auszuführenden Befehls steht.

Der **Befehlsinterpreter** führt die Befehle des Programms aus. Die Befehle beschreiben Aktionen, die den Zustand des Programmzählers, des Stapels und der Halde ändern können. Die Befehle werden immer am Stück ausgeführt, das heißt die Ausführung eines Befehls beginnt erst, nachdem die Ausführung des vorangegangenen Befehls vollständig abgeschlossen ist.

Beim Start der Maschine sind der Stapel und die Halde leer, das Programm steht im Programmspeicher und der Programmzähler hat den Wert 0. Die Maschine führt dann mithilfe des Befehlsinterpreters die Befehle des Programms aus. Dabei wird jeweils der durch den Programmzähler bestimmte Befehl ausgeführt. Da der Programmzähler zu Beginn den Wert 0 hat, beginnt die Maschine mit der Ausführung des nullten Befehls des Programms.

Die Ausführung eines Befehls terminiert immer, wobei zwischen regulärer und irregulärer Terminierung zu unterscheiden ist. Bei regulärer Terminierung fährt die Maschine mit der Ausführung des Befehls fort, dessen Adresse zu diesem Zeitpunkt im Befehlszähler steht. Bei irregulärer Terminierung bleibt die Maschine stehen.

Die meisten Befehle erhöhen den Wert des Programmzählers um eins, sodass als Nächstes der im Programm folgende Befehl ausgeführt wird. Mit den sogenannten Sprungbefehlen können durch entsprechende Veränderung des Programmzählers Programmteile übersprungen oder wiederholt werden.

Da der Stapel für die Funktionsweise der Maschine M von besonderer Bedeutung ist, bezeichnen wir M auch als **Stapelmaschine**.

Wir werden mit einer Realisierung der Maschine M in Standard ML arbeiten, die uns in die Lage versetzt, mit der Maschine zu experimentieren. Außerdem liefert uns die Realisierung in Standard ML eine präzise Definition der Maschine, die die informellen Ausführungen dieses Kapitels ergänzt. Die Realisierung der Maschine finden Sie auf den Webseiten dieses Buchs.

Die Befehle der Maschine M stellen wir durch die Werte eines Konstruktortyps *instruction* dar, dessen Deklaration Sie in Abbildung 16.1 finden. Jede Liste von Befehlen stellt ein mögliches Programm für die Maschine dar und wird als **Code** bezeichnet. Wir unterscheiden die folgenden Befehlsgruppen:

- Arithmetische Befehle: *con*, *add*, *sub*, *mul*, *halt*.
- Test- und Sprungbefehle: *leq*, *branch*, *cbranch*.
- Imperative Variablen: *getS*, *putS*.
- Benutzung der Halde: *new*, *getH*, *putH*.
- Prozeduren: *proc*, *arg*, *call*, *return*, *callR*.

Aufgabe 16.1 Beantworten Sie die folgenden Fragen:

a) Welchen Wert hat der Programmzähler beim Start der Maschine?

b) Was passiert, wenn die Ausführung eines Befehls regulär terminiert?

c) Kann die Ausführung eines Befehls divergieren?

```
type index = int
type noi   = int                (* number of instructions      *)
type noa   = int                (* number of arguments         *)
type sa    = int                (* stack address               *)
type ca    = int                (* code address                *)

datatype instruction =
    halt                        (* halt machine                *)
  | con     of int              (* push constant               *)
  | add                         (* addition                    *)
  | sub                         (* substraction                *)
  | mul                         (* multiplication              *)
  | leq                         (* less or equal test          *)
  | branch  of noi              (* unconditional branch        *)
  | cbranch of noi              (* conditional branch          *)
  | getS    of sa               (* push value from stack cell  *)
  | putS    of sa               (* update stack cell           *)
  | new     of noa              (* allocate block of size noa  *)
  | getH    of index            (* push value from heap cell   *)
  | putH    of index            (* update heap cell            *)
  | proc    of noa * noi        (* begin of procedure code     *)
  | arg     of index            (* push argument               *)
  | call    of ca               (* call procedure              *)
  | return                      (* return from procedure call  *)
  | callR   of ca               (* call procedure and return   *)

type code = instruction list
```

Abbildung 16.1: *Befehle der Maschine M*

16.2 Arithmetische Befehle

Der einfachste Befehl ist *halt*. Dieser Befehl hält die Maschine an und wirft die Ausnahme *Halt*. Der Befehl *con n* legt die Zahl *n* auf den Stapel. Der Befehl *add* nimmt zwei Zahlen vom Stapel, addiert sie und legt das Ergebnis auf den Stapel. Nach der Ausführung des Befehls *add* hat der Stapel also einen Eintrag weniger als vorher. Die Befehle *con* und *add* erhöhen den Programmzähler jeweils um eins. Damit wissen wir genug, um das folgende Programm zu verstehen, das die Zahlen 4 und 7 addiert und das Ergebnis auf den Stapel legt:

```
[con 4, con 7, add, halt]
```

Im Folgenden gehen wir davon aus, dass Sie die Deklarationen für die Maschine M in einen Standard ML Interpreter geladen haben. Sie können dann Maschinenprogramme mit der Prozedur *exec* : *code* → *int list* ausführen. Diese lädt das gegebene Programm in

den Programmspeicher, setzt den Programmzähler auf 0 und versetzt den Stapel und die Halde in den leeren Anfangszustand. Danach wird das Programm beginnend mit dem ersten Befehl ausgeführt. Wenn die Ausführung des Programms regulär terminiert, liefert *exec* die dann vorliegende Stapelbelegung:

```
exec [con 1, con 2, con 3, con 4, halt]
```
[1, 2, 3, 4] : int list

```
exec [con 4, con 7, add, halt]
```
[11] : int list

Mit den Befehlen *sub* und *mul* kann man analog zu *add* subtrahieren und multiplizieren. Bei der Subtraktion ist die Reihenfolge der Argumente von Bedeutung. Generell müssen die Argumente so auf den Stapel gelegt werden, dass der konsumierende Befehl seine Argumente in der richtigen Reihenfolge bekommt, wenn er sie nacheinander vom Stapel nimmt. Das Programm

```
[con 4, con 7, sub, halt]
```

legt erst 4 und dann 7 auf den Stapel. Entsprechend nimmt der Subtraktionsbefehl zuerst 7 und dann 4 vom Stapel und berechnet die Differenz $7-4$.

Den Wert des Ausdrucks $4*3-7*2$ können wir mit dem folgenden Programm berechnen:

```
[con 2, con 7,       (*  2, 7      *)
 mul,                (*  14        *)
 con 3, con 4,       (*  14, 3, 4  *)
 mul,                (*  14, 12    *)
 sub, halt]          (*  -2        *)
```

In den Kommentarklammern ist jeweils der Stapelzustand nach Ausführung der entsprechenden Zeile angegeben.

Beim Suchen nach Programmierfehlern ist es oft hilfreich, das Programm schrittweise auszuführen und die Stapelbelegung nach jedem Befehl zu prüfen. Das kann mit den Prozeduren *load* : *code* → *unit* und *step* : *unit* → *int list* geschehen:

```
load [con 2, con 7, con 8, add, sub, halt]
```
() : unit

```
step ()
```
[2] : int list

```
step ()
```
[2, 7] : int list

```
step ()
```
[2, 7, 8] : int list

```
step ()
```
[2, 15] : int list

```
step ()
```
[13] : int list

Die bisher eingeführten Befehle reichen aus, um beliebig geschachtelte arithmetische Ausdrücke in Maschinenprogramme zu übersetzen. Hinter dieser Übersetzung steckt ein allgemeines Prinzip, gemäß dem baumartige Strukturen in lineare Strukturen überführt werden können. Dabei reduziert sich bei Algorithmen Baumrekursion auf Endrekursion. Die nachfolgenden Aufgaben sollen Ihnen helfen, diesen wichtigen Zusammenhang zu verstehen.

Aufgabe 16.2 Schreiben Sie Maschinenprogramme, die die folgenden Ausdrücke auswerten. Testen Sie Ihre Programme mit *exec*, *load* und *step*.

a) $17-3-5$

b) $17-(3-5)$

c) $(7-3)(8+17)+6$

Aufgabe 16.3 (Übersetzung) Schreiben Sie eine Prozedur $compile : exp \to code$, die einen Ausdruck in ein Maschinenprogramm übersetzt, das den Ausdruck auswertet. Legen Sie dabei die folgenden Ausdrücke zugrunde:

```
datatype exp =
    Con of int              (* constant       *)
  | Add of exp * exp        (* addition       *)
  | Sub of exp * exp        (* subtraction    *)
  | Mul of exp * exp        (* multiplication *)
```

Aufgabe 16.4 (Arithmetische Maschine) Schreiben Sie eine endrekursive Prozedur $run : int\ list \to code \to int$, sodass *run nil* (*compile e*) den Wert des Ausdrucks *e* liefert. Das erste Argument von *run* soll den für die Ausführung des Maschinenprogramms benötigten Stapel darstellen (oberstes Element des Stapels als erstes Element der Liste). Für Maschinenprogramme, die nicht mit *compile* darstellbar sind, soll *run* die Ausnahme *Domain* werfen.

Aufgabe 16.5 (Rückübersetzung) Wenn man arithmetische Ausdrücke nach dem beschriebenen Schema in Maschinenprogramme übersetzt, geht keine Information verloren. Also ist eine Rückübersetzung der Maschinenprogramme in die Ausdrücke möglich. Dies gelingt mit einer Maschine, die auf ihrem Stapel die bereits erkannten Teilausdrücke ablegt. Schreiben Sie eine entsprechende Prozedur $decompile : code \to exp$. Für Maschinenprogramme, die nicht mit *compile* darstellbar sind, soll *decompile* die Ausnahme *Domain* werfen.
Hilfestellung: Schreiben Sie *decompile* mithilfe einer an der Prozedur *run* aus Aufgabe 16.4 angelehnten Prozedur $decompile' : exp\ list \to code \to exp$.

Aufgabe 16.6 (Baumrekonstruktion aus der Postlinearisierung) Die Übersetzung von arithmetischen Ausdrücken in Code für M entspricht bis auf die Reversion der Argumente der Postlinearisierung von Bäumen (§ 7.6.3). Dementsprechend können wir einen Baum aus seiner Postlinearisierung rekonstruieren, indem wir die Postlinearisierung mit einer Maschine ausführen, die auf ihrem Stapel die bereits erkannten Teilbäume ablegt.

a) Schreiben Sie eine Prozedur $post : tree \to int\ list$, die die Postlinearisierung eines Baums liefert.

b) Schreiben Sie eine Prozedur $depost : int\ list \to tree$, die mit *post* übersetzte Bäume rückübersetzt. Verwenden Sie dabei eine endrekursive Hilfsprozedur $depost' : tree\ list \to tree\ list \to int\ list \to tree$, die die bereits gebildeten Bäume und eine Unterbaumliste als Akkus erhält.

c) Schreiben Sie eine Prozedur $depre : int\ list \to tree$, die die Prälinearisierung von Bäumen rückübersetzt. Reversieren Sie dazu die Prälinearisierung und verwenden Sie eine Variante der Prozedur $depost'$, die beim Bilden eines Baums die Unterbaumlisten reversiert.

d) Vergleichen Sie die Rekonstruktion von Bäumen in dieser Aufgabe mit der auf rekursivem Abstieg beruhenden Rekonstruktion in Aufgabe 13.7 auf S. 266. Machen Sie sich klar, dass das hier verwendete stapelbasierte Verfahren im Gegensatz zum rekursiven Abstieg nur Endrekursion benötigt.

16.3 Sprungbefehle und Konditionale

Für die Realisierung von Konditionalen stellt die Maschine M einen Testbefehl und zwei Sprungbefehle zur Verfügung:

- Der **Testbefehl** *leq* nimmt zwei Zahlen vom Stapel und vergleicht diese. Wenn die erste Zahl kleiner gleich der zweiten ist, wird die Zahl 1 auf den Stapel gelegt, ansonsten die Zahl 0.
- Der **unbedingte Sprungbefehl** *branch i* addiert i zum Programmzähler hinzu ($pc :=! pc + i$).
- Der **bedingte Sprungbefehl** *cbranch i* nimmt die oberste Zahl vom Stapel. Falls diese 0 ist, wird i wie beim unbedingten Sprungbefehl zum Programmzähler hinzuaddiert. Ansonsten wird der Programmzähler wie bei den arithmetischen Befehlen um eins erhöht.

Die Sprungbefehle ermöglichen es, im Programm vor (positives i) oder zurück (negatives i) zu springen. Als Beispiel betrachten wir den konditionalen Ausdruck

```
if 1<=2 then 4-3 else 7*5
```

den wir mit dem folgenden Programm auswerten können:

```
[con 2, con 1, leq, cbranch 5,    (* if 1<=2   *)
 con 3, con 4, sub, branch 4,     (* then 4-3   *)
 con 5, con 7, mul,               (* else 7*5   *)
 halt]
```

Zuerst kommen die Befehle für die Bedingung. Danach kommt ein bedingter Sprung, der die Konsequenz überspringt, falls die Bedingung zu 0 evaluiert. Danach kommen die Befehle für die Konsequenz und ein Sprung über die Alternative. Schließlich folgen die Befehle für die Alternative.

Aufgabe 16.7 Schreiben Sie ein Maschinenprogramm, das den konditionalen Ausdruck *if* $2 \cdot 3 \leq 12 - 3$ *then 3 else 5* auswertet.

Aufgabe 16.8 Geben Sie ein möglichst kurzes Maschinenprogramm an, dessen Ausführung divergiert.

Aufgabe 16.9 Geben Sie ein möglichst kurzes Programm p an, sodass das Programm *con* k :: p für positives k terminiert und für nicht positives k divergiert.

16.4 Imperative Variablen und Schleifen

Mit den Sprungbefehlen lassen sich neben Konditionalen auch Schleifen (§ 15.8) formulieren. Damit wir sinnvolle Programme mit Schleifen schreiben können, benötigen wir allerdings Zellen, die wir lesen und schreiben können. Diese allozieren wir im Stapel. Mit den folgenden Befehlen können diese Zellen gelesen und geschrieben werden:

- *getS i* legt den Wert auf den Stapel, der im Stapel an der Adresse i steht. Dabei hat die unterste Zelle des Stapels die Adresse 0.
- *putS i* nimmt das oberste Element vom Stapel und überschreibt damit den Wert, der an der Adresse i des Stapels steht.

Als Beispiel betrachten wir das Programm

```
[con 7, con 5, getS 1, getS 0, sub, putS 0, mul, halt]
```

dessen Ausführung den Stapel wie folgt verändert:

$$[] \to [7] \to [7,5] \to [7,5,5] \to [7,5,5,7] \to [7,5,2] \to [2,5] \to [10]$$

Wir sind jetzt in der Lage, interessante Berechnungen mit Schleifen zu programmieren. Als Beispiel wollen wir ein Programm betrachten, das die Fakultät von 10 berechnet. Damit wir den Überblick behalten, formulieren wir das Programm zunächst in einer gedachten Programmiersprache, die wir **W** nennen:[1]

[1] W steht für while.

```
var n := 10
var a := 1
while n>=2 do
   a := n*a ;
   n := n-1
end
return a
```

Bei der Übersetzung in ein Maschinenprogramm allozieren wir die **imperativen Variablen** n und a an den Adressen 0 und 1 des Stapels.

```
[con 10,                          (* var n := 10       [0] *)
 con 1,                           (* var a := 1        [1] *)
 getS 0, con 2, leq, cbranch 10,  (* while 2 <= n do       *)
 getS 1, getS 0, mul, putS 1,     (*   a := n * a ;        *)
 con 1, getS 0, sub, putS 0,      (*   n := n - 1          *)
 branch ~12,                      (* end                   *)
 putS 0, halt]                    (* return a              *)
```

Die Schleife wird mit einem bedingten **Vorwärtssprung** (positives Argument) und einem unbedingten **Rückwärtssprung** (negatives Argument) realisiert. Der Befehl *putS 0* vor dem Haltebefehl deallociert die Variable a, sodass nur noch der finale Wert von a auf dem Stapel liegt.

Es ist sinnvoll, bei imperativen Programmen zwischen der **Eingabe** und der **Ausgabe** von Daten zu unterscheiden, wobei es die Aufgabe des Programms ist, aus den Eingabedaten die Ausgabedaten zu berechnen. In W erfolgt die Ausgabe mit der mit *return* formulierten Anweisung. Die Eingabe kann über eine Variable erfolgen, für die kein initialer Wert deklariert ist:

```
var n                             (* Eingabevariable *)
var a := 1
while n>=2 do
   a := n*a ;
   n := n-1
end
return a
```

In der Sprache M erfolgt die Kommunikation der Ein- und Ausgabedaten über den Stapel. Wenn wir bei dem obigen Maschinenprogramm für die Berechnung der Fakultät den einleitenden Befehl *con 10* weglassen, haben wir ein Programm p, das zu n die Fakultät von n berechnet. Um mit p die Fakultät für ein konkretes k zu berechnen, muss M allerdings das Programm *con* k :: p ausführen. Der einleitende Befehl *con* k dient dazu, die Eingabezahl k auf den Stapel zu legen.

Aufgabe 16.10 Schreiben Sie ein Programm, das zu x und n mit einer Schleife die Potenz x^n berechnet. Schreiben Sie das Programm so wie oben gezeigt erst in W und dann in M. Achten Sie darauf, dass Ihr Programm nur die Potenz x^n im Stapel zurücklässt.

Aufgabe 16.11 Schreiben Sie ein Programm, das zu zwei positiven Zahlen x und y mit einer Schleife den größten gemeinsamen Teiler berechnet. Schreiben Sie das Programm so wie oben gezeigt erst in W und dann in M.

Aufgabe 16.12 Nehmen Sie an, dass der Programmspeicher der Maschine wie folgt belegt ist:

0	1	2	3	4	5	6	7	8	9	10
con 2	*getS 0*	*con 0*	*leq*	*cbranch 6*	*con 1*	*getS 0*	*sub*	*putS 0*	*branch ~8*	*halt*

Simulieren Sie die Ausführung des Programms mit Papier und Bleistift.

a) Geben Sie die Folge der Adressen an, die der Programmzähler durchläuft. Hilfe: Die Folge beginnt mit 0 und endet mit 10. Sie hat insgesamt 33 Elemente.

b) Wird jeder Befehl des Programms mindestens einmal ausgeführt?

c) Geben Sie die Adressen im Programmspeicher an, die genau einmal zur Ausführung kommen.

d) Skizzieren Sie das Maschinenprogramm durch ein Programm in W.

16.5 Verwendung der Halde

Wir haben bereits gelernt, dass sich zusammengesetzte Objekte linear in einer Halde darstellen lassen. Mit den folgenden Befehlen können wir in der Halde der Maschine M Blöcke allozieren und deren Zellen lesen und ihnen Werte zuweisen:

- *new n* nimmt $n \geq 1$ Werte vom Stapel und legt die Adresse eines neu allozierten Blocks auf den Stapel, in dessen Zellen die vom Stapel genommenen Werte gelegt werden. Die Zellen des Blocks werden mit den Indizes 0 bis $n-1$ adressiert, wobei die Zelle 0 mit dem zuerst vom Stapel genommenen Wert belegt wird.
- *getH i* nimmt einen Wert a vom Stapel und legt dafür den Wert der i-ten Zelle des durch a adressierten Blocks auf den Stapel.
- *putH i* nimmt zwei Werte a, x vom Stapel und schreibt x in die i-te Zelle des durch a adressierten Blocks.

Als Beispiel betrachten wir ein Maschinenprogramm, das das Tupel $(1,2,3)$ in der Halde ablegt und die Adresse des Tupels auf den Stapel legt.

```
exec [con 3, con 2, con 1, new 3, halt]
[0] : int list

showHeap ()
[(0, 1), (1, 2), (2, 3)] : (int * int) list
```

Wir wagen uns jetzt an ein schwierigeres Programm, das zu n die Liste $[0,1,\ldots,n]$ berechnet. Dabei stellen wir die Liste so wie in § 15.10 gezeigt in der Halde dar und liefern die Adresse des ersten Blocks der Liste als Ergebnis. Zunächst skizzieren wir das Programm in W:

```
var n
var xs := -1
while 0<=n do
    xs := new(n,xs) ;
    n := n-1
end
return xs
```

Der Ausdruck *new*(n,xs) alloziert einen Block mit zwei Zellen, in die er die Werte der Variablen n und xs ablegt. Als Ergebnis liefert er die Adresse des neu allozierten Blocks. Durch Übersetzung bekommen wir das folgende Maschinenprogramm:

```
val p =
[con ~1,                          (* var xs := -1           *)
 getS 0, con 0, leq, cbranch 10,  (* while 0<=n do          *)
 getS 1, getS 0, new 2, putS 1,   (*     xs := new(n,xs) ;  *)
 con 1, getS 0, sub, putS 0,      (*     n := n-1           *)
 branch ~12,                      (* end                    *)
 putS 0, halt]                    (* return xs              *)
```

Die Ausführung des Programms mit der Eingabe 4 liefert den Stapel

```
exec (con 4::p)
[8] : int list
```

und die Halde

```
showHeap ()
[(0,4), (1,~1), (2,3), (3,0), (4,2), (5,2), (6,1), (7,4), (8,0), (9,6)] : (int * int) list
```

Aufgabe 16.13 Zeichnen Sie die verzeigerte Blockdarstellung (§ 15.10) der durch die obige Programmausführung allozierten Liste.

Aufgabe 16.14 Schreiben Sie ein Programm, das das geschachtelte Tupel $(1,(2,3))$ in der Halde darstellt. Orientieren Sie sich dabei an der linearen Darstellung von Ausdrücken aus § 15.12. Testen Sie Ihr Programm mit den Prozeduren *load*, *step* und *showHeap*. Den Wert des Programmzählers können Sie durch die Eingabe von !*pc* abfragen.

Aufgabe 16.15 Schreiben Sie ein Programm (in W und M), das zu n die Liste $[n,\ldots,0]$ berechnet.

Aufgabe 16.16 Schreiben Sie ein Maschinenprogramm, das die Länge einer Liste berechnet. Halten Sie sich an die folgende Skizze in W:

```
var xs
var n := 0
while 0<=xs do
   n := n+1 ;
   xs := xs.1
end
return n
```

Der Ausdruck *xs.1* liefert den Wert in der zweiten Zelle des Blocks, dessen Adresse der Wert der Variablen *xs* ist.

Aufgabe 16.17 Sie sollen Programme in W und M schreiben, die Listen auf zwei verschiedene Weisen reversieren.

a) Schreiben Sie die Programme so, dass die Darstellung der zu reversierenden Liste nicht verändert wird.

b) Schreiben Sie die Programme so, dass keine neuen Blöcke in der Halde alloziert werden. Stattdessen soll die Darstellung der zu reversierenden Liste verändert werden. Benutzen Sie in W Zuweisungen der Form *xs.1* := *ys*, um den Zellen von Blöcken Werte zuzuweisen. Kniffelig!

Aufgabe 16.18 (Obskure Programme) M hat die für maschinennahe Sprachen typische Eigenschaft, dass es nicht explizit zwischen Adressen und Zahlen unterscheidet. Betrachten Sie dazu die folgenden obskuren Programme und analysieren Sie, was diese tun.

a) `[con 5,getH 0, halt]`

b) `[con 3, con 5, new 2, con 1, getH 0, halt]`

c) `[con 8, new 1, con 7, con 1, new 2, getS 0, getH 2, halt]`

d) `[con 2, con 3, new 2, con 7, con 1, getS 0, add, putH 0, halt]`

16.6 Ein Übersetzer

Es ist nicht schwer, einen Übersetzer für W zu schreiben. Darunter wollen wir eine Prozedur verstehen, die Programme der Sprache W in Maschinenprogramme für M übersetzt. Um den Übersetzer schreiben zu können, benötigen wir zunächst eine abstrakte Syntax für W.

Wir beschränken uns auf eine vereinfachte Version W1 von W, die auf Eingabevariablen und den Zugriff auf die Halde verzichtet. Die konkrete und abstrakte Syntax von W1 beschreiben wir etwas salopp mit der Grammatik in Abbildung 16.2. In W1 werden die Bedingungen von Konditionalen und Schleifen mit ganzzahligen Ausdrücken beschrieben, wobei 0 als false und die anderen Zahlen als true interpretiert werden. Die Ausführung von Ausdrücken liefert einen Wert, kann aber die Werte der Variablen nicht verändern. Die Ausführung von Anweisungen liefert dagegen keinen Wert, kann aber die Werte der

$$
\begin{aligned}
\textit{Programm} &::= \textit{Deklaration} \ldots \textit{Deklaration} \\
&\quad\ \textit{Hauptanweisung} \\
&\quad\ \textit{Ausgabeanweisung} \\
\textit{Deklaration} &::= \texttt{"var"}\ \textit{Bezeichner}\ \texttt{":="}\ \textit{Ausdruck} \\
\textit{Ausdruck} &::= \textit{Zahl} \mid \textit{Bezeichner} \mid \texttt{"("}\ \textit{Ausdruck}\ \texttt{")"} \\
&\quad \mid \textit{Ausdruck}\ (\texttt{"+"} \mid \texttt{"-"} \mid \texttt{"*"} \mid \texttt{"}\leq\texttt{"})\ \textit{Ausdruck} \\
\textit{Hauptanweisung} &::= \textit{Anweisung} \\
\textit{Anweisung} &::= \textit{Zuweisung} \mid \textit{Konditional} \mid \textit{Schleife} \\
&\quad \mid \textit{Sequentialisierung} \mid \texttt{"("}\ \textit{Anweisung}\ \texttt{")"} \\
\textit{Zuweisung} &::= \textit{Bezeichner}\ \texttt{":="}\ \textit{Ausdruck} \\
\textit{Konditional} &::= \texttt{"if"}\ \textit{Ausdruck}\ \texttt{"then"}\ \textit{Anweisung}\ \texttt{"else"}\ \textit{Anweisung} \\
\textit{Schleife} &::= \texttt{"while"}\ \textit{Ausdruck}\ \texttt{"do"}\ \textit{Anweisung}\ \texttt{"end"} \\
\textit{Sequentialisierung} &::= \textit{Anweisung}\ \texttt{";"}\ \textit{Anweisung} \\
\textit{Ausgabeanweisung} &::= \texttt{"return"}\ \textit{Ausdruck}
\end{aligned}
$$

Abbildung 16.2: *Syntax von W1*

Variablen verändern. Von dieser Regel weicht die Ausgabeanweisung ab, die einen Wert als Ergebnis liefert und die Zellen für die deklarierten Variablen im Stapel deallociert.

Die in Abbildung 16.3 gezeigten Typdeklarationen realisieren die abstrakte Syntax von W1 in Standard ML. Den Übersetzer für W1 realisieren wir durch eine Prozedur *compile* : *program* → *code*, die in Abbildung 16.4 auf S. 338 deklariert ist. Zunächst werden die Deklarationen des Programms mit der Prozedur *compDec* übersetzt. Die deklarierten Variablen werden beginnend mit der Adresse 0 im Stapel alloziert. Der Zusammenhang zwischen den Bezeichnern und den Adressen der Variablen wird durch Umgebungen dargestellt, die so wie im Kapitel über Semantik realisiert sind (siehe Abbildung 12.6 auf S. 248). Nach den Deklarationen wird zunächst die Hauptanweisung des Programms mit der Prozedur *compSta* übersetzt. Danach wird die Ausgabeanweisung mit den Prozeduren *compExp* und *compRet* übersetzt. Dabei erledigt *compRet* die Deallokation der für die Variablen allozierten Zellen.

Abschließend wollen wir noch einige der für die Übersetzung von W1 wichtigen Aspekte festhalten:

1. Variablen werden gemäß den Deklarationen im Stapel alloziert und durch die Ausgabeanweisung deallociert. Der Zusammenhang zwischen Bezeichnern und Adressen wird durch Umgebungen dargestellt.
2. Ein Ausdruck wird in eine Befehlsliste übersetzt, deren Ausführung den Stapel um den Wert erweitert, den die Auswertung des Ausdrucks liefert. Die bereits existierenden Stapelelemente bleiben dabei unverändert.

```
type     id  = string                (* identifier   *)
datatype exp =                       (* expression   *)
     Con    of int
   | Var    of id
   | Add    of exp * exp
   | Sub    of exp * exp
   | Mul    of exp * exp
   | Leq    of exp * exp
datatype sta =                        (* statement   *)
     Assign of id  * exp
   | If     of exp * sta * sta
   | While  of exp * sta
   | Seq    of sta * sta
type     dec = id * exp               (* declaration *)
type program = dec list * sta * exp
```

Abbildung 16.3: *Realisierung der abstrakten Syntax von W1*

3. Die Hauptanweisung wird in eine Befehlsliste übersetzt, deren Ausführung die Werte der im Stapel allozierten Variablen verändern kann. Nach Ausführung der Befehlsliste hat der Stapel genau so viele Elemente wie vorher.

Aufgabe 16.19 Stellen Sie das Programm für die Berechnung der zehnten Fakultät aus § 16.4 durch einen Ausdruck in Standard ML dar. Übersetzen Sie den Ausdruck mit der Prozedur *compile*.

Aufgabe 16.20 Erweitern Sie den Übersetzer für W1 um Eingabevariablen. Erweitern Sie dazu die in Abbildung 16.3 auf S. 337 gezeigte Programmdarstellung um eine Komponente für die Liste der Eingabevariablen. Sorgen Sie dafür, dass die Ausgabeanweisung auch die Zellen für die Eingabevariablen deralloziert.

Aufgabe 16.21 Erweitern Sie die Syntax (Abbildung 16.2 auf S. 336), die Realisierung der abstrakten Syntax (Abbildung 16.3) und den Übersetzer (Abbildung 16.4) für W1 um Konstrukte für die Benutzung der Halde. Orientieren Sie sich an den Beispielen in § 16.5.

Aufgabe 16.22 (Interpreter) Schreiben Sie eine Prozedur $eval : program \to int$, die W1-Programme ausführt und die Ergebnisse liefert. Orientieren Sie sich an der Architektur des Übersetzers für W1 und verwenden Sie Umgebungen des Typs *int ref env*.

```
fun compExp f (Con c)      = [con c]
  | compExp f (Var x)      = [getS (f x)]
  | compExp f (Add(e1,e2)) = compExp f e2 @ compExp f e1 @ [add]
  | compExp f (Sub(e1,e2)) = compExp f e2 @ compExp f e1 @ [sub]
  | compExp f (Mul(e1,e2)) = compExp f e2 @ compExp f e1 @ [mul]
  | compExp f (Leq(e1,e2)) = compExp f e2 @ compExp f e1 @ [leq]
```

compExp : sa env → exp → code

```
fun compDec f sa nil = (f, nil)
  | compDec f sa ((x,e)::dr) =
              let val (f', cdr) = compDec (update f x sa) (sa+1) dr
              in  (f', compExp f e @ cdr) end
```

*compDec : sa env → sa → dec list → sa env * code*

```
fun compSta f (Assign(x,e)) = compExp f e @ [putS (f x)]
  | compSta f (If(e,s1,s2)) =
              let val (ce,cs1,cs2) = (compExp f e, compSta f s1, compSta f s2)
              in  ce  @ [cbranch (length cs1 + 2)] @
                  cs1 @ [branch (length cs2 + 1)] @
                  cs2
              end
  | compSta f (While(e,s)) =
              let val (ce,cs) = (compExp f e, compSta f s)
              in  ce @ [cbranch (length cs + 2)] @
                  cs @ [branch (~(length cs + 1 + length ce))]
              end
  | compSta f (Seq(s1,s2)) = compSta f s1 @ compSta f s2
```

compSta : sa env → sta → code

```
fun compRet n = if n=0 then [halt] else putS (n-1) :: compRet (n-1)
```

compRet : int → code

```
fun compile (ds,s,e) =
              let val (f,cds) = compDec empty 0 ds
              in  cds @ compSta f s @
                  compExp f e @ compRet (length ds)
              end
```

compile : program → code

Abbildung 16.4: *Übersetzer für W1*

16.7 Prozedurbefehle

Die Maschine M verfügt auch über Befehle zur Realisierung von Prozeduren. Dabei handelt es sich um **stellige Prozeduren**, die über eine frei wählbare Anzahl von Argumenten verfügen. Eine n-stellige Prozedur wird durch eine Befehlssequenz

$$proc(n,k),\ Befehl_1,\ \ldots\ ,\ Befehl_{k-1}$$

beschrieben. Der einleitende Befehl $proc(n,k)$ spezifiziert die Anzahl der Argumente (n) und die Anzahl der die Prozedur beschreibenden Befehle (k). Als konkretes Beispiel betrachten wir eine zweistellige Prozedur *max*, die das Maximum zweier Zahlen liefert:

```
val max =
[proc(2,9),                          (*  fun max x y =  *)
 arg 2, arg 1, leq, cbranch 3,       (*  if x<=y        *)
 arg 2, return,                      (*  then y         *)
 arg 1, return]                      (*  else x         *)
```

Der **Argumentbefehl** *arg i* legt das i-te Argument auf den Stapel. Der **Rückkehrbefehl** *return* beendet die Ausführung des Prozeduraufrufs und liefert das oberste Element des Stapels als Ergebnis. Hier ist ein Programm, das mithilfe der Prozedur *max* das Maximum der Zahlen 7 und 13 berechnet:

```
exec ([con 7, con 13, call 4, halt] @ max)
```
[13] : int list

Der Aufruf der Prozedur *max* erfolgt mit dem **Aufrufbefehl** *call 4*, der die aufzurufende Prozedur durch die Anfangsadresse ihrer Befehlssequenz im Programmspeicher identifiziert. Die Argumente und das Ergebnis des Prozeduraufrufs werden dabei wie bei den arithmetischen Befehlen mithilfe des Stapels übergeben.

Die Ausführung des Befehls $proc(n,k)$ hat denselben Effekt wie der unbedingte Sprungbefehl *branch k*. Das bedeutet, dass wir die Befehlssequenz für eine Prozedur an einer beliebigen Stelle im Programm platzieren können, da sie bei der Ausführung übersprungen wird:

```
exec ([con 7, con 13] @ max @ [call 2, halt])
```
[13] : int list

Beachten Sie, dass die Maschine bei einem Prozeduraufruf *call k* mit dem Befehl $k+1$ fortfährt, damit der einleitende Prozedurbefehl nicht zur Ausführung kommt.

Mit den Prozedurbefehlen lassen sich auch rekursive Prozeduren realisieren. Als Beispiel betrachten wir eine Prozedur *power*, die zu x und n die Potenz x^n liefert:

```
val power =
[proc(2,15),                          (*  fun power x n =         *)
 con 0, arg 2, leq, cbranch 3,        (*  if n<=0                 *)
 con 1, return,                       (*  then 1                  *)
 con 1, arg 2, sub, arg 1, call 0,    (*  else x * power x (n-1)  *)
 arg 1, mul, return]
```

Der rekursive Aufruf von *power* geht davon aus, dass die Befehlssequenz der Prozedur im Programmspeicher mit der Adresse 0 beginnt. Die Potenz 2^{10} können wir mit *power* wie folgt berechnen:

```
exec (power @ [con 10, con 2, call 0, halt])
```
[1024] : int list

Aufgabe 16.23 Realisieren Sie eine dreistellige Prozedur, die ihr kleinstes Argument als Ergebnis liefert:

a) Realisieren Sie die Prozedur in Standard ML.

b) Realisieren Sie die Prozedur mit einer Befehlssequenz für M.

c) Schreiben Sie ein Maschinenprogramm, das das Minimum der Zahlen 7, 3, 9 mithilfe der Prozedur bestimmt.

Aufgabe 16.24 Übersetzen Sie die folgende Prozedur nach M:

```
fun fac n = if n<2 then 1 else fac(n-1)*n
```

Nehmen Sie dabei an, dass die Befehlssequenz für die Prozedur im Programmspeicher mit der Adresse 0 beginnt.

Aufgabe 16.25 Übersetzen Sie die folgende Prozedur nach M:

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

Aufgabe 16.26 Übersetzen Sie die folgende Prozedur nach M:

```
fun revi (nil, ys) = ys
  | revi (x::xr, ys) = revi (xr, x::ys)
```

Aufgabe 16.27 (Übersetzer für S) Schreiben Sie einen Übersetzer für eine einfache funktionale Sprache S mit der folgenden Syntax:

Programm ::= *Deklaration* … *Deklaration* "`return`" *Ausdruck*

Deklaration ::= "`fun`" *Bezeichner Bezeichner* … *Bezeichner* "`=`" *Ausdruck*

Ausdruck ::= *Zahl* | *Bezeichner* | "`(`" *Ausdruck* "`)`"
| *Ausdruck* ("`+`" | "`-`" | "`*`" | "`≤`") *Ausdruck*
| *Bezeichner Ausdruck* … *Ausdruck*
| "`if`" *Ausdruck* "`then`" *Ausdruck* "`else`" *Ausdruck*

a) Realisieren Sie eine abstrakte Syntax für S in SML.

b) Schreiben Sie eine Prozedur *compExp* : *int env* → *bool* → *exp* → *code*, die Ausdrücke gemäß einer Umgebung und einem Booleschen Modus übersetzt. Die Umgebung liefert für Prozedurbezeichner Adressen im Programmspeicher und für Argumentbezeichner Indizes. Wenn der Modus *true* ist, liefert der Ausdruck das Ergebnis einer Prozedur und soll daher in Code übersetzt werden, der den Befehl *return* beinhaltet.

c) Schreiben Sie eine Prozedur *compDec* : *ca env* → *ca* → *dec list* → *ca env* ∗ *code*, die die Deklarationen eines Programms übersetzt.

d) Schreiben Sie eine Prozedur *compile*, die Programme übersetzt.

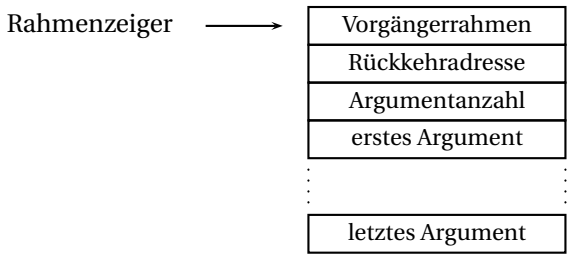
16.8 Aufrufrahmen

Die Realisierung der Prozedurbefehle erfordert einige clevere Ideen. Zunächst stellen wir fest, dass die Ausführung des Aufrufbefehls beendet sein muss, bevor der Befehlsinterpreter, dem Inhalt des Programmzählers folgend, mit der Ausführung des ersten Befehls des Prozedurrumpfs beginnt. Eine Schwierigkeit besteht nun darin, dass bei der Ausführung eines Rückkehrbefehls die Befehlsadresse im Zustand der Maschine gefunden werden muss, mit der nach der Beendigung des aktuellen Prozeduraufrufs fortgefahren werden soll (die sogenannte **Rückkehradresse**). Der Aufrufbefehl muss also im Zustand der Maschine die Informationen hinterlassen, die für die Realisierung des Rückkehrbefehls erforderlich sind. Da sich zu einem Zeitpunkt mehrere Aufrufe derselben Prozedur in der Ausführung befinden können (Rekursion), müssen die benötigten Informationen pro Aufruf dargestellt werden (also eventuell mehrfach pro Prozedur).

Die Lösung dieser Problematik besteht darin, dass man die sich in der Ausführung befindlichen Prozeduraufrufe auf dem Stapel darstellt, und zwar durch Blöcke, die als **Aufrufrahmen** bezeichnet werden.

- Der Aufrufbefehl *call k* initiiert die Ausführung eines Prozeduraufrufs, indem er einen neuen Aufrufrahmen auf den Stapel legt. Außerdem setzt er den Programmzähler auf $k+1$, damit die Maschine beim nächsten Schritt mit der Ausführung der Befehlssequenz der Prozedur beginnt.
- Der Rückkehrbefehl *return* entfernt den aktuellen Aufrufrahmen aus dem Stapel. Außerdem sorgt er dafür, dass die Argumente des zu beendenden Aufrufs im Stapel durch das Ergebnis des Aufrufs ersetzt werden. Schließlich setzt *return* den Programmzähler auf die im gerade entfernten Aufrufrahmen abgelegte Rückkehradresse, damit die Maschine beim nächsten Schritt den Befehl ausführt, der auf den Aufrufbefehl folgt, der den gerade beendeten Aufruf initiiert hat.
- Der Argumentbefehl *arg i* liefert das i-te Argument des aktuellen Prozeduraufrufs. Die Argumente eines Prozeduraufrufs befinden sich in dem den Prozeduraufruf darstellenden Aufrufrahmen.

Ein Aufrufrahmen wird im Stapel gemäß dem folgenden Schema abgelegt:



Der **Rahmenzeiger** ist eine zusätzliche, dem Stapel zugeordnete Zelle, in der die Adresse der obersten Zelle des aktuellen Aufrufrahmens steht. Der Rahmenzeiger ist notwendig, damit die Befehle *arg* und *return* realisiert werden können. Neben dem Rahmenzeiger gibt es noch den **Stapelzeiger**, der die Adresse der obersten allozierten Stapelzelle enthält. Mithilfe des Rahmen- und des Stapelzeigers kann der Aufrufbefehl *call k* wie folgt realisiert werden:

1. Lege die Anzahl der Argumente der Prozedur auf den Stapel. Diese findet sich im Programmspeicher unter der Adresse k in Form eines Befehls $proc(n,l)$.
2. Lege den um eins erhöhten Wert des Programmzählers auf den Stapel (d.h. die Rückkehradresse).
3. Lege den Wert des Rahmenzeigers auf den Stapel (d.h. die Adresse des Vorgängerrahmens).
4. Setze den Rahmenzeiger auf den Wert des Stapelzeigers.
5. Setze den Programmzähler auf $k+1$.

Die Realisierung des Rückkehrbefehls *return* kann wie folgt geschehen:

1. Schreibe das Ergebnis des Aufrufs (den Wert in der obersten Stapelzelle) in die Zelle, in der bisher das letzte Argument des Aufrufes stand.
2. Setze den Stapelzeiger auf die Adresse der Zelle, in der bisher das letzte Argument stand.
3. Setze den Programmzähler auf die Rückkehradresse.
4. Setze den Rahmenzeiger auf die Adresse des Vorgängerrahmens (steht in der Zelle, auf die der Rahmenzeiger gerade zeigt).

Abbildung 16.5 zeigt einen Zustand der Maschine, in dem sich drei Aufrufe der rekursiven Prozedur *power* aus § 16.7 in der Ausführung befinden. Gemäß dem Zustand der Maschine wird als Nächstes der Rückkehrbefehl an der Adresse 6 ausgeführt. Dieser beendet den im Stapel zuoberst dargestellten Aufruf $power(3,0)$.

Beachten Sie, dass die Maschine M allgemeine Rekursion mithilfe von Endrekursion realisiert. Mit der Sprache M können Prozeduren mit allgemeiner Rekursion beschrieben werden. Die ausführende Maschine kann jedoch problemlos ohne Rekursion nur mit Schleifen implementiert werden. Möglich wird dies dadurch, dass die Rekursion der in M beschriebenen Prozeduren im Stapel mit Aufrufrahmen realisiert wird.

Programmzähler 6

Programmspeicher

0	*proc*(2, 15)
1	*con 0*
2	*arg 2*
3	*leq*
4	*cbranch 3*
5	*con 1*
6	*return*
7	*con 1*
8	*arg 2*
9	*sub*
10	*arg 1*
11	*call 0*
12	*arg 1*
13	*mul*
14	*return*
15	*con 2*
16	*con 3*
17	*call 0*
18	*halt*

Stapelzeiger 15
Rahmenzeiger 14

Stapel

15	1	Ergebnis	
14	9	Vorgängerrahmen	*power 3 0*
13	12	Rückkehradresse	
12	2	Argumentanzahl	
11	3	Argument 1	
10	0	Argument 2	
9	4	Vorgängerrahmen	*power 3 1*
8	12	Rückkehradresse	
7	2	Argumentanzahl	
6	3	Argument 1	
5	1	Argument 2	
4	-1	Vorgängerrahmen	*power 3 2*
3	18	Rückkehradresse	
2	2	Argumentanzahl	
1	3	Argument 1	
0	2	Argument 2	

Halde leer

Abbildung 16.5: *Ein Maschinenzustand mit drei Aufrufrahmen*

Aufgabe 16.28 Simulieren Sie die Ausführung des in Abbildung 16.5 gezeigten Maschinenprogramms mit Papier und Bleistift.

a) Geben Sie die Folge der Adressen an, die der Programmzähler durchläuft.

b) Stellen Sie den Maschinenzustand dar, der vorliegt, wenn der Befehl an der Adresse 1 erstmals ausgeführt wird.

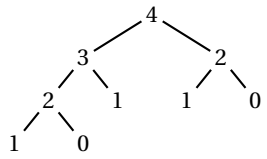
c) Stellen Sie den Maschinenzustand dar, der vorliegt, wenn der Befehl an der Adresse 12 erstmals ausgeführt wird.

Aufgabe 16.29 Geben Sie die Adresse des i-ten Arguments des aktuellen Aufrufs im Stapel an, wenn der Rahmenzeiger den Wert r hat.

Aufgabe 16.30 Warum wird für die Realisierung des Rückkehrbefehls die im Aufrufrahmen abgelegte Argumentanzahl benötigt?

16.9 Endaufrufe

Es ist aufschlussreich, sich für rekursive Prozeduren den Zusammenhang zwischen Rekursionsbäumen und den auf dem Stapel von M allozierten Aufrufrahmen zu verdeutlichen. Betrachten Sie dazu den Rekursionsbaum für die Fibonacci-Prozedur *fib* (Aufgabe 16.25, S. 340) und das Argument 4:



Der Verlauf der Ausführung des Aufrufs *fib 4* auf M entspricht einer Standardtour durch den Rekursionsbaum. Für jeden Knoten des Baums wird beim ersten Besuch ein Aufrufrahmen alloziert und beim letzten Besuch wieder dealloziert. Die Allokation der Rahmen erfolgt also gemäß der Präordnung, und die Deallokation gemäß der Postordnung. Wenn der Rahmen für einen Knoten v alloziert oder dealloziert wird, sind ansonsten nur die Rahmen für die v übergeordneten Knoten alloziert. Das bedeutet, dass für die Ausführung des Aufrufs *fib 4* maximal 4 Rahmen gleichzeitig alloziert sind (Rekursionstiefe plus 1). Allgemein ergibt sich, dass der Platzbedarf der für die Ausführung eines Prozeduraufrufs erforderlichen Rahmen auf dem Stapel linear mit der Rekursionstiefe des Aufrufs wächst.

Die Anzahl der gleichzeitig auf dem Stapel liegenden Rahmen kann durch den Einsatz des **Endaufrufbefehls** *callR* verringert werden. Insbesondere können endrekursive Prozeduren mit *callR* so übersetzt werden, dass bei ihrer Ausführung zu jedem Zeitpunkt immer nur ein Rahmen auf dem Stapel erforderlich ist. Ein Endaufrufbefehl *callR p* ersetzt einen Aufrufbefehl *call p* und einen unmittelbar folgenden Rückkehrbefehl *return*:

$$\mathit{call}\ p,\ \mathit{return} \quad \rightsquigarrow \quad \mathit{callR}\ p$$

Um die Funktionsweise des Endaufrufbefehls zu erklären, betrachten wir die Ausführung der Befehlssequenz *call p, return*. Den für die Ausführung der Befehlssequenz maßgeblichen Aufrufrahmen bezeichnen wir mit A (der Rückkehrbefehl ist nur sinnvoll, wenn gerade ein Prozeduraufruf ausgeführt wird). Die Ausführung des Aufrufbefehls *call p* alloziert einen weiteren Rahmen B auf dem Stapel. Nach der Deallokation von B wird als Nächstes der Rückkehrbefehl *return* ausgeführt, was zur Deallokation von A führt. Im Prinzip wäre es daher möglich, den Rahmen A bereits vor der Allokation von B vom Stapel zu nehmen und dafür die Rücksprungadresse von A als Rücksprungadresse für B zu verwenden. Genau das macht der Endaufrufbefehl *callR p*. Damit wird ein unnötiges Glied der Rücksprungkette eingespart.

Als Beispiel betrachten wir die endrekursive Prozedur

```
fun gcd(x,y) = if x<=y then if y<=x then x else gcd(x, y-x)
               else gcd(x-y, y)
```
*val gcd : int * int → int*

Wir können *gcd* wie folgt nach M übersetzen:

```
[proc(2,21),                                (* fun gcd(x,y) =  *)
 arg 2, arg 1, leq, cbranch 12,             (* if x<=y         *)
 arg 1, arg 2, leq, cbranch 3,              (* then if y<=x    *)
 arg 1, return,                             (* then x          *)
 arg 1, arg 2, sub, arg 1, callR 0,         (* else gcd(x,y-x) *)
 arg 2, arg 2, arg 1, sub, callR 0]         (* else gcd(x-y,y) *)
```

Damit haben wir eine Übersetzung der Prozedur *gcd*, deren Ausführung unabhängig von der Rekursionstiefe immer nur einen Rahmen auf dem Stapel benötigt.

Eine Prozeduranwendung im Rumpf einer Prozedurdeklaration steht in **Endposition**, wenn die Ausführung der Anwendung das Ergebnis der Prozedur liefert. Beispielsweise stehen die zwei rekursiven Anwendungen in der obigen Deklaration der Prozedur *gcd* beide in Endposition. Jede Prozeduranwendung in Endposition kann mit dem Endaufrufbefehl übersetzt werden, unabhängig davon, ob es sich um eine rekursive oder eine nichtrekursive Anwendung handelt. Eine Prozedurdeklaration beschreibt genau dann eine endrekursive Prozedur, wenn alle rekursiven Anwendungen der Prozedur in Endposition stehen.

Aufgabe 16.31 Markieren Sie in den folgenden Prozedurdeklarationen alle Prozeduranwendungen in Endposition:

```
fun f(x,y) = if x<y then y else f(x+1,y)

fun g x = if x<0 then x else 1+f(x,2*x)

fun h (1::xs) = h xs
  | h xs      = p xs
and p (2::xs) = h xs
  | p (x::xs) = g x + 5
```

Aufgabe 16.32 Stellen Sie den Zustand der Maschine M dar, der bei der Ausführung des Programms *gcdc@*[*con 16, con 12, call 0, halt*] vorliegt, nachdem der Endaufrufbefehl zum zweiten Mal ausgeführt wurde. Dabei bezeichnet *gcdc* den oben gezeigten Code für die Prozedur *gcd*. Orientieren Sie sich an der Darstellung in Abbildung 16.5 auf S. 343.

Aufgabe 16.33 Formulieren Sie die Prozeduren

```
fun fac'(a,n) = if n<=1 then a else fac'(a*n,n-1)
fun fac n = fac'(1,n)
```

in M. Achten Sie darauf, dass Sie nur Endaufrufbefehle verwenden.

Aufgabe 16.34 Formulieren Sie die verschränkt rekursiven Prozeduren

```
fun even n = n=0 orelse odd(n-1)
and odd n = n>=1 andalso even(n-1)
```

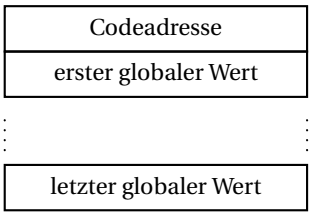
in M. Achten Sie darauf, dass Sie nur Endaufrufbefehle verwenden.

Aufgabe 16.35 (Übersetzer für S) Erweitern Sie den Übersetzer für S aus Aufgabe 16.27 auf S. 340 so, dass er Prozeduranwendungen in Endposition mit Endaufrufen realisiert.

16.10 Dynamische Prozeduren

In M werden Prozeduren durch Befehlssequenzen beschrieben. Der Aufrufbefehl identifiziert die aufzurufende Prozedur durch die Anfangsadresse ihrer Befehlssequenz im Programmspeicher. Da die Befehlssequenzen im Programmspeicher während der Programmausführung nicht geändert werden können, kann ein Programm nur mit Prozeduren rechnen, die statisch im Programm beschrieben sind. Wir sprechen daher von **statischen Prozeduren**.

Sprachen wie Standard ML verfügen über ein allgemeineres Prozedurmodell, bei dem Prozeduren durch ihren Code (statisch) und zusätzlich durch eine Umgebung (dynamisch) für die freien Bezeichner des Codes beschrieben werden (§ 2.5 und § 12.5). Solche **dynamischen Prozeduren** können in der Halde durch sogenannte **Prozedurblöcke** dargestellt werden. Diese enthalten neben der Anfangsadresse



der Befehlssequenz der Prozedur im Programmspeicher (die sogenannte **Codeadresse**) die Werte, die die Umgebung der Prozedur für die freien Bezeichner des Codes bereitstellt (die sogenannten **globalen Werte**).

Um in M mit dynamischen Prozeduren arbeiten zu können, benötigen wir spezielle Befehle. Die Befehlssequenz einer dynamischen Prozedur leiten wir mit dem Befehl

$dproc(m,n,k)$

ein, wobei m die Anzahl der globalen Werte, n die Anzahl der Argumente und k die Länge der Befehlssequenz angibt. Bei der Ausführung nimmt der Befehl die globalen Werte vom Stapel, alloziert einen entsprechenden Prozedurblock in der Halde, legt die Adresse des Prozedurblocks auf den Stapel und erhöht den Programmzähler um k. Der Aufruf

```
(fn (x,y) => fn f => f x - y) (3,4)
```

```
[con 4, con 3,
 dproc(0,2,11),                          (*  fn (x,y) =>   *)
 arg 2, arg 1, dproc(2,1,7),             (*  fn f =>       *)
 glob 2, glob 1, arg 1, dcall, sub,      (*  f x - y       *)
 return, return,
 dcall]
```

Abbildung 16.6: *Übersetzung einer kaskadierten Abstraktion*

einer dynamischen Prozedur erfolgt mit dem Befehl *dcall*, der die Adresse des Prozedurblocks über den Argumenten des Aufrufs im Stapel erwartet. Mit dem Befehl *glob i* kann die Befehlssequenz einer Prozedur auf den i-ten globalen Wert der Prozedur zugreifen (Nummerierung der globalen Werte beginnt wie bei den Argumenten mit 1). Abbildung 16.6 zeigt die Übersetzung eines Ausdrucks, dessen Auswertung eine dynamische Prozedur liefert.

Aufgabe 16.36 Übersetzen Sie die folgenden Deklarationen nach M mit dynamischen Prozeduren. Realisieren Sie Endaufrufe mit dem Endaufrufbefehl *dcallR*, der *dcall* mit *return* kombiniert.

a) `fun iter n s f = if n<=0 then s else iter (n-1) (f s) f`

b) `fun foldl f s nil = s | foldl f s (x::xr) = foldl f (f(x,s)) xr`

c) `val rev = foldl op:: nil`

Nehmen Sie bei der Deklaration von *rev* an, dass Sie die Adresse des Prozedurblocks für *foldl* mit dem Befehl *getS 0* erhalten können.

Aufgabe 16.37 Machen Sie sich klar, dass die Rahmen für die Aufrufe von dynamischen Prozeduren bis auf eine Änderung genauso aussehen können wie die für statische Prozeduren: Statt der für statische Prozeduren erforderlichen Argumentanzahl kann für dynamische Prozeduren die Adresse des Prozedurblocks im Rahmen abgelegt werden. Über den Prozedurblock kann dann auf die Argumentanzahl und die globalen Werte zugegriffen werden.

16.11 Automatische Speicherbereinigung

So wie wir M bisher beschrieben haben, gibt es keine Möglichkeit, einmal in der Halde allozierte Blöcke wieder freizugeben. Oft aber werden Blöcke nach ihrer Allokation nur für kurze Zeit benötigt (siehe § 15.13). Wir wollen jetzt ein automatisches Speicherbereinigungsverfahren skizzieren, das nicht mehr zugängliche Blöcke erkennt und den von ihnen belegten Speicherplatz für eine Wiederverwendung freigibt.

Zunächst müssen wir die Semantik von M an zwei kritischen Punkten ändern, um die Voraussetzungen für eine automatische Speicherbereinigung zu schaffen:

1. Die Adressen der Zellen der Halde müssen sich von den entsprechenden Zahlen unterscheiden. Beispielsweise darf 0 nicht gleichzeitig die Adresse 0 und die Zahl 0 darstellen. Da Adressen und Zahlen beide durch Werte von *int* dargestellt werden, bleibt uns nichts anderes übrig, als den verfügbaren Zahlenbereich einzuschränken (z.B. auf $\{-2^{29},\ldots,2^{29}-1\}$, siehe auch § 1.13.1) und die restlichen Werte von *int* für die Darstellung von Adressen zu verwenden.
2. Die in der Halde allozierten Blöcke müssen mit ihrer Länge markiert sein. Dazu statten wir jeden Block bei der Allokation mit einer zusätzlichen, nach außen nicht sichtbaren Zelle aus, in der die Länge des Blocks abgelegt ist.

Ein Block in der Halde heißt **erreichbar**, wenn seine Adresse auf dem Stapel liegt oder wenn seine Adresse in einer Zelle eines erreichbaren Blocks steht. Da wir Adressen von Zahlen unterscheiden können und da Blöcke aufgrund der internen Längenangabe ihre Zellen kennen, sind wir in der Lage, die erreichbaren Blöcke zu finden und die nicht erreichbaren Blöcke bei Bedarf neu zu allozieren.

In der Praxis arbeitet man meist mit kopierenden Verfahren, die über zwei Halden verfügen. Wenn der Platz in der einen Halde erschöpft ist, werden alle erreichbaren Blöcke dicht in die andere, bisher leere Halde kopiert. Danach hat man wieder eine leere Halde. Zudem hat man eine dicht belegte Halde, in der die freien Speicherzellen am Stück verfügbar sind.

Damit eine Programmiersprache mit einer automatischen Speicherbereinigung realisiert werden kann, muss sichergestellt sein, dass der Programmierer Adressen nicht durch Zahlen darstellen kann. Damit M diese Voraussetzung erfüllt, müssen der Befehl *con* und die arithmetischen Operationen so eingeschränkt werden, dass sie keine Adressen auf den Stapel legen können (siehe Aufgabe 16.18 auf S. 335).

C und C++ sind Beispiele für Sprachen, die nicht mit einer automatischen Speicherbereinigung realisiert werden können, da sie es dem Programmierer erlauben, Adressen durch Zahlen darzustellen. Dagegen handelt es sich bei Standard ML und Java um Sprachen, die grundsätzlich mit einer automatischen Speicherbereinigung realisiert werden.

Aufgabe 16.38 Würden Sie die nach außen nicht sichtbare Zelle für die Länge eines Blocks an den Anfang oder an das Ende des Blocks legen? Schauen Sie sich dazu das Aufrufrahmenformat in § 16.8 an und beachten Sie den Zusammenhang zwischen Argumentanzahl und Länge.

Aufgabe 16.39 Betrachten Sie die Prozeduren *putlist* und *getlist* in § 15.10. Warum kann *getlist* die von *putlist* in der Halde dargestellte Liste rekonstruieren, ohne dass dabei die Voraussetzungen für eine automatische Speicherbereinigung gegeben sind (Adressen von Zahlen unterscheidbar, Blöcke kennen ihre Länge)?

16.12 Voll- und Halbübersetzung

Bei der Realisierung von Programmiersprachen unterscheidet man zwischen zwei Ansätzen, die als **Vollübersetzung** und **Halbübersetzung** bezeichnet werden. Bei der Vollübersetzung werden die Programme der Sprache in die Maschinensprache des ausführenden Computers übersetzt. Bei der Halbübersetzung werden die Programme in eine lineare Zwischensprache übersetzt, die wir uns ähnlich wie M vorstellen können. Die Programme der Zwischensprachen werden dann durch eine Maschine ausgeführt, die mit einer bereits verfügbaren Programmiersprache (typischerweise C++) realisiert ist. Der Übersetzer in die lineare Zwischensprache kann in einer beliebigen, bereits realisierten Sprache programmiert werden.

In der Praxis schreibt man einen Übersetzer Ü für eine Programmiersprache S meistens in S. Damit Ü auch benutzt werden kann, benötigt man allerdings eine bereits existierende Realisierung R von S. Mit R kann man Ü ausführen. Damit ist man in der Lage, Ü mit Ü zu übersetzen. Danach hat man eine eigenständige Realisierung von S, die von R unabhängig ist. Der Übersetzer Ü kann dann schrittweise verbessert werden, ohne dass R nochmals erforderlich wäre. Dieses Vorgehen wird als **Bootstrapping** bezeichnet. Die Effizienz von R spielt für die Effizienz von Ü keine Rolle, da R nur für das Bootstrapping benötigt wird.

Bemerkungen

In diesem Kapitel haben wir zwei maschinennahe Sprachen W und M betrachtet. Die dynamische Semantik von M haben wir durch eine Maschine beschrieben, die mit einem Stapel und einer Halde arbeitet. Beiden Sprachen ist gemeinsam, dass sie nur über Werte des Typs *int* verfügen. Zusammengesetzte Werte wie Listen und Bäume können durch verzeigerte Blöcke in der Halde dargestellt werden. In M sind Programme Folgen von primitiven Befehlen, die nicht weiter geschachtelt sind und die ohne Bezeichner auskommen. Dagegen verfügt W über Bezeichner und eine rekursive Programmstruktur mit arithmetischen Ausdrücken, Schleifen und Konditionalen. Für W haben wir einen Übersetzer nach M angegeben, der die rekursive Programmstruktur von W auf die lineare Programmstruktur von M reduziert. Dabei werden Bezeichner durch Adressen realisiert, Konditionale und Schleifen durch Sprungbefehle.

M verfügt über Befehle für die Formulierung von Prozeduren. Bei der Ausführung werden Prozeduraufrufe durch Rahmen im Stapel dargestellt. Ein Rahmen enthält die Argumente und die Rücksprungadresse des Aufrufs. Damit haben wir das klassische Ausführungsmodell für Programmiersprachen mit rekursiven Prozeduren kennengelernt. Dabei ist bemerkenswert, dass das Ausführungsmodell für die Sprache M wegen der linearen Programmstruktur ohne Rekursion nur mithilfe von Schleifen formuliert werden kann.

Das Ausführungsmodell für Prozeduren hat die Eigenschaft, dass der Platzbedarf für

die Rahmen im Stapel linear mit der Rekursionstiefe eines Aufrufs wächst. Endrekursive Prozeduren können jedoch mithilfe des Endaufrufbefehls so formuliert werden, dass ihre Ausführung im Stapel nur jeweils einen Rahmen benötigt.

M realisiert statische Prozeduren, die sich vollständig durch eine fixe Befehlssequenz im Programmspeicher darstellen lassen. Für Sprachen wie Standard ML benötigt man jedoch dynamische Prozeduren, die durch Blöcke in der Halde und durch eine Befehlssequenz im Programmspeicher dargestellt werden. Durch zusätzliche Befehle kann M um dynamische Prozeduren erweitert werden.

Im Zusammenhang mit M sind wir dem Konzept der automatischen Speicherbereinigung begegnet. Eine automatische Speicherbereinigung ist nur für solche Sprachen möglich, in denen der Programmierer Adressen nicht durch Zahlen darstellen kann.

Ein Übersetzer für eine Programmiersprache ist ein komplexes Programm, für dessen Verständnis fast alle Konzepte notwendig sind, die Sie in diesem Buch kennengelernt haben. Zunächst muss die Zeichendarstellung des Programms in die Baumdarstellung übersetzt werden (Lexing und Parsing, Kapitel 13). Dann muss die Baumdarstellung gemäß der Regeln der statischen Semantik geprüft werden (Elaborierung, Kapitel 12). Schließlich muss die Baumdarstellung in eine Befehlsfolge für die Zielmaschine übersetzt werden. Dabei ist bemerkenswert, dass ein Übersetzer mit einer linearen Darstellung beginnt (dem Programmtext), daraus eine Baumdarstellung gewinnt, und die Baumdarstellung wieder in eine lineare Darstellung übersetzt (den Code für die Zielmaschine). Aus Sicht der Baumdarstellung ähnelt die Wortdarstellung eines Programms der Prälinearisierung (Aufgabe 13.7, S. 266), und der Code eines Programms der Postlinearisierung der Baumdarstellung (Aufgabe 16.6, S. 330).

Über M haben Sie einen ersten Eindruck von der Software-Hardware-Schnittstelle von Computern bekommen. Wenn Sie mehr über dieses Thema wissen wollen, empfehle ich Ihnen den Klassiker von Patterson und Hennessy [4].

Verzeichnis

Stapelmaschine M; Befehlsinterpreter, Programmspeicher, Stapel und Halde; Programmzähler; Stapelzeiger; Befehle und Code; Maschinenzustände.

Arithmetische Befehle; Testbefehl; bedingter und unbedingter Sprungbefehl; Vorwärts- und Rückwärtssprünge.

Prozedurbefehle: Aufrufbefehl, Argumentbefehl, Rückkehrbefehl; stellige Prozeduren.

Aufrufrahmen; Rückkehradresse, Vorgängerrahmen, Rahmenzeiger.

Endaufrufe, Prozeduranwendungen in Endposition, Endaufrufbefehl.

Statische und dynamische Prozeduren; Prozedurblöcke mit Codeadresse und globalen Werten.

Automatische Speicherbereinigung; erreichbare Blöcke.

Sprache W: imperative Variablen, Schleifen, Anweisungen, Eingabe und Ausgabe.

Übersetzer für W; Vollübersetzung, Halbübersetzung, Bootstrapping.

Baumrekonstruktion aus der Postlinearisierung.

A Klammersparregeln

Damit man nicht so viele Klammern schreiben muss, legen Programmiersprachen für ihre Infixoperatoren eine Klammerhierarchie fest. Bei **Infixoperatoren** handelt es sich um zweistellige Operatoren, die zwischen ihre Argumente geschrieben werden (z.B. $x + y$). Hier ist die Klammerhierarchie für die Infixoperatoren von Standard ML:

:=	*o*					*links*
=	<>	<	<=	>	>=	*links*
::	@					*rechts*
+	−	^				*links*
*	/	*div*	*mod*			*links*

Operatoren, die in der Hierarchie weiter unten stehen, klammern vor Operatoren, die in der Hierarchie weiter oben stehen (z.B. "*" vor "+"). Zusätzlich zu den durch die Hierarchie beschriebenen Vorrangregeln legt Standard ML fest, dass die Listenoperatoren "::" und "@" rechts klammern (z.B. $x :: (y :: zs)$), und dass alle anderen Operatoren links klammern (z.B. $(x + y) + z$). Hier sind Beispiele:

$$\begin{aligned} 3 * x - x \; div \; 2 + x \; mod \; 2 &\rightsquigarrow ((3 * x) - (x \; div \; 2)) + (x \; mod \; 2) \\ a + 2 < x - y - z &\rightsquigarrow (a + 2) < ((x - y) - z) \\ xs @ x + 2 :: ys &\rightsquigarrow xs @ ((x + 2) :: ys) \end{aligned}$$

Die Schlüsselwörter *handle*, *orelse* und *andalso* klammern nach den Operatoren, wobei *handle* nach *orelse* und *orelse* nach *andalso* klammert. Zum Beispiel:

$$\begin{aligned} x < 2 \; orelse \; y > 2 \; andalso \; f \; x &\rightsquigarrow (x < 2) \; orelse \; ((y > 2) \; andalso \; (f \; x)) \\ f \; x \; orelse \; g \; y \; handle \; E \Rightarrow true &\rightsquigarrow ((f \; x) \; orelse \; (g \; y)) \; handle \; E \Rightarrow true \end{aligned}$$

Für Prozeduranwendungen gibt es zwei Klammersparregeln:

$f \; 3 + 4$	$\rightsquigarrow$	$(f \; 3) + 4$	Prozeduranwendung vor Operatoranwendung
$f \; g \; 3$	$\rightsquigarrow$	$(f \; g) \; 3$	Prozeduranwendung klammert links

Schließlich gibt es noch zwei Klammersparregeln für Typen:

$int * int \rightarrow int * int$	$\rightsquigarrow$	$(int * int) \rightarrow (int * int)$	Stern vor Pfeil
$int \rightarrow int \rightarrow int$	$\rightsquigarrow$	$int \rightarrow (int \rightarrow int)$	Pfeil klammert rechts

Literaturverzeichnis

[1] Cormen, Thomas H. / Leiserson, Charles E. / Rivest, Ronald L.: *Algorithmen – eine Einführung*, Oldenbourg Wissenschaftsverlag, 2007.

[2] Graham, Ronald L. / Knuth, Donald E. / Patashnik, Oren: *Concrete Mathematics, A Foundation for Computer Science*, Addison-Wesley, 1994.

[3] Milner, Robin / Tofte, Mads / Harper, Robert / MacQueen, David: *The Definition of Standard ML (Revised)*, The MIT Press, 1997.

[4] Patterson, David A. / Hennessy, John L.: *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2004.

[5] Pierce, Benjamin C.: *Types and Programming Languages*, The MIT Press, 2002.

[6] Rosen, Kenneth H.: *Discrete Mathematics and Its Applications*, McGraw-Hill, 2006.

Index