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:

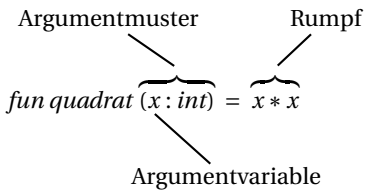
```
quadrat 2 + quadrat 3
```
13 : int

```
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* : $int \to int$, 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*

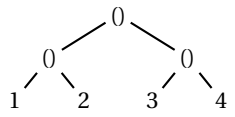
```
(7+2, 2*7)
```
*(9, 14) : int * int*

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
```

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

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

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 $O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(b^n)$.

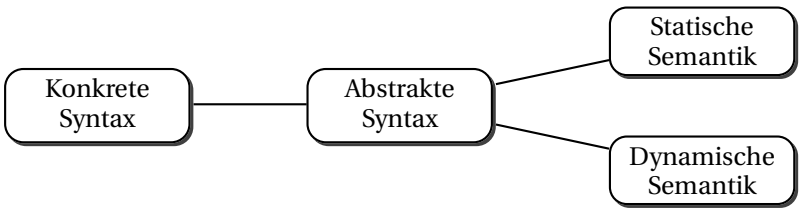
Rekurrenzsätze.

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:

$$T \in TE = Id \xrightarrow{\text{fin}} Ty$$

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.

Index