Grundlagen funktionaler Programmierung

von
Dr. Martin Erwig,
FernUniversität Hagen

R. Oldenbourg Verlag München Wien 1999

Dr. rer. nat. Martin Erwig:
Geboren 1963 in Marl. Studium der Informatik mit Nebenfach Betriebswirtschaftslehre an der Universität Dortmund (1984 – 1989). Wiss. Mitarbeiter an der FernUniversität Hagen (1989 – 1994). Promotion 1994 bei R. H. Güting. Seit 1994 Hochschulassistent an der FernUniversität Hagen.

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Erwig, Martin:
Grundlagen funktionaler Programmierung / von Erwig Martin. -
München ; Wien : Oldenbourg, 1999
ISBN 3-486-25100-7

© 1999 R. Oldenbourg Verlag
Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0, Internet: http://www.oldenbourg.de

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Lektorat: Margarete Metzger
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Druckhaus „Thomas Müntzer“ GmbH, Bad Langensalza

Für Anja und Alexander

Vorwort

Dieses Buch vermittelt

- grundlegende Konzepte,
- fortgeschrittene Programmiertechniken,
- formale Grundlagen sowie
- Implementierungsaspekte

moderner funktionaler Programmiersprachen. Die Bezeichnung *funktionale Programmierung* weist auf die intensive Verwendung von Funktionen als Programmierkonstrukte hin. Warum überhaupt bedient man sich Funktionen beim Programmieren?

Funktionen als Grundlage der Programmierung

Funktionen werden schon seit langem in der Mathematik sehr erfolgreich eingesetzt; sie sind relativ einfach zu begreifende Objekte und finden gleichzeitig in sehr vielen Bereichen Anwendung. Insbesondere bieten Funktionen ein anschauliches Konzept von Berechnungen und damit auch von Programmen. Die funktionale Programmierung bezieht einen Großteil ihrer Attraktivität aus der Tatsache, daß der zugrundeliegende Funktionenbegriff sehr eng an den aus der Mathematik angelehnt ist.

Funktionen in der Mathematik

Eine Funktion $f : A \to B$ ist durch ihren *Definitionsbereich* A, ihren Wertebereich B sowie eine *Abbildungsvorschrift* gegeben. f ist das *Funktionssymbol* (oder der Name) der Funktion. Die Abbildungsvorschrift kann durch Aufzählung von Wertepaaren aus $A \times B$ oder aber durch eine Funktionsgleichung gegeben sein. Eine solche Funktionsgleichung führt auf der linken Seite eine oder mehrere *Variablen* ein, die auf der rechten Seite benutzt werden und für Werte aus dem Definitionsbereich stehen. Die rechte Seite der Funktionsgleichung ist durch einen *Ausdruck* gegeben, in dem *Konstanten* und Funktionen verwandt werden.

Als Beispiel betrachten wir die durch eine Gleichung definierte Funktion zur Berechnung des Kreisinhalts bei gegebenem Radius.

$$K(r) = \pi * r^2$$

Definitions- und Wertebereich von K sind die reellen Zahlen, das heißt $K : \mathbb{R} \rightarrow \mathbb{R}$. Der definierende Ausdruck der Funktionsgleichung enthält die Konstante π und verwendet die Multiplikations- und Quadratfunktion auf den reellen Zahlen.

Die Funktion K läßt sich in einer funktionalen Sprache genauso definieren und benutzen, wie man es aus der Mathematik gewohnt ist. Im Unterschied zu LISP oder Scheme zeichnen sich neuere funktionale Sprachen insbesondere durch sogenannt starke, polymorphe Typsysteme sowie (benutzerdefinierbare) Datentypen aus. Ein Typsystem verlangt, vereinfacht gesagt, daß jede Funktion einen genau festgelegten Definitions- und Wertebereich besitzen muß. Ein ganz wichtiger Aspekt von Datentypen ist die Möglichkeit, komplexe Wertebereiche zu formulieren und dafür Funktionsdefinitionen über pattern matching vorzunehmen.

Referentielle Transparenz

Eine ganz fundamentale Eigenschaft jeder mathematischen Notation ist, daß ein und dieselbe Variable an verschiedenen Stellen innerhalb ihres Geltungsbereiches immer den gleichen Wert bezeichnet. Dieses Prinzip der *referentiellen Transparenz* gilt in imperativen Programmiersprachen nicht, da ihre Semantik von einem impliziten Zustand abhängt. So erlaubt die Wertzuweisungsoperationen beispielsweise die folgende Programmzeile:

Wertzuweisung

```
i = i + 1
```

Mathematisch betrachtet stellt diese Gleichung eine nicht erfüllbare Aussage dar, das heißt, für die Gleichung gibt es keine Lösung. Imperative Sprachen halten dagegen sehr wohl eine Lösung bereit: Auf der linken Seite steht `i` z. B. für den Wert 5 und auf der rechten Seite für den Wert 4. Noch gravierendere Effekte ergeben sich aus der Verwendung von Seiteneffekten. Ist beispielsweise `f` eine Prozedur, die bei jedem Aufruf eine globale Variable um 1 erhöht und den Wert der Variable als Ergebnis liefert, so gilt

Seiteneffekte

$$\texttt{f} \neq \texttt{f}$$

In funktionalen Sprachen bestehen Programme grundsätzlich aus einer Reihe von Werte- und Funktionsdefinitionen; Zuweisungen an Variablen gibt es im allgemeinen nicht. Dadurch wird ein hohes Maß an referentieller Transparenz erreicht, wodurch funktionale Programme mathematischen Methoden

zugänglich werden: Man kann beispielsweise eine Variable in einem Ausdruck durch ihre Definition ersetzen, ohne die Bedeutung des Ausdrucks zu verändern. Dieses Prinzip nennt man auch *Substituierbarkeit*. Dies kann einerseits dazu dienen, Eigenschaften bestimmter Funktionen nachzuweisen, oder andererseits auch dazu, aus sehr einfachen Spezifikationen durch Transformationen effiziente Implementierungen zu gewinnen.

Substituierbarkeit

Inhalt des Buches

Im ersten Kapitel werden die Konzepte moderner funktionaler Sprachen (Funktionen, Typisierung, Polymorphismus, Datentypen und pattern matching, Funktionen höherer Ordnung) am Beispiel der Sprache ML vorgestellt. Eine Reihe von Autoren vertritt die Meinung, daß darüber hinaus auch verzögerte Auswertung ein grundlegendes Element funktionaler Programmierung ist und somit Bestandteil einer funktionalen Sprache sein sollte. Dementsprechend wird in einigen Sprachen (z. B. Miranda oder Haskell) jeder (Teil-) Ausdruck grundsätzlich nur dann ausgewertet, wenn er zur Berechnung des Gesamtergebnisses auch wirklich benötigt wird. Es gibt in der Tat Situationen, in denen verzögerte Auswertung sehr hilfreich ist und Vorteile gegenüber der *call-by-value* Parameterübergabe bietet, wie man sie von C oder Pascal kennt. Wir diskutieren diesen Aspekt im zweiten Kapitel und stellen dort eine Technik vor, mit der man verzögerte Auswertung von Ausdrücken in call-by-value Sprachen wie ML realisieren kann.

Das dritte Kapitel ist formalen Methoden gewidmet, mit denen man Eigenschaften von Programmen beweisen kann oder aber auch Programme transformieren kann (um z. B. deren Effizienz zu steigern).

In Kapitel 4 betrachten wir den Lambda-Kalkül, der ein formales Funktionenmodell zur Beschreibung aller rekursiven Funktionen darstellt. Mit der Formalisierung als Kalkül wird die automatische Verarbeitung von Funktionen ermöglicht. Den Lambda-Kalkül kann man sicherlich als *das* formale Modell für alle funktionalen Programmiersprachen bezeichnen.

Danach beschreiben wir im fünften Kapitel die Formalisierung von Typsystemen für funktionale Sprachen. Insbesondere geben wir einen Algorithmus an, der für ein Programm den jeweils allgemeinsten Typ automatisch zu ermitteln vermag.

Das letzte Kapitel schließlich vertieft das Verständnis funktionaler Sprachen durch die Untersuchung verschiedener Implementierungsaspekte. Dies umfaßt die Beschreibung eines Interpreters, einer speziellen Maschinenarchitektur und eines Compilers. Für Interpreter, Compiler und Maschine beschreiben wir auch die Realisierung in ML selbst.

Adressaten und Lernziele

Dieses Buch wendet sich primär an Informatik-Studenten und Programmierer, die bereits etwas Erfahrung im Programmieren haben, und die sich einen raschen, aber dennoch tiefergehenden Einblick in die funktionale Programmierung verschaffen wollen.

Ein Lernziel ist der Erwerb spezieller Techniken der funktionalen Programmierung (z. B. pattern matching, polymorphe Funktionen, Funktionen höherer Ordnung, verzögerte Auswertung). Diese werden an kleineren (in den Kapiteln 1 und 2) und auch größeren Beispielen (in Kapitel 6) erprobt, so daß der Leser nach Durcharbeiten des Buches in der Lage sein sollte, selbst komplizierte Programmieraufgaben durch konsequente Definition geeigneter Datentypen und Funktionen zu bewältigen.

Ein weiteres Lernziel bildet das Verständnis der formalen Grundlagen funktionaler Sprachen. Dies beinhaltet zum einen natürlich den Stoff der Kapitel 4 und 5, zum anderen aber vermittelt gerade die Implementierung einer Sprache Einsichten in innere Strukturen, die durch rein formale Definitionen zwar genau erfaßt werden können, oft aber nur schwer zu verstehen sind.

Schließlich soll demonstriert werden, daß wegen ihrer starken formalen Basis funktionale Sprachen einen einfachen Zugang mathematischer Methoden erlauben (z. B. zum Zwecke des Korrektheitsbeweises oder der optimierenden Transformationen). Damit soll dokumentiert werden, daß es in funktionalen Sprachen sehr viel einfacher ist, korrekte Software zu realisieren, als in imperativen Sprachen, ohne daß dies zu Effizienzeinbußen führen muß.

Weitere Informationen

Einige zusätzliche Informationen zu diesem Buch sind über das World-Wide Web zugänglich:

```
http://www.fernuni-hagen.de/inf/pi4/erwig/buch/fp/
```

Dort sind insbesondere online-Versionen aller im Buch abgebildeten Programme verfügbar. Einige Links bezüglich der im Kurs verwandten Programmiersprache ML sind unter anderem unter

```
http://www.dcs.ed.ac.uk/home/cdw/archive.html
http://cm.bell-labs.com/cm/cs/what/smlnj/links.html
```

zusammengetragen. Neben allgemeiner Information und Dokumentation sind dort auch Verweise auf verschiedene Implementierungen zu finden.

Alle abgebildeten Programme genügen dem aktuellen ML-Standard von 1997. In den Beispielen werden zuweilen die Antworten des ML-Interpreters auf die Eingaben gezeigt. Daran kann man insbesondere Details des Typsystems erklären und auch mögliche Programmierfehler besprechen. Wir verwenden hier das frei erhältliche System „Standard ML of New Jersey", in der Version 110. Dies ist unter anderem für Unix und Windows 95/Windows NT verfügbar:

```
http://cm.bell-labs.com/cm/cs/what/smlnj/
```

Danksagung

Grundlage für das vorliegende Buch sind Kursmaterialien zu einer Vorlesung, die wiederholt an der FernUniversität Hagen gehalten worden ist. Dabei haben eine Reihe von Studenten durch Hinweise auf Fehler zu Verbesserungen beigetragen. Außerordentliches Engagement haben dabei André Piotrowski und Thomas Salvador gezeigt, denen ich an dieser Stelle besonders dafür danken möchte.

Ungeachtet der vielen Überarbeitungen sind gewiß noch einige Fehler in diesem Buch verblieben. Für Hinweise auf solche Fehler wäre ich sehr dankbar.

Inhaltsverzeichnis

Vorwort **7**

1 Elemente funktionaler Programmierung **15**
1.1 Werte und Ausdrücke 15
1.2 Tupel, Records und Listen 21
1.3 Variablen und Definitionen 25
1.4 Funktionen 31
1.5 Typen und Polymorphismus 37
1.6 Datentypen und Pattern Matching 42
1.7 Funktionen höherer Ordnung 47
1.8 Literaturhinweise 52

2 Verzögerte Auswertung **53**
2.1 Parameterübergabe und Auswertungsstrategien 53
2.2 Unendliche Datenstrukturen 57
2.3 Literaturhinweise 59

3 Verifikation und Programm-Transformation **61**
3.1 Induktion 61
3.1.1 Korrektheit rekursiv definierter Funktionen 62
3.1.2 Strukturelle Induktion auf Datentypen 66
3.1.3 Wohlfundierte Relationen und wohlfundierte Induktion 71
3.2 Optimierung mit dem Unfold/Fold-Verfahren 75
3.3 Der Bird/Meertens-Formalismus 83
3.3.1 Beispiel: Das Auswahlproblem 83
3.3.2 Eigenschaften von `map` und `foldl` 86
3.3.3 Berechnung von Listensegmenten mittels `scanl` 88
3.3.4 Optimale Berechnung der maximalen Teilsumme 91
3.4 Literaturhinweise 93

4 Der Lambda-Kalkül **95**
4.1 Die Syntax des Lambda-Kalküls 95
4.2 Reduktion und Normalformen 99
4.3 Repräsentation von Datentypen 107

4.4 Rekursion 109
4.5 Ausdrucksstärke des Lambda-Kalküls 112
4.6 Literaturhinweise 115

5 Typisierung und Typinferenz 117
5.1 Einführung in Typsysteme 117
5.2 Ein polymorphes Typinferenzsystem 120
5.3 Automatische Typinferenz 128
5.4 Literaturhinweise 133

6 Implementierungstechniken 135
6.1 Implementierung von Programmiersprachen 135
6.2 Ein Interpreter für Mini-ML 137
6.2.1 Interne Repräsentation von Programmen 138
6.2.2 Die Interpreterfunktion `eval` 140
6.3 Ein Compiler für Mini-ML 147
6.3.1 Die SECD-Maschine 148
6.3.2 Übersetzung von Mini-ML in SECD-Code 153
6.3.3 Behandlung von Rekursion 155
6.3.4 Implementierung der SECD-Maschine in ML 159
6.4 Literaturhinweise 163

Lösungen zu den Aufgaben 165

Literatur 179

Index 185

1 Elemente funktionaler Programmierung

In diesem Kapitel beschreiben wir einige wesentliche Bestandteile moderner funktionaler Programmiersprachen am Beispiel der Sprache ML. Alle angegebenen Beispiele genügen dem ML-Standard von 1997 und sollten somit in jedem System, das diesen implementiert, nachvollziehbar sein.

Ein funktionales Programm besteht im wesentlichen aus einer Menge von *Definitionen D*, die dann in einem (u. U. sehr komplexen) *Ausdruck exp* verwendet werden. Der Ausdruck *exp* entspricht in etwa dem Hauptprogramm in einer imperativen Sprache, und die Definitionen *D* entsprechen einzelnen Prozedur- und Variablen-Deklarationen. Der „Abarbeitung" eines Programms oder einer Prozedur entspricht in der funktionalen Welt das *Auswerten* eines Ausdrucks zu einem *Wert*.

Korrespondenz imperatives ↔ funktionales Programm

Wir werden zunächst in Abschnitt 1.1 verschiedene Arten von einfachen Werten vorstellen. Wir zeigen, wie man Ausdrücke bildet und wie diese dann ausgewertet werden. Komplexe Werte kann man auf einfache Weise mittels Tupel- oder Listen-Konstruktoren erzeugen, die wir in Abschnitt 1.2 ansprechen. In Abschnitt 1.3 zeigen wir, wie man Definitionen vornimmt. Dabei gehen wir auch kurz auf den Begriff des statischen Bindens ein. Funktionen besprechen wir dann in Abschnitt 1.4, wobei einige Aspekte erst in den folgenden Abschnitten behandelt werden können. Das Typsystem von ML behandeln wir in Abschnitt 1.5; insbesondere werden wir dort die Idee des Typ-Polymorphismus erläutern. Datentypen und pattern matching folgen dann in Abschnitt 1.6, bevor wir schließlich in Abschnitt 1.7 Funktionen höherer Ordnung untersuchen.

1.1 Werte und Ausdrücke

Werte sind Ausdrücke, die nicht weiter ausgewertet werden können. Es gibt einfache Werte wie Zahlen oder Zeichenketten und konstruierte Werte wie Records oder Listen. Darüber hinaus stellen Funktionen ebenfalls Werte dar, allerdings mit der Besonderheit, daß sie auf andere Werte angewandt

(*appliziert*) werden können. Alle Werte in ML, also auch Funktionen, haben einen eindeutig bestimmten *Typ*. Werte, die keine Funktionen sind oder enthalten, nennt man auch *Konstanten*. Aus Werten und Funktionen können *Ausdrücke* geformt werden, die vom ML-Interpreter auf Werte reduziert werden. Diesen Vorgang nennt man das *Auswerten* von Ausdrücken.

Notation ganzer Zahlen

Ganze Zahlen. Ganze Zahlen (*integers*) werden durch Folgen von Ziffern dargestellt, wobei negativen Zahlen eine Tilde ~ (und nicht das Minuszeichen!) vorangestellt wird. Zahlenkonstanten werden wie alle übrigen Werte vom ML-Interpreter zu sich selbst ausgewertet, das heißt, sie werden unverändert als Ergebnis zurückgeliefert, z. B.:

```
5;
> val it = 5 : int
```

Die Antwort des ML-Systems enthält jeweils folgende Informationen:

(1) Art des Ergebnisses (hier: `val`, das heißt ein Wert)

(2) Variable, an die das Ergebnis gebunden ist. Dies ist beim Auswerten von Ausdrücken immer die Variable `it`.

(3) Gleichheitszeichen gefolgt von dem Ergebnis selbst (hier: die Zahl 5)

(4) Falls das Ergebnis ein Wert ist, ein Doppelpunkt gefolgt vom Typ des Wertes (hier: `int`)

Ausdrücke

Ein Ausdruck wird durch Applikation einer Funktion auf Werte oder Ausdrücke gebildet. Als Funktionen auf ganzen Zahlen stehen zunächst die binären Operationen `+`, `-`, `*`, `div` und `mod` zur Verfügung, wobei die ersten beiden eine niedrigere Präzedenz als die übrigen haben. Durch Verwendung von Klammern kann man die Auswertungsreihenfolge allerdings verändern. Alle Operationen sind vom Typ

```
int * int -> int
```

Funktionstypen

Das heißt, sie nehmen integer-Paare als Argumente und liefern integer-Werte als Ergebnis. Dabei bezeichnet der Pfeil (`->`) den Typ von Funktionen: Der Typ links vom Pfeil gibt den Typ der Argumente und der Typ rechts vom Pfeil gibt den der Ergebnisse an. `->` ist ein Beispiel für einen *Typkonstruktor*, das heißt, er kann eine ganze Reihe verschiedener Typen bezeichnen. Durch Anwendung auf zwei Typen (hier ein Argumenttyp `int * int` sowie ein Ergebnistyp `int`) wird dann ein konkreter Typ *konstruiert* (hier der Typ der Funktionen, die integer-Paare auf einzelne integer-Werte abbilden). In

diesem Beispiel ist nun der Argumenttyp wieder ein konstruierter Typ, nämlich der Typ aller integer-Paare. Der entsprechende Typkonstruktor `*` erinnert an die Bildung des kartesischen Produkts. Der Grund dafür, daß der obige Typausdruck nicht etwa als Typ von Paaren jeweils bestehend aus einem integer-Wert und einer unären integer-Funktion interpretiert wird, liegt wiederum an der höheren Präzedenz des `*`-Konstruktors gegenüber des `->`-Konstruktors.

Infix-Notation

Da die verwandten Funktionen alle in *Infix-Notation* verwandt werden, brauchen wir an dieser Stelle noch nicht auf die Frage einzugehen, wie man Paare von Werten erzeugt. Dies geschieht im Abschnitt 1.2.

Unäre Operationen auf ganzen Zahlen sind die Negationsfunktion `~`, die das Vorzeichen einer Zahl umdreht, sowie die Funktion `abs`, die den Absolutbetrag einer ganzen Zahl liefert. Beide Funktionen sind vom Typ `int -> int`.

Präfix-Notation

Im Gegensatz zu den zuvor erwähnten binären Infix-Operationen werden `~` und `abs` in *Präfix-Notation* verwandt.

```
~3 + abs (4-5)
```

Präzedenz der Applikation

Dabei steht die anzuwendende Funktion direkt vor ihrem Argument, möglicherweise durch Leerzeichen getrennt (das heißt, man könnte auch `~␣3+1` oder aber auch `abs(4-5)` schreiben). Die Funktionsapplikation hat die höchste syntaktische Präzedenz. Dies bedeutet, daß im ersten Beispiel die Negation vor der Addition ausgewertet wird und daß beispielsweise der Ausdruck `abs 4-5` den Wert `~1` ergibt. Diese Tatsache führt oft zu Fehlern.

Assoziativität der Applikation

Ebenso wichtig ist die Tatsache, daß die Funktionsapplikation eine *linksassoziative* Operation ist, das heißt, ein Ausdruck der Form

$$f\ g\ x$$

wird implizit als

$$(f\ g)\ x$$

geklammert. Auch dies führt sehr oft zu Verwirrungen. Man darf nämlich *nicht* schreiben

```
~ abs (4-5);
> std_in:1.1 Error: overloaded variable not defined at type
>  symbol: ~
>  type: int -> int
```

denn ML interpretiert dies als Applikation der Negationsfunktion auf die Funktion `abs`, was zu einem Typfehler führt. Die Fehlernachricht ist in diesem Fall wie folgt zu interpretieren: Im allgemeinen kann ein Funktionssymbol wie `~` (hier „variable" genannt) durchaus verschiedene Funktionen bezeichnen (hier: die Negation für ganze Zahlen und für reelle Zahlen). Dann sagt man, das Symbol ist *überladen*. In dem obigen Ausdruck jedoch wird versucht, `~` auf `abs` zu applizieren, eine Funktion vom Typ und `int -> int`, und die Fehlermeldung besagt nun, daß `~` nicht für diesen Typ überladen ist.

Man sollte sich also der Assoziativität von Funktionsapplikationen und der beabsichtigten Klammerung von Argumenten stets bewußt sein.

Notation reeller Zahlen

Reelle Zahlen. Reelle Zahlen (*reals*) werden ebenso wie ganze Zahlen durch eine Folge von Ziffern mit optional vorangestellter Tilde dargestellt, wobei sie zur Unterscheidung immer entweder (i) einen Dezimalpunkt und eine oder mehrere Nachkommaziffern oder (ii) ein `E` gefolgt von einer ganzen Zahl, oder beides enthalten. Dabei wird durch die Exponentenschreibweise $n\mathrm{E}m$ die reelle Zahl $n \times 10^m$ bezeichnet. Der Typ der reellen Zahlen heißt wenig überraschend `real`. Beispiele für reelle Zahlen sind: `2.3`, `1.5E~2` oder `2E1`. Man beachte, daß der letzte Wert *nicht* identisch ist mit der entsprechenden ganzen Zahl `20`. Dies liegt ganz einfach daran, daß sie unterschiedliche Typen besitzen. Werte sind jedoch nur dann vergleichbar, wenn sie den gleichen Typ besitzen.

$20.0 \neq 20$

Die integer-Operationen `+`, `-`, `*`, `~` und `abs` sind auch für reelle Zahlen definiert; die Division auf reellen Zahlen wird mit `/` bezeichnet. Die unären Funktionen haben erwartungsgemäß den Typ `real -> real`, und die binären Operationen sind analog zu denen für integer-Werte vom Typ `real * real -> real`.

Hierbei fällt nun auf, daß Operationen wie `+` oder `*` zwei verschiedene Typen haben. Ist dies nicht ein Widerspruch zu der eingangs erwähnten Eigenschaft, daß jeder Wert einen eindeutig bestimmten Typ hat? Des Rätsels Lösung liegt in der Unterscheidung von Werten und deren Notation. So sind beispielsweise die Addition auf reellen und ganzen Zahlen in der Tat verschiedene Funktionswerte, die aber beide durch das Symbol `+` bezeichnet werden. Man sagt auch, das Symbol `+` ist *überladen* (siehe auch Abschnitt 1.4). ML versucht nun, bei der Interpretation von Ausdrücken aus dem Kontext heraus den richtigen Wert zu einem überladenen Symbol zu ermitteln. In den bisherigen Beispielen war dies stets möglich, da die Argumente der Funktionen jeweils immer vom Typ `int` waren.

Überladen von Symbolen

Zur Konvertierung zwischen reellen und ganzen Zahlen gibt es die beiden Funktionen `real: int -> real` und `floor: real -> int`. Während `real` den eigentlichen Wert des Argumentes unverändert läßt und lediglich

eine Typkonversion vornimmt, rundet `floor` auf eine ganze Zahl, und zwar auf die größte Zahl, die kleiner oder gleich dem Argument ist.

Zeichen und Zeichenketten. Der Typ `string` umfaßt Zeichenketten (*strings*). Diese werden als Folgen von (ASCII-) Zeichen dargestellt, die durch Hochkommata eingeklammert sind. Der Typ `char` enthält dagegen einzelne Zeichen. Die Notation unterscheidet sich durch vorangestelltes `#`. Den Code eines einzelnen Zeichens liefert die Funktion `ord: char -> int` in Form eines integer-Wertes, und invers dazu bildet die Funktion `chr: int -> char` ganze Zahlen im Bereich von 0 bis 255 auf die durch die Zahl kodierten Zeichen ab.

```
ord #"a";
> val it = 97 : int
```

Einzelne Zeichen können mit der Funktion `str: char -> string` in einelementige Zeichenketten umgewandelt werden. Im folgenden betrachten wir nur noch Zeichenketten und behandeln einzelne Zeichen stets als einelementige Zeichenketten.

Als Operationen auf Zeichenketten steht zunächst die Konkatenation `^: string * string -> string` zur Verfügung:

```
"A" ^ "BC" ^ str #"D";
> val it = "ABCD" : string
```

Die Länge einer Zeichenkette kann man mit dem Befehl `size: string -> int` ermitteln. Als Operation zur Bearbeitung von Teilstrings gibt es die Funktion `substring: string * int * int -> string`, die einen Teil einer Zeichenkette liefert. Die Positionen im string-Wert werden dazu bei 0 beginnend durchnumeriert. Beispiel:

Operationen auf Substrings

```
substring ("01234",1,3);
> val it = "123" : string
```

Wahrheitswerte. Die beiden Konstanten `true` und `false` bilden den Typ `bool`. Rein logische Operationen sind zunächst einmal die Negation (`not`), die Konjunktion (`andalso`) und die Disjunktion (`orelse`). Die Namensgebung der beiden letzteren Funktionen deutet auf eine Besonderheit bei deren Auswertung hin: Das zweite Argument von `andalso` (bzw. `orelse`) wird nur dann ausgewertet, wenn das erste Argument `true` (bzw. `false`) ist, da ansonsten der Wert des Ausdrucks nämlich schon allein durch das erste Argument bestimmt ist. Beispiele folgen gleich.

Richtig nützlich werden logische Operationen und Wahrheitswerte erst, wenn man Vergleichsoperationen auf Werten zur Verfügung hat. Alle bisher eingeführten Werte können auf Gleichheit (mit =) bzw. Ungleichheit (mit <>) getestet werden. Ganze und reelle Zahlen können darüber hinaus auch noch auf größer/kleiner (oder gleich) verglichen werden, und zwar mit den binären Operationen >, >=, <, <=. Die Vergleichsoperationen haben geringere Präzedenz als die arithmetischen Operationen.

Auswertung von `andalso` *und* `orelse`

Betrachten wir nun noch einmal die Operationen `andalso` und `orelse`. Man beachte im folgenden Beispiel, daß der zweite Vergleich nicht ausgewertet wird, da der erste Vergleich bereits `false` liefert und damit der gesamte Ausdruck gar nicht mehr `true` werden kann.

```
2<1 andalso 4>1;
> val it = false : bool
```

Man kann diese auf den ersten Blick möglicherweise sehr verzwickt erscheinende Eigenschaft durchaus beim Programmieren ausnutzen. So kann z. B. durch geschicktes Ordnen der Bedingungen einer Konjunktion oder Disjunktion möglicherweise Rechenzeit eingespart werden.

Es gibt noch einen weiteren Aspekt dieser Definition der beiden logischen Operatoren: Wenn der erste logische Wert schon das Ergebnis bestimmt, so führen möglicherweise undefinierte Ausdrücke in der zweiten Bedingung nicht zu einem Laufzeitfehler. So ist z. B. die Division durch 0 nicht definiert und führt zum Abbruch der Auswertung, der folgende Ausdruck aber, obwohl er einen nicht-definierten Teilausdruck enthält, wird ohne Fehler verarbeitet:

```
2>1 orelse (1 div 0) > 0;
> val it = true : bool
```

Striktheit von Funktionen

Eine Funktion, die immer dann undefiniert ist, wenn irgendeines ihrer Argumente undefiniert ist, nennt man auch *strikt*. In diesem Sinne ist die Funktion `orelse`, genauso wie `andalso`, in ihrem zweiten Argument nicht strikt (siehe auch Abschnitt 2.1).

Fallunterscheidung

Ebenfalls nicht strikt (im zweiten und dritten Argument) ist die Fallunterscheidung. Sie hat die Form

`if` *cond* `then` exp_1 `else` exp_2

Die Bedingung *cond* muß ein Ausdruck vom Typ `bool` sein, und die beiden Alternativen exp_1 und exp_2 müssen den gleichen Typ haben. Bei der