Informatik I – Einführung in Algorithmen und in die Programmierung

François Bry

http://www.pms.informatik.uni-muenchen.de/
bry@informatik.uni-muenchen.de

Institut für Informatik, Ludwig-Maximilians–Universität München
Oettingenstraße 67, D-80538 München

2001, 2002

© François Bry (2001, 2002)

Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Inhaltsverzeichnis

0 Einführung in Algorithmen und in die Programmierung — Vorwort 1
0.1 Syllabus der Vorlesung „Informatik I“ . . . 1
0.2 Auslegung des Syllabus . . . 2
0.3 Hinweise für Hörer der Vorlesung . . . 3
0.4 Hinweise zur Prüfungsvorbereitung . . . 4
0.5 Legende . . . 5
0.6 Danksagung . . . 6

1 Einleitung 7
1.1 Spezifikation, Algorithmus und Programm — Begriffserläuterung am Beispiel der Multiplikation . . . 7
1.1.1 Informelle Spezifikation des Multiplikationsalgorithmus . . . 8
1.1.2 Beispiel einer Anwendung des Falles 1 . . . 9
1.1.3 Rekursion und Terminierung . . . 10
1.1.4 Kritik an der Algorithmusbeschreibung aus Abschnitt 1.1.1 . . . 12
1.1.5 Zum Begriff „Algorithmus“ . . . 12
1.1.6 Formale Spezifikation eines Algorithmus . . . 13
1.1.7 Eigenschaften eines Algorithmus: Partielle und totale Korrektheit . 16
1.1.8 Beweisprinzip der vollständigen Induktion . . . 16
1.1.9 Programm . . . 17
1.1.10 Eigenschaften eines Algorithmus: Zeit- und Speicherplatzkomplexität 17
1.2 Was ist Informatik? . . . 18
1.3 Die Programmiersprache der Vorlesung . . . 19
1.4 Inhaltsverzeichnis der Vorlesung . . . 20
1.5 Literatur . . . 20

2 Einführung in die Programmierung mit SML 22
2.1 Antipasti . . . 22
2.1.1 Der Datentyp „ganze Zahl“ . . . 23
2.1.2 Gleichheit für ganze Zahlen . . . 24
2.1.3 Der Datentyp „Boole’scher Wert“ . . . 24
2.1.4 Gleichheit für Boole’sche Werte . . . 25

2.1.5 Überladen . . . 26
2.1.6 Weitere Typen . . . 27
2.1.7 Vergleichsfunktionen für ganze Zahlen und für reelle Zahlen . . . 27
2.1.8 Weitere nützliche Funktionen für ganze Zahlen . . . 27
2.2 Ausdrücke, Werte, Typen und polymorphe Typüberprüfung . . . 28
2.2.1 Ausdrücke, Werte und Typen . . . 28
2.2.2 Typen in Programmiersprachen . . . 29
2.3 Präzedenz- und Assoziativitätsregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken . . . 30
2.4 Namen, Bindungen und Deklarationen . . . 31
2.4.1 Konstantendeklaration — Wertdeklarationen . . . 31
2.4.2 Funktionsdeklaration . . . 31
2.4.3 Funktion als Wert — Anonyme Funktion . . . 32
2.4.4 Formale und aktuelle Parameter einer Funktion . . . 32
2.4.5 Rumpf oder definierender Teil einer Funktionsdeklaration . . . 33
2.4.6 Namen, Variablen und Bezeichner . . . 33
2.4.7 Typ-Constraints . . . 33
2.4.8 Syntax von Namen . . . 34
2.4.9 Dateien laden (einlesen) . . . 34
2.5 Fallbasierte Definition einer Funktion . . . 35
2.5.1 if-then-else . . . 35
2.5.2 Pattern Matching („Musterangleich“) . . . 35
2.6 Definition von rekursiven Funktionen . . . 36
2.6.1 Rekursive Berechnung der Summe der n ersten ganzen Zahlen . . . 36
2.6.2 Effiziente Berechnung der Summe der n ersten ganzen Zahlen . . . 36
2.6.3 Induktionsbeweis . . . 37
2.6.4 Alternativer Beweis . . . 38
2.6.5 Terminierungsbeweis . . . 38
2.7 Wiederdeklaration eines Namens — Statische Bindung — Umgebung . . . 39
2.7.1 Wiederdeklaration eines Namens . . . 39
2.7.2 Statische und dynamische Bindung . . . 39
2.7.3 Umgebung . . . 40
2.8 Totale und partielle Funktionen (Fortsetzung) . . . 40
2.9 Kommentare . . . 40
2.10 Die Standardbibliothek von SML . . . 41
2.11 Beispiel: Potenzrechnung . . . 41
2.11.1 Einfache Potenzrechnung . . . 41
2.11.2 Terminierungsbeweis für die einfache Potenzrechnung . . . 41
2.11.3 Zeitbedarf der einfachen Potenzberechnung . . . 42
2.11.4 Effizientere Potenzrechnung . . . 42

2.11.5 Zeitbedarf der effizienteren Potenzberechnung 42
2.11.6 Bessere Implementierung der effizienteren Potenzrechnung 43

3 Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen) 44

3.1 Auswertung von Ausdrücken 44
3.1.1 Arten von Ausdrücken 44
3.1.2 Die Auswertung von Ausdrücken als Algorithmus 45
3.1.3 Die Auswertung von Ausdrücken als rekursive Funktion 46
3.1.4 Unvollständigkeit des obigen Algorithmus 48
3.1.5 Zweckmäßigkeit des obigen Algorithmus 48
3.1.6 Beispiel einer Durchführung des Auswertungsalgorithmus 49
3.1.7 Substitutionsmodell 51
3.2 Auswertung in applikativer und in normaler Reihenfolge 51
3.2.1 Auswertungsreihenfolge 51
3.2.2 Auswertung in applikativer Reihenfolge 52
3.2.3 Auswertung in normaler Reihenfolge 53
3.2.4 Vorteil der applikativen Reihenfolge gegenüber der normalen Reihenfolge 53
3.3 Verzögerte Auswertung 53
3.3.1 Vorteil der normalen Reihenfolge gegenüber der applikativen Reihenfolge 53
3.3.2 Verweis 55
3.3.3 Auswertungsreihenfolge von SML 56
3.4 Auswertung der Sonderausdrücke 56
3.4.1 Wertdeklarationen (`val` und `fun`) 56
3.4.2 if-then-else 56
3.4.3 Pattern Matching 58
3.4.4 Die Boole'schen Operatoren `andalso` und `orelse` 59
3.4.5 Infixoperator-Deklarationen und Präzedenzen 60
3.4.6 Erweiterter Auswertungsalgorithmus mit Behandlung von Sonderausdrücken 61
3.5 Funktionale Variablen versus Zustandsvariablen 62
3.5.1 Funktionale Variablen 62
3.5.2 Zustandsvariablen 63
3.5.3 Zustandsvariablen in SML: Referenzen 67
3.6 Funktionale Programmierung versus Imperative Programmierung 68
3.6.1 Überschatten versus Zustandsänderung 68
3.6.2 Funktion versus Prozedur 68
3.6.3 Unzulänglichkeit des Substitutionsmodells zur Behandlung von Zustandsvariablen 69

3.6.4 Rein funktionale Programme und Ausdrücke . . . 69
3.6.5 Nebeneffekte . . . 70
3.6.6 Reihenfolge der Parameterauswertung . . . 70

4 Prozeduren zur Abstraktionsbildung 71
4.1 Die „Prozedur": Ein Kernbegriff der Programmierung . . . 71
4.1.1 Prozeduren zur Programmzerlegung . . . 71
4.1.2 Vorteile von Prozeduren . . . 72
4.1.3 Funktion versus Prozedur . . . 73
4.1.4 Definition von Funktionen und Prozeduren in SML . . . 73
4.2 Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen . 74
4.2.1 Lokale Deklarationen mit „`let`" . . . 74
4.2.2 Lokale Deklarationen mit „`local`" . . . 76
4.2.3 Unterschied zwischen `let` und `local` . . . 77
4.2.4 Blockstruktur und Überschatten . . . 78
4.2.5 Festlegung der Geltungsbereiche von Namen — Einführung . . . 79
4.2.6 Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung . . . 79
4.2.7 Überschatten durch verschachtelte lokale Deklarationen . . . 80
4.2.8 Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung — Fortsetzung . . . 81
4.3 Prozeduren versus Prozesse . . . 85
4.3.1 Notwendigkeit der Rekursion . . . 85
4.3.2 Rekursion versus Iteration: Grundkonstrukte beider Berechnungsmodelle im Vergleich . . . 85
4.3.3 Rekursion versus Iteration: Komplexitätsaspekte . . . 86
4.3.4 Endrekursion . . . 89
4.3.5 Lineare und quadratische Rekursion, Rekursion der Potenz n . . . 90
4.3.6 Iterative Auswertung der baumrekursiven Funktion fib . . . 91
4.3.7 Memoisierung . . . 92
4.3.8 Prozedur versus Prozess . . . 93
4.4 Ressourcenbedarf — Größenordnungen . . . 93
4.5 Beispiel: Der größte gemeinsame Teiler . . . 95

5 Die vordefinierten Typen von SML 100
5.1 Was sind Typen? . . . 100
5.2 Die Basistypen von SML . . . 101
5.2.1 Ganze Zahlen . . . 101
5.2.2 Reelle Zahlen . . . 101
5.2.3 Boole'sche Werte . . . 103
5.2.4 Zeichenfolgen . . . 103

5.2.5 Zeichen . . . 104
5.2.6 `unit` . . . 105
5.3 Zusammengesetzte Typen in SML . . . 105
5.3.1 Vektoren (Tupel) . . . 105
5.3.2 Deklaration eines Vektortyps . . . 108
5.3.3 Verbunde (Records) . . . 108
5.3.4 Deklaration eines Vektor- oder Verbundstyps . . . 110
5.3.5 Vektoren als Verbunde . . . 110
5.4 Listen . . . 110
5.4.1 Der Begriff „Liste“ in Algorithmenspezifikations- und Programmiersprachen . . . 111
5.4.2 Die Listen in SML . . . 112
5.4.3 Mono- und Polytypen . . . 113
5.5 Beispiele: Grundlegende Listenfunktionen . . . 113
5.5.1 Länge einer Liste . . . 113
5.5.2 Letztes Element einer nichtleeren Liste . . . 113
5.5.3 Kleinstes Element einer nichtleeren Liste von ganzen Zahlen . . . 114
5.5.4 n-tes Element einer Liste . . . 114
5.5.5 `head` . . . 114
5.5.6 `tail` . . . 115
5.5.7 `append` . . . 115
5.5.8 `naive-reverse` . . . 116
5.5.9 reverse . . . 117
5.6 Hinweis auf die Standardbibliothek von SML . . . 118

6 Typprüfung 119
6.1 Die Typprüfung: Eine nützliche Abstraktion für die Entwicklung von korrekten Programmen . . . 119
6.2 Statische versus dynamische Typprüfung . . . 120
6.3 Die Polymorphie: Eine wünschenswerte Abstraktion . . . 121
6.3.1 Polymorphe Funktionen, Konstanten und Typen . . . 121
6.3.2 Typen von Vorkommen eines polymorphen Ausdrucks . . . 122
6.3.3 Vorteile der Polymorphie . . . 122
6.4 Polymorphie versus Überladung . . . 122
6.5 Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML 123
6.5.1 Typvariablen . . . 123
6.5.2 Typinferenz . . . 124
6.5.3 Typausdrücke . . . 125
6.5.4 Typkonstanten . . . 125
6.5.5 Typ-Constraints . . . 125

6.5.6 Zusammengesetzte Typausdrücke und Typkonstruktoren 126
6.5.7 Die ''-Typvariablen zur Polymorphie für Typen mit Gleichheit . . 126
6.6 Typkonstruktor versus Wertkonstruktor 127
6.7 Schlussregeln für die Typinferenz 128
6.7.1 Eine Vereinfachung von SML: SMalL 128
6.7.2 Logischer Kalkül 129
6.7.3 Gestalt der Schlussregeln eines logischen Kalküls 129
6.7.4 Beweisbegriff in logischen Kalkülen 130
6.7.5 Die Schlussregeln für die Typinferenz oder „Typisierungsregeln“ . . 131
6.7.6 Typisierungsbeweise 132
6.7.7 Beispiele für Typisierungsbeweise 132
6.8 Der Unifikationsalgorithmus 134
6.9 Ein Verfahren zur automatischen Typinferenz 135
6.9.1 Prinzip des Verfahrens 135
6.9.2 Behandlung der Überschattung 135
6.9.3 Beispiele 137

7 Abstraktionsbildung mit Prozeduren höherer Ordnung 140
7.1 Prozeduren als Parameter und Wert von Prozeduren 140
7.2 Currying 143
7.2.1 Prinzip 143
7.2.2 Andere Syntax zur Deklaration von „curried“ Funktionen 144
7.2.3 Einfache Deklaration von curried Funktionen 145
7.2.4 Die Funktion höherer Ordnung `curry` zur Berechnung der curried Form einer binären Funktion 146
7.2.5 Umkehrung der Funktion `curry` 147
7.2.6 Nicht-curried und curried Funktionen im Vergleich 147
7.3 Funktionskomposition 148
7.3.1 Funktionskomposition 148
7.3.2 Die Kombinatoren `I`, `K` und `S` 149
7.4 Grundlegende Funktionen höherer Ordnung 150
7.4.1 `map` 150
7.4.2 Vorteil der curried Form am Beispiel der Funktion `map` 151
7.4.3 `filter` 151
7.4.4 Vorteil der curried Form am Beispiel der Funktion `filter` 152
7.4.5 `foldl` und `foldr` 152
7.4.6 `exists` und `all` 155
7.4.7 Wiederholte Funktionsanwendung 156
7.5 Beispiel: Ein Rekursionsschema zur Akkumulation 156
7.5.1 `summe` 156

7.5.2 `produkt` . . . 156
7.5.3 Das gemeinsame Rekursionsschema . . . 157
7.5.4 Anwendung zur Integralschätzung . . . 157

8 Abstraktionsbildung mit neuen Typen 159
8.1 Typen im Überblick . . . 159
8.1.1 Typ als Wertemenge . . . 159
8.1.2 Typen mit atomaren und zusammengesetzten Werten . . . 159
8.1.3 Typen in Programmiersprachen mit erweiterbaren Typsystemen . . . 160
8.1.4 Monomorphe und Polymorphe Typen . . . 160
8.1.5 (Wert-)Konstruktoren und (Wert-)Selektoren eines Typs . . . 160
8.1.6 Typkonstruktoren . . . 160
8.2 Deklarationen von Typabkürzungen in SML: type-Deklarationen . . . 161
8.2.1 Typabkürzungen . . . 161
8.2.2 Grenzen der Nutzung von Typabkürzungen . . . 161
8.2.3 Nützlichkeit von Typabkürzungen: Erstes Beispiel . . . 162
8.2.4 Nützlichkeit von Typabkürzungen: Zweites Beispiel . . . 163
8.2.5 Polymorphe Typabkürzungen . . . 163
8.3 Definition von Typen: datatype-Deklarationen . . . 164
8.3.1 Definition von Typen mit endlich vielen atomaren Werten . . . 164
8.3.2 Definition von Typen mit zusammengesetzten Werten . . . 166
8.3.3 Gleichheit für Typen mit zusammengesetzten Werten . . . 167
8.3.4 „Typenmix“ . . . 168
8.4 Definition von rekursiven Typen . . . 168
8.4.1 Wurzel, Blätter, Äste, Bäume und Wälder . . . 169
8.4.2 Induktive Definition . . . 171
8.4.3 Induktive Definition und rekursive Algorithmen . . . 171
8.4.4 Darstellung von Bäumen: graphisch und durch Ausdrücke . . . 171
8.4.5 Rekursive Typen zum Ausdrücken von induktiven Definitionen — Der Binärbaum . . . 173
8.4.6 Polymorphe Typen . . . 174
8.4.7 Suche in Binärbäumen . . . 175
8.4.8 Die Liste als Binärbaum . . . 175
8.5 Beweisprinzip der strukturellen Induktion . . . 176
8.6 Beispiele: Implementierungen des Konzepts der „Menge“ . . . 177
8.6.1 Was ist eine „Menge“ . . . 177
8.6.2 Was ist eine „Datenstruktur”? . . . 178
8.6.3 Die Menge als charakteristische Funktion . . . 179
8.6.4 Die Menge als Liste . . . 180
8.6.5 Die Menge als sortierte Liste . . . 181

8.6.6 Die Menge als binärer Suchbaum . . . 182
8.7 Beispiele: Grundlegende Funktionen für binäre (Such-)Bäume . . . 188
8.7.1 Selektoren und Prädikate . . . 190
8.7.2 Durchlauf in Infix-Reihenfolge . . . 190
8.7.3 Durchlauf in Präfix-Reihenfolge . . . 191
8.7.4 Durchlauf in Postfix-Reihenfolge . . . 191
8.7.5 Infix-/Präfix-/Postfix-Reihenfolge mit Akkumulatortechnik . . . 192
8.7.6 Tiefendurchlauf (Depth-First-Durchlauf) . . . 193
8.7.7 Breitendurchlauf (Breadth-First-Durchlauf) . . . 195

9 Pattern Matching 197
9.1 Zweck des Pattern Matching . . . 197
9.1.1 Muster in Wertdeklarationen . . . 197
9.1.2 Muster zur Fallunterscheidung in Funktionsdefinitionen . . . 198
9.1.3 Muster zur Fallunterscheidung in `case`–Ausdrücken . . . 200
9.2 Prinzip des Pattern Matching . . . 200
9.2.1 Angleichregel . . . 200
9.2.2 Prüfung einer Angleichregel gegen einen Wert . . . 201
9.2.3 Prüfung eines Angleichmodells gegen einen Wert . . . 201
9.2.4 Typkorrektheit eines Angleichmodells . . . 202
9.2.5 Herkömmliche Angleichmodelle in SML . . . 202
9.3 Pattern Matching und statische Typprüfung — Angleichfehler zur Laufzeit 202
9.3.1 Pattern Matching und statische Typprüfung . . . 202
9.3.2 Angleichfehler zur Laufzeit . . . 203
9.4 Das Wildcard–Muster von SML . . . 203
9.5 Das Verbund-Wildcard–Muster von SML . . . 204
9.6 Die gestuften Muster von SML . . . 205
9.7 Linearitätsbedingung für Muster . . . 206
9.8 Der Pattern-Matching-Algorithmus . . . 206
9.8.1 Informelle Spezifikation des Pattern-Matching–Algorithmus . . . 206
9.8.2 Umgebung (Wiederholung aus Kapitel 2) . . . 207
9.8.3 Formale Spezifikation des Pattern-Matching–Algorithmus . . . 207
9.8.4 Beispiel einer Anwendung des Pattern-Matching–Algorithmus . . . 208
9.8.5 Korrektheit und Terminierung des Pattern-Matching–Algorithmus . 209
9.8.6 Pattern Matching und Unifikation . . . 210
9.8.7 Folgen der Linearätsbedingung für den Pattern-Matching–Algorithmus . . . 210

10 Auswertung und Ausnahmen 212
10.1 Die Programmiersprache SMalL . . . 212
10.1.1 Typen in SMalL . . . 212

10.1.2 Verzweigung in SMalL . . . 213
10.1.3 Globale und lokale Deklarationen in SMalL . . . 213
10.1.4 Rekursive Funktionen in SMalL . . . 214
10.2 Die abstrakte Syntax von SMalL . . . 214
10.2.1 Abstrakte Syntax versus konkrete Syntax . . . 214
10.2.2 SML-Typdeklarationen für SMalL-Ausdrücke . . . 215
10.2.3 Beispiele von SMalL-Ausdrücken in konkreter und abstrakter Syntax 216
10.3 Ein Auswerter für SMalL: Datenstrukturen . . . 222
10.3.1 Werte und Umgebungen . . . 222
10.3.2 Darstellung von SMalL-Werten und SMalL-Umgebungen in SML . . . 223
10.3.3 SML-Typdeklarationen für SMalL-Werte und SMalL-Umgebungen . 225
10.3.4 Typ des Auswerters für SMalL . . . 225
10.4 Ein Auswerter für SMalL: Programm `eval1` . . . 226
10.4.1 Auswertung von ganzen Zahlen . . . 226
10.4.2 Auswertung von unären Operationen über ganzen Zahlen . . . 226
10.4.3 Auswertung binärer Operationen über ganzen Zahlen . . . 227
10.4.4 Auswertung von Verzweigungen . . . 227
10.4.5 Auswertung von Variablen (oder Namen oder Bezeichnern) . . . 228
10.4.6 Auswertung von Deklarationen . . . 228
10.4.7 `val`–Funktionsdeklarationen versus `val-rec`–Funktionsdeklarationen 229
10.4.8 Auswertung von Funktionsausdrücken . . . 230
10.4.9 Auswertung von Funktionsanwendungen . . . 230
10.4.10 Auswertung von Sequenzen . . . 231
10.4.11 Abhängigkeit des Auswerters `eval1` von SML . . . 231
10.4.12 Gesamtprogramm des Auswerters für SMalL . . . 231
10.4.13 Beispiele . . . 231
10.4.14 Die Start-Umgebung . . . 234
10.5 Behandlung fehlerhafter SMalL-Ausdrücke — `eval2` und `eval3` . . . 237
10.5.1 Schwäche des Auswerters `eval1` für SMalL . . . 237
10.5.2 Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2` 237
10.5.3 Veränderter Auswerter für SMalL mit Sonderwerten . . . 239
10.5.4 Unzulänglichkeit des veränderten Auswerters mit Sonderwerten . . . 239
10.5.5 Verbesserung des Auswerters mit SML-Ausnahmen — `eval3` . . . 241
10.6 Der SML-Typ `exn` („exception“) . . . 243
10.6.1 Der vordefinierte Typ `exn` . . . 243
10.6.2 Ausnahmekonstruktoren . . . 243
10.6.3 Ausnahmen erheben (oder werfen) . . . 243
10.6.4 Ausnahmen als Werte . . . 244
10.6.5 Ausnahme versus Wert . . . 245
10.6.6 Ausnahmen behandeln (oder einfangen) . . . 246

10.6.7 Prinzip der Auswertung von `raise`– und `handle`–Ausdrücken . . . 248
10.6.8 Vordefinierte Ausnahmen von SML 250
10.7 Erweiterung von SMalL um SMalL-Ausnahmen — `eval4` 250
10.7.1 Erweiterung der Programmiersprache SMalL — Konkrete Syntax . 250
10.7.2 Erweiterung der Programmiersprache SMalL — Abstrakte Syntax . 250
10.7.3 Erweiterung der SML-Typdeklarationen des Auswerters 251
10.7.4 Erweiterung des Auswerters 251
10.7.5 Beispiele 253
10.7.6 Weitergabe von Sonderwerten und Ausnahmebehandlung im Vergleich 253
10.8 Rein funktionale Implementierung des Auswerters — `eval5` 255
10.8.1 Verweise als Funktionen 255
10.8.2 Ein rein funktionaler Auswerter für SMalL 257
10.9 Meta- und Objektsprache, Bootstrapping 258

11 Bildung von Abstraktionsbarrieren mit abstrakten Typen und Modulen 260
11.1 Vorzüge des Verbergens 260
11.2 Abstrakte Typen in SML 261
11.2.1 Motivationsbeispiel: Das Farbmodell von HTML 261
11.2.2 Ein SML-Typ zur Definition des Farbmodells von HTML 262
11.2.3 Ein abstrakter Typ zur Definition der Farben von HTML 263
11.2.4 Implementierungstyp und Schnittstelle eines abstrakten Typs 265
11.2.5 Vorteile des Verbergens mit abstrakten Typen 265
11.3 Beispiel: Abstrakte Typen zur Implementierung der Datenstruktur „Menge“ 266
11.3.1 Erster abstrakter Typ zur Implementierung der Menge als Liste . . 267
11.3.2 Zweiter abstrakter Typ zur Implementierung der Menge als Liste . 269
11.4 Module in SML 271
11.4.1 SML-Strukturen 272
11.4.2 SML-Signaturen 273
11.4.3 Spezifikation versus Deklaration in SML-Signaturen 275
11.4.4 `eqtype`–Spezifikationen in SML-Signaturen 275
11.4.5 `datatype`–Spezifikationen in SML-Signaturen 275
11.4.6 Angleich einer Struktur an eine Signatur — Struktursichten 275
11.4.7 Parametrisierte Module in SML: SML-Funktoren 276
11.4.8 Generative und nichtgenerative Strukturdeklarationen 278
11.4.9 Weiteres über Module in SML 278
11.5 Hinweis auf die Standardbibliothek von SML 279

12 Imperative Programmierung in SML 281
12.1 SML-Referenzen 281
12.1.1 Deklaration einer Referenz — Referenzierungsoperator in SML . . . 281

12.1.2 Typ einer Referenz . . . 282
12.1.3 Dereferenzierungsoperator in SML . . . 282
12.1.4 Sequenzierungsoperator in SML . . . 283
12.1.5 Zuweisungsoperator in SML . . . 283
12.1.6 Druckverhalten von SML bei Referenzen . . . 284
12.1.7 Gleichheit zwischen Referenzen . . . 284
12.1.8 Vordefinierte SML-Prozeduren über Referenzen . . . 285
12.2 Iteration in SML . . . 285
12.3 SML-Felder . . . 286
12.3.1 Deklaration eines Feldes . . . 286
12.3.2 Zugriff auf die Komponenten eines Feldes . . . 287
12.3.3 Veränderung der Komponenten eines Feldes . . . 287
12.3.4 Länge eines Feldes . . . 287
12.3.5 Umwandlung einer Liste in ein Feld . . . 288
12.3.6 Umwandlung eines Feldes in eine Liste . . . 288
12.3.7 Gleichheit für Felder . . . 288
12.3.8 Hinweis auf die Standardbibliothek von SML . . . 289
12.4 Beispiel: Sortieren eines Feldes durch direktes Einfügen (straight insertion sort) . . . 289
12.4.1 Totale Ordnung . . . 289
12.4.2 Sortieren . . . 289
12.4.3 Internes Sortieren durch direktes Einfügen (straight insertion sort) . 290
12.4.4 Komplexität des internen Sortierens durch direktes Einfügen . . . 292
12.5 Ein- und Ausgaben in SML . . . 293
12.5.1 Datenströme (streams) . . . 293
12.5.2 Ausgabestrom . . . 294
12.5.3 Eingabestrom . . . 295
12.5.4 Standard–Ein- und -Ausgabetextströme `TextIO.stdIn` und `TextIO.stdOut` . . . 296
12.5.5 Die vordefinierte Prozedur `print` . . . 296
12.5.6 Beispiel: Inhalt einer Datei einlesen und an die Standardausgabe weiterleiten . . . 296
12.5.7 Hinweis auf das Modul `TextIO` der Standardbibliothek von SML . . 297

13 Formale Beschreibung der Syntax und Semantik von Programmiersprachen 298

13.1 Formale Beschreibung der Syntax einer Programmiersprache . . . 298
13.1.1 Syntax versus Semantik . . . 298
13.1.2 Ziele von formalen Beschreibungen der Syntax von Programmiersprachen . . . 300

13.1.3 Lexikalische Analyse und Symbole versus Syntaxanalyse und Programme . . . 300
13.1.4 EBNF-Grammatiken zur formalen Syntaxbeschreibungen . . . 301
13.1.5 Kontextfreie Grammatiken zur formalen Beschreibung der Syntax von Programmen . . . 304
13.1.6 Syntaxdiagramme . . . 307
13.2 Formale Beschreibungen der Semantik einer Programmiersprache: Ziele und Ansätze . . . 308
13.3 Einführung in die denotationelle Semantik funktionaler Programmiersprachen 309
13.3.1 Mathematische Funktionen zur Repräsentation von Programmfunktionen . . . 309
13.3.2 Programmfunktionen versus mathematische Funktionen . . . 310
13.3.3 Werte von (mathematischen) Funktionen zur Repräsentation von Programmausdrücken ohne (Programm-) Werte . . . 311
13.3.4 Syntaktische und semantische (Wert-)Domäne, Semantikfunktionen 312
13.3.5 Strikte und nicht-strikte (mathematische) Funktionen . . . 312
13.3.6 Approximationsordnung . . . 313
13.3.7 Denotation einer Funktionsdeklaration . . . 317
13.3.8 Semantische Domäne . . . 318
13.3.9 Denotationelle Semantik einer rein funktionalen Programmiersprache 325
13.3.10 Fixpunktsemantik rekursiver Funktionen . . . 330

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 0

Einführung in Algorithmen und in die Programmierung — Vorwort

0.1 Syllabus der Vorlesung „Informatik I“

Der Inhalt der Vorlesung „Informatik I – Einführung in Algorithmen und in die Programmierung“ sowie der anderen Informatik-Vorlesungen des Grundstudiums der Studiengänge Informatik und Bioinformatik an der Universität München wurde im Herbst 1998 von den Professoren des Instituts für Informatik mit dem Ziel festgelegt, alle Jahre in diesen Vorlesungen dieselben Inhalte zu behandeln.

So wird Studenten, die eine Vorlesung des Grundstudiums mehrmals hören, das Lernen erleichtert.

Nach dieser Vereinbarung ist der Syllabus der Vorlesung „Informatik I”wie folgt. Dabei handelt es sich um einen Themenkatalog, der zu keiner bestimmten Reihenfolge verpflichtet:

1. Einführung in die Programmierung mit der funktionalen Programmiersprache SML (Standard ML)
   (a) Grundbegriffe: Programmaufbau, Basisdatentypen und Funktionen
   (b) Rekursion
   (c) Typen und Typausdrücke
   (d) Vektoren (Tupel) und Listen
   (e) Zusammengesetzte Datentypen: Ansatz, Enumerationstypen und benutzerdefinierte Produkttypen
   (f) Rekursive Datentypen
   (g) „Musterangleich“ (Pattern Matching)
   (h) Ausnahmen
   (i) Funktionen höherer Ordnung
2. Grundlagen der Programmierung
   (a) Der Algorithmusbegriff

   (b) Pseudo-Code
   (c) Präfix-, Infix und Postfixoperatoren, Präzedenz- und Assoziativitätsregeln für Operatoren
   (d) Baumdarstellung von Ausdrücken, konkrete und abstrakte Syntax
   (e) Prozeduren, lokale Deklarationen und Überschatten
   (f) Statische und Dynamische Bindung
   (g) Polymorphie
   (h) Moduln
   (i) Der Keller und die Binärbäume

3. Prinzip der Auswertung von funktionalen Programmen

   (a) Auswertung in applikativer und in normaler Reihenfolge, verzögerte Auswertung
   (b) Substitutionsmodell
   (c) Umgebung
   (d) Ausnahmebehandlung

4. Formale Beschreibung von Programmiersprachen

   (a) Syntaxbeschreibung: Backus-Naur-Form und Syntaxdiagramme
   (b) Statische und dynamische Typprüfung, Typinferenz
   (c) Semantik der Rekursion: Domäne, Approximationsordnung und Fixpunktsatz von Kleene

5. Einführung in die imperative Programmierung

   (a) Zustandsvariablen
   (b) Zuweisung
   (c) Sequenzierung
   (d) Fallunterscheidung
   (e) Wiederholungsanweisungen
   (f) Zeiger (Pointer)

0.2 Auslegung des Syllabus

Dieses Skriptum wurde für die Vorlesung des Wintersemesters 2000/2001 verfasst und für die Vorlesung im Wintersemester 2001/2002 überarbeitet. Die folgenden Themen, die im Syllabus nicht erwähnt sind, sind aus den folgenden Gründen in diesem Skriptum enthalten:

- Einführung in die Komplexität von Algorithmen: Zeit- und Speicherplatzkomplexität, O-Notation.

  Ohne Komplexitätsfragen zu untersuchen, lassen sich Eigenschaften von Programmen kaum beweisen. Die Zeiten sind aber längst vorbei, zu denen Programme entwickelt werden konnten, ohne dabei ihre Eigenschaften formal zu untersuchen. Zudem ist es in Studiengängen mit hohem mathematischen Anteil sinnvoll, früh in formale Aspekte der Programmentwicklung einzuführen.

- Pattern-Matching– und Unifikationsalgorithmen.

  Zur Erläuterung des Typinferenzalgorithmus ist der Unifikationsalgorithmus unabdingbar. Wird er eingeführt, dann bietet sich an, ebenfalls den Pattern-Matching–Algorithmus zu behandeln, der eine sehr verständliche Vereinfachung des Unifikationsalgorithmus ist.

- Auswertungsalgorithmus für rein funktionale Programme.

  Die Behandlung dieses Algorithmus in der Vorlesung „Informatik I“ hat mehrere Vorteile. Zum einen trägt die Kenntnis des Auswertungsalgorithmus wesentlich zum guten Verständnis der Programmiersprache bei. Zum zweiten stellt der Auswertungsalgorithmus einen nicht zu kleinen, jedoch übersichtlichen Algorithmus dar, dessen Behandlung in der Vorlesung zu einer praxisnahen Lehre beiträgt. Zum dritten sind Ausnahmen und die Ausnahmebehandlung im Zusammenhang mit dem Auswertungsalgorithmus leichter zu vermitteln als für sich allein. Viertens ist es sicherlich lehrreich für Informatikanfänger, mit dem Auswertungsalgorithmus zu erfahren, mit welchen überraschend einfachen Mitteln mächtige Konzepte von Programmiersprachen wie Rekursion, das Überschatten und die Funktionen höherer Ordnung implementiert werden.

- Semantikfunktionen zur denotationellen Semantik einer rein funktionalen Programmiersprache

  Wird der Fixpunktsatz von Kleene zur Formalisierung der Semantik von rekursiven Programmen behandelt, was der Syllabus vorschreibt, so ist der zusätzliche Aufwand zur Erläuterung des Begriffes Semantikfunktion sehr gering. Der Einblick in die denotationelle Semantik lohnt sich, weil er einen wichtigen Aspekt der Entwicklung einer sicheren Software, nämlich die Formalisierung seiner Semantik, bekannt macht.

Der Datentyp Keller wird nicht losgelöst von einer Anwendung behandelt, sondern in Zusammenhang mit der Umgebung zur Verwaltung von Variablenbindungen während der Auswertung von Programmen eingeführt.

0.3 Hinweise für Hörer der Vorlesung

Die Vorlesung „Informatik 1“ ist die erste von 6 (Studiengang Informatik), 5 (Studiengang Medieninformatik) bzw. 3 (Studiengang Bioinformatik) Grundstudiumsvorlesungen in Informatik, die eine „Grand Tour“ durch die Teilgebiete der Informatik bilden. Für die

Studiengänge Bioinformatik und Medieninformatik gibt es jeweils eigene Grundstudiumsveranstaltungen, die in die spezifischen Fragestellungen dieser Fächer einführen.

Der empfehlenswerte Arbeitsaufwand eines Hörers dieser Vorlesung ist wie folgt (wobei der tatsächliche Aufwand erheblich von den persönlichen Vorkenntnissen und fachlichen Neigungen abhängen kann):

Richtwerte für den Studiengang Informatik

Lineare Algebra	12	Wochenstunden (einschließlich persönliche Arbeit)
Analysis	12	
Informatik 1	12	
Nebenfach	9	
	45	Wochenstunden

Richtwerte für den Studiengang Bioinformatik

Bioinformatik I	9	Wochenstunden (einschließlich persönliche Arbeit)
Lineare Algebra	12	
Informatik 1	12	
Biologie/Chemie	12	
	45	Wochenstunden

Richtwerte für den Studiengang Medieninformatik

Einfhrung in die Kommunikationswissenschaft	3	Wochenstunden (einschließlich persönliche Arbeit)
Kommunikationstheorien	3	
Medienkunde	3	
Lineare Algebra	12	
Analysis	12	
Informatik 1	12	
	45	Wochenstunden

0.4 Hinweise zur Prüfungsvorbereitung

Das vorliegende Skriptum gibt (in eigener Weise) die Inhalte wieder, die im Wintersemester 1999/2000 in der Vorlesung „Informatik I“ vom Prof. Dr. Martin Wirsing behandelt worden sind. Zudem sind meine Vorlesung „Informatik I“ des Wintersemesters 2000/2001 und die Vorlesung „Informatik I“ von Prof. Dr. Stefan Conrad des Wintersemesters 2001/2002 nach diesem Skriptum gehalten worden. Folglich ist dieses Skriptum zur Prüfungsvorbereitung gleichermaen für Hörer geeignet ist, die die Vorlesung „Informatik I“ in den Wintersemestern 1999/2000, 2000/2001 und 2001/2002 gehört haben.

0.5 Legende

Im diesem Skriptum werden die verschiedenen Textelemente wie folgt dargestellt:

Definition (Begriff)
...

Satz:
...

Beweis:

...

qed.

Algorithmus

```
Programmbeispiel

fun append(nil, l) = l
  | append(h :: t, l) = h :: append(t, l);
```

```
Sitzung

- fun append(nil, l) = l
    | append(h :: t, l) = h :: append(t, l);
val append = fn : 'a list * 'a list -> 'a list
```

```
Ablaufprotokoll

quadrat(quadrat(2))
quadrat(2 * 2)
quadrat(4)
4 * 4
16
```

0.6 Danksagung

Die folgenden Personen haben mir bei der Erstellung dieses Vorlesungsskriptums geholfen: Dr. Norbert Eisinger hat stilistische und inhaltliche Verbesserungsvorschläge gemacht und den rein funktionalen Auswerter eval5 (in Abschnitt 10.5) aus dem Auswerter eval4 (in Abschnitt 10.4) entworfen. Dr. Reinhard Kahle hat das 2. Beispiel in Abschnitt 6.9.3 verfasst. Stefan Fischer und Johannes Martens haben das Skriptum unter Verwendung von XHTML 1.0 und CSS2 formatiert. Letztlich haben die Hörer und Übungsleiter der Vorlesung im Wintersemester 2000/01 mit ihren Fragen und Anregungen einen wichtigen Beitrag zu diesem Skriptum geleistet.

Ich bedanke mich bei allen herzlich.

München, im Juli 2001 François Bry

Die vorliegende Fassung des Skriptums, die kleine Fehler beseitigt, beruht auf der HTML-Fassung vom 2001, die unter

http://www.pms.informatik.uni-muenchen.de/publikationen/lecture-notes/info1/

abrufbar ist. Sie wurde im Wintersemester 2000/2001 von Prof. Dr. Stefan Conrad, Eshref Januzaj, Karin Kailing, Peer Kröger und Martin Pfeifle erstellt.

Ich bedanke mich bei ihnen herzlich.

München, im Februar 2002 François Bry

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 1

Einleitung

Ziel dieses Kapitels ist es, einige zentrale Begriffe dieser Vorlesung informell einzuführen: „Spezifikation“, „Algorithmus“, „Funktion“, „Rekursion“, „Programm“, usw. Dann wird die Frage „Was ist die Informatik?“ kurz angesprochen. Abschließend werden das Inhaltsverzeichnis der Vorlesung sowie Literaturhinweise angegeben.

1.1 Spezifikation, Algorithmus und Programm — Begriffserläuterung am Beispiel der Multiplikation

Einer der bekanntesten Algorithmen ist das Verfahren zur Multiplikation zweier natürlicher Dezimalzahlen.

```
 4 2 3   x    2  4  3  1
            ------------
             (1)
              7  2  9  3

           4  8  6  2  0
        (1)(1)
         9  7  2  4  0  0
       ------------------
      (1)(1)(1)(1)(1)
       1  0  2  8  3  1  3
```

Wie kann dieses Verfahren im allgemeinen Fall beschrieben werden?

1.1.1 Informelle Spezifikation des Multiplikationsalgorithmus

Seien `Multiplikator` und `Multiplikand` die natürlichen Zahlen, die multipliziert werden sollen:

```
Multiplikator x  Multiplikand
              ---------------
                      Produkt
```

Eine Multiplikationstabelle enthalte ferner alle Ergebnisse der Multiplikation einstelliger Zahlen; die also nur aus einer Ziffer bestehen („das kleine 1x1“).

Fall 1: `Multiplikator` ist eine Ziffer

Fall 1.1: `Multiplikand` ist eine Ziffer. Ermittle aus der Multiplikationstabelle:
`Ergebnis = Multiplikator x Multiplikand`.
Liefere `Ergebnis`

Fall 1.2: `Multiplikand` ist keine Ziffer. Es gilt also:
`Multiplikand = Z Rest`,
wobei `Z` eine Ziffer und `Rest` eine natürliche Zahl ist.

Wenn `Produkt1 = Multiplikator x Rest` bekannt wäre, könnte man `Produkt` wie folgt berechnen.

`Produkt2 = Multiplikator x Z` liefert die Multiplikationstabelle,
weil `Multiplikator` und `Z` Ziffern sind.
Sei n die Länge von `Rest`, d.h. die Anzahl der Ziffern in `Rest`.
Sei `GeschobenProdukt2` die natürliche Zahl, die sich ergibt, wenn n Nullen an `Produkt2` angefügt werden.
Dann gilt: `Produkt = Produkt1 + GeschobenProdukt2`

Es ist aber unwesentlich, dass `Produkt2` noch nicht ermittelt wurde, um die Gleichung `Produkt1 = Multiplikator x Rest` aufzustellen. Zur Berechnung (des Wertes) von `Produkt1` kann man auf den Fall 1 zurückgreifen. (Dieser Fall trifft zu, weil nach Annahme `Multiplikator` eine Ziffer ist.)

Der Fall 1.2 lässt sich also wie folgt zusammenfassen:

`Multiplikand` ist keine Ziffer.
Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:
`Multiplikand = Z Rest`
Berechne (mit demselben Algorithmus Fall 1):
`Produkt1 = Multiplikator x Rest`
Ermittle aus der Multiplikationstabelle:
`Produkt2 = Multiplikator x Z`
Berechne
`GeschobenProdukt2 = Produkt2` gefolgt von n Nullen,
wobei n die Länge von `Rest` ist.
Berechne
`Ergebnis = Produkt1 + GeschobenProdukt2`
Liefere `Ergebnis`

Fall 2: `Multiplikator` ist keine Ziffer (`Multiplikator` enthält also mindestens 2 Ziffern).

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:
`Multiplikator = Z Rest`

Berechne (mit dem Algorithmus Fall 1):
`Produkt1 = Z x Multiplikand`

Berechne (mit demselben Algorithmus Fall 1 oder 2, je nach dem, wie lang Rest ist):
`Produkt2 = Rest x Multiplikand`

Berechne
`GeschobenProdukt1 = Produkt1` gefolgt von n Nullen,
wobei n die Länge von Rest ist.

Berechne
`Ergebnis = GeschobenProdukt1 + Produkt2`

Liefere `Ergebnis`

Die Bezeichner `Z`, `Rest`, `Produkt1`, `Produkt2` und `Ergebnis` sind in beiden Fälle mit unterschiedlichen Bedeutungen verwendet worden. Ist das zulässig? Ja, weil die Geltungsbereiche dieser Bezeichner (d.h. die Fälle) sich nicht überlappen.

Einen solchen Gebrauch von Bezeichnern kennt man aus dem tagtäglichen Leben, z.B. wenn verschiedene Gebrauchsanweisungen mit demselben Bezeichner, z.B. „A“, Knöpfe oder sonstige Teile von verschiedenen Geräten benennen.

Ist der Wert eines Bezeichners wie Produkt1 eindeutig, wenn derselbe Algorithmus Bezug auf sich selbst (oder auf Teilalgorithmen von sich) nimmt? Ja, weil die Geltungsbereiche dieser Bezeichner (d.h. die Fälle) sich auch bei solchen „Wiederverwendungen“ nicht überlappen.

1.1.2 Beispiel einer Anwendung des Falles 1

`Multiplikator = 2`, `Multiplikand = 94`

Der Multiplikationsalgorithmus aus Abschnitt 1.1.1 läuft wie folgt ab:

```
2 x    9 4
   -------
         8
   + 1 8 0  (180 ist 2 x 9 gefolgt von 1 Null)
   -------
     1 8 8
```

Der Fall 1.2 trifft zu (`Multiplikand = 94` ist keine Ziffer).

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass: `Multiplikand = Z Rest`

Also `Z = 9`, `Rest = 4`

Berechne (mit demselbem Algorithmus Fall 1): `Produkt1 = Multiplikator x Rest`

Also `Produkt1 = 2 x 4`

Hier wird die Berechnung von `2 x 94` unterbrochen und (mit demselbem Algorithmus Fall 1) `2 x 4` berechnet. Zur Verdeutlichung dieser Nebenberechnung, wird sie eingerückt aufgeführt:
`Multiplikator = 2`, `Multiplikand = 4`
Der Fall 1.1 trifft zu (`Multiplikand = 4` ist eine Ziffer)
Die Multiplikationstabelle liefert den Wert `Ergebnis = Multiplikator x Multiplikand` d.h. `Ergebnis = 8`
Liefere `Ergebnis`

Da das Ergebnis von der Nebenberechnung geliefert wurde, ist diese Nebenberechnung nun beendet und die Hauptberechnung kann fortgesetzt werden. Aus der Nebenberechnung ergibt sich:

Produkt1 = 8

Ermittle aus der Multiplikationstabelle:

```
Produkt2 = Multiplikator x Z = 2 x 9
```

Also

```
Produkt2 = 18
```

Berechne

`GeschobenProdukt2 = Produkt2` gefolgt von n Nullen wobei n die Länge von `Rest`, d.h. `1` ist.

Also `GeschobenProdukt2 = 180`

Berechne

```
Ergebnis = Produkt1 + GeschobenProdukt2
```

Also `Ergebnis = 8 + 180 = 188`

Liefere `Ergebnis`.

1.1.3 Rekursion und Terminierung

Algorithmen wie der obige Algorithmus, die Bezug auf sich selbst oder auf Teilalgorithmen von sich nehmen, nennt man „rekursiv“. Die dabei verwendete Technik nennt man „Rekursion“. Die Rekursion ist eine zentrale Technik der Informatik.

Manche rekursiven Algorithmen wirken befremdlich, obwohl die Rekursion im Grunde eine wohlvertraute Technik zur Beschreibung von Verfahren ist.

Ein Viereck kann z.B. wie folgt auf den Boden gezeichnet werden:

Zeichne eine Seite wie folgt:
Gehe 3 Schritte nach vorne und zeichne dabei eine Linie;
Wende Dich dann um 90 Grad nach rechts.
Wenn Du nicht am Startpunkt stehst, dann rufe denselben Algorithmus auf
(d.h. zeichne eine Seite unter Anwendung des hier geschilderten Verfahrens).

Dieses Beispiel eines rekursiven Algorithmus stört uns nicht, weil

1. das Verfahren überschaubar ist,
2. die Terminierungsbedingung leicht erkennbar ist.

Bei rekursiven Algorithmen ist es meistens nicht trivial festzustellen, ob der Algorithmus terminiert. Ohne die Bedingung „wenn Du nicht am Startpunkt stehst“ würde der Viereck-Algorithmus seinen Zweck erfüllen, d.h. sich zum Zeichnen eines Vierecks eignen, aber nicht terminieren.

Frage: Kann man sich vergewissern, dass der Multiplikationsalgorithmus aus Abschnitt 1.1.1 terminiert?

Antwort: Siehe die mit ** markierten Zeilen in der folgenden Darstellung des Algorithmus.

Beweis:

Fall 1: `Multiplikator` ist eine Ziffer

Fall 1.1: `Multiplikand` ist eine Ziffer. Ermittle aus der Multiplikationstabelle:
`Ergebnis = Multiplikator x Multiplikand`.
Liefere `Ergebnis`
** Der Fall 1.1 terminiert offenbar.

Fall 1.2: `Multiplikand` ist keine Ziffer.
Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:
`Multiplikand = Z Rest`
** Die Ermittlung von `Z` und `Rest` terminiert offenbar.
** `Rest` ist echt kürzer als `Multiplikand` (*)
Berechne (mit demselben Algorithmus Fall 1):
`Produkt1 = Multiplikator x Rest`
** Wegen (*) sind die Multiplikanden der rekursiven Anwendungen
** des Falles 1 immer echt kürzer, so dass letztendlich der
** (terminierende!) Fall 1.1 eintreten wird. Die obige rekursive
** Berechnung terminiert also.
Ermittle aus der Multiplikationstabelle:
`Produkt2 = Multiplikator x Z`
** Terminiert offenbar.
Berechne
`GeschobenProdukt2 = Produkt2` gefolgt von n Nullen
wobei n die Länge von `Rest` ist.
** Terminiert offenbar.
Berechne
`Ergebnis = Produkt1 + GeschobenProdukt2`
** Unter der Annahme, dass ein terminierender Algorithmus zur
** Addition zweier natürlichen Zahlen zur Verfügung steht,
** terminiert diese Berechnung
Liefere `Ergebnis`

Fall 2: `Multiplikator` ist keine Ziffer (`Multiplikator` enthält also mindestens 2 Ziffern)
Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:
`Multiplikator = Z Rest` (**)
** Terminiert offenbar.
Berechne (mit dem Algorithmus Fall 1):
`Produkt1 = Z x Multiplikand`
** Der Fall 1 terminiert (oben festgestellt).
Berechne (mit demselben Algorithmus Fall 1 oder 2, je nach dem, wie lang `Rest` ist):
`Produkt2 = Rest x Multiplikand`

** Wegen (*) und (**) ist bei jeder rekursiven Anwendung entweder
** der Multiplikand, oder der Multiplikator immer echt kürzer als
** der Multiplikand oder Multiplikator der vorherigen Anwendung, so
** dass letztendlich der (terminierende!) Fall 1.1 eintreten wird. Die
** obige rekursive Berechnung terminiert also.

Berechne
`GeschobenProdukt1 = Produkt1` gefolgt von n Nullen
wobei n die Länge von `Rest` ist.

** Terminiert offenbar.

Berechne
Ergebnis = GeschobenProdukt1 + Produkt2

** Unter der Annahme, dass ein terminierender Algorithmus zur
** Addition zweier natürlichen Zahlen zur Verfügung steht,
** terminiert diese Berechnung

Liefere Ergebnis.

qed.

1.1.4 Kritik an der Algorithmusbeschreibung aus Abschnitt 1.1.1

1. Sie ist nicht ganz präzise: Sie definiert die Geltungsbereiche von Variablen wie `Z` und `Rest` nicht explizit, so dass es unklar sein kann, was genau gemeint ist. Sie ist lang und ziemlich unklar.

2. Sie eignet sich kaum für einen präzisen Beweis der Terminierung oder der Korrektheit des Verfahrens.

Aus diesen Gründen werden zur Beschreibung von solchen Verfahren, „Algorithmen“ genannt, formale Spezifikationen bevorzugt.

1.1.5 Zum Begriff „Algorithmus“

In dem Buch (Seiten 1-2)

> Jean-Luc Chabert et al. A history of algorithms - From the pebble to the microchip. Springer Verlag, ISBN 3-540-63369-3, 1999

wird folgende Erläuterung gegeben:

> „Algorithms have been around since the beginning of time and existed well before a special word had been coined to describe them. Algorithms are simply a set of step by step instructions, to be carried out quite mechanically, so as to achieve some desired result. [...] Algorithms are not confined to mathematics [...]. The Babylonians used them for deciding points of law, Latin teachers used them to get the grammar right, and they have been used in all the cultures for predicting the future, for deciding medical treatment, or for preparing food. Everybody today uses algorithms of some sort or another, often unconsciously, when following a recipe, using a knitting pattern, or operating household gadgets. [...] Today, principally because of the influence of computing, the idea of finiteness has entered into the meaning of algorithm as an essential element, distinguishing it from vaguer notions such as process, method, or technique.“

Das Wort „Algorithmus“ stammt von dem (arabischen) Namen eines berühmten iranischen Mathematikers der ersten Hälfte des 9. Jahrhunderts nach Christus, Muhammad ibn Musa al-Khwarzem, d.h. Muhammad Sohn des Musa von Khwarzem. Khwarzem ist der Name einer Region in Zentralasien südlich des Aral-Meeres. Der Titel eines Buches dieses Mathematikers, „al-Mukhtaar f Hisb al-Jabr wa l-Muqbala“, lieferte übrigens das Wort „Algebra“.

Definition (Intuitive Definition des Algorithmusbegriffs)

Ein Algorithmus ist ein Verfahren mit einer *präzisen* (d.h. in einer genau festgelegten Sprache formuliert), *endlichen Beschreibung* unter Verwendung *effektiver* (d.h. tatsächlich ausführbarer) *elementarer Verarbeitungsschritte*.

Zu jedem Zeitpunkt der Abarbeitung des Algorithmus benötigt der Algorithmus nur *endlich viele Ressourcen*.

Wichtige Eigenschaften von Algorithmen:

- Ein Algorithmus heißt *terminierend*, wenn er für alle zulässigen Schrittfolgen stets nach endlich vielen Schritten endet.
- Er ist *deterministisch*, wenn in der Auswahl der Verarbeitungsschritte keine Freiheit besteht.
- Er ist *determiniert*, wenn das Resultat eindeutig bestimmt ist.
- Er wird *sequenziell* genannt, wenn die Schritte stets hintereinander ausgeführt werden.
- Er heißt *parallel* (*nebenläufig*), wenn gewisse Verarbeitungsschritte nebeneinander (im Prinzip gleichzeitig) ausgeführt werden.

1.1.6 Formale Spezifikation eines Algorithmus

Unter Verwendung der Begriffe „totale“ und „partielle Funktion“ kann der Algorithmus aus Abschnitt 1.1.1 präzise spezifiziert werden.

Definition Definition (Totale und partielle Funktion):

- Eine totale Funktion, kurz Funktion, $f : A \to B$ (von A nach/in B) ist eine Teilmenge des Kartesischen Produkts $A \times B$, so dass es für jedes $a \in A$ ein eindeutiges $b \in B$ gibt mit $(a, b) \in f$.
  Eindeutig bedeutet hier: wenn $(a, b_1) \in f$ und $(a, b_2) \in f$, dann $b_1 = b_2$.
  Man sagt: eine totale Funktion ist eine linkstotale und rechtseindeutige binäre (2-stellige) Relation.
- Eine partielle Funktion $f : A \to B$ ist eine Teilmenge des Kartesischen Produkts $A \times B$, so dass es $A' \subset A$ gibt mit:
  - $A' = \{\ a \mid \text{es gibt } b \in B : (a, b) \in f\ \}$
  - $\{\ (a, b) \mid (a, b) \in f\ \}$ ist eine totale Funktion von A' in B.

  Man sagt: eine partielle Funktion ist eine rechtseindeutige binäre (2-stellige) Relation.

Spricht ein Mathematiker von einer Funktion ohne anzugeben, ob sie total oder partiell ist, dann meint er in der Regel eine totale Funktion. Spricht aber ein Informatiker von einer Funktion ohne anzugeben, ob sie total oder partiell ist, dann meint er in der Regel eine partielle Funktion!

Legen wir zunächst fest, welche Typen von Daten verwendet werden. Dies ist aus der informellen Spezifikation aus Abschnitt 1.1.1 leicht erkennbar:

`digit:` (Ziffer)

Ein „digit“ ist ein Element aus $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$.

`integer:` (ganze Zahl; hier nur nichtnegative ganze, also natürliche Zahlen)

Ein „integer“ ist eine beliebige, nicht-leere, endliche Sequenz von „digits“. Führende Nullen sind also *nicht* ausgeschlossen.

`boolean:` (Wahrheitswert)

`true` oder `false`

```
function
(integer,integer) ---> integer
integer_mult(multiplicator, multiplicand) =
   if one_digit_integer(multiplicator)
   then (* Fall 1 *) integer_digit_mult(multiplicator, multiplicand)
   else (* Fall 2 *)
        (let product1 = integer_digit_mult(head(multiplicator), multiplicand)
             product2 = integer_mult(tail(multiplicator), multiplicand)
         in
         integer_add(shift(tail(multiplicator), product1), product2)
        )
end
```

```
function
(digit, integer) ---> integer
integer_digit_mult(multiplicator, multiplicand) =
     if one_digit_integer(multiplicand)
     then (* Fall 1.1 *)
          digit_mult(multiplicator, multiplicand)
     else (* Fall 1.2 *)
          (let product1 = integer_digit_mult(multiplicator, tail(multiplicand))
              product2 = digit_mult(multiplicator, head(multiplicand))
           in
           integer_add(product1, shift(tail(multiplicand), product2))
          )
end
```

Verwendete Funktionen:

`one_digit_integer: integer` → `boolean`	liefert true, wenn das Argument ein 1-stelliger `integer`-Wert ist, sonst false.
`digit_mult: (digit, digit)` → `integer`	die Multiplikationstabelle
`head: integer` → `digit`	liefert das am weitesten links stehende digit eines integer
`tail: integer` → `integer`	liefert das integer, das sich ergibt, wenn das am weitesten links stehende digit des Arguments entfernt wird.
`integer_add: (integer, integer)` → `integer`	liefert die Summe zweier integer
`shift: (integer, integer)` → `integer`	ergänzt das 2. Argument nach rechts mit so viel Nullen, wie das 1. Argument digits enthält

Die Funktion `tail` ist nicht total, weil sie z.B. für „`2`“ nicht definiert ist. Eine leere Sequenz ist nach Annahme kein `integer`-Wert. Die sonstigen verwendeten Funktionen sind total.

Der Formalismus, in dem die Funktionen `integer_mult` und `integer_digit_mult` spezifiziert sind, erinnert an eine Programmiersprache. Dies liegt vor allem an der Schreibweise, die z.B. keine Indizes und ausdruckskräftige Namen für Funktionen verwendet. In der Tat liegt hier keine Programmiersprache vor, sondern nur ein Formalismus zur Algorithmenbeschreibung. Dieser Formalismus ist präziser als der Formalismus vom Abschnitt1.1.1, u.a. weil die Geltungsbereiche der lokalen Variablen mit expliziten Klammerungen definiert sind.

Die folgenden Begriffe werden in der Vorlesung eingeführt:

- Funktionsdefinition und -aufruf
- Formale Parameter und (Aufruf-)Parameter
- Globale und lokale Variablen
- Datentyp

1.1.7 Eigenschaften eines Algorithmus: Partielle und totale Korrektheit

Eine wünschenswerte Eigenschaft eines Algorithmus (z.B. des Multiplikationsalgorithmus vom Abschnitt 1.1.1) ist seine „Korrektheit“.

Was heißt aber, dass der Multiplikationsalgorithmus korrekt ist? Um korrekt zu sein, darf ein Algorithmus offensichtlich nie ein falsches Ergebnis liefern.

Dieses Kriterium ist aber nicht ausreichend für die Korrektheit. Die folgende Spezifikation der Multiplikation erfüllt sicherlich das Kriterium, dass sie nie ein falsches Ergebnis liefert! Trotzdem würde man diese Spezifikation kaum als „korrekt“ im intuitiven Sinn ansehen:

```
function
(integer, integer) ---> integer
mult(x, y) = (mult(y, x))
```

Man unterscheidet wie folgt zwischen „partieller“ und „totaler Korrektheit”:

1. Partielle Korrektheit:

   Liefert der betrachtete Algorithmus ein Ergebnis, so ist es das erwartete/richtige Ergebnis

   (Bsp.: liefert `mult(a, b)` einen Wert `c`, so gilt `c = a * b`)

2. Totale Korrektheit:

   Der Algorithmus terminiert für alle möglichen Eingaben und liefert jeweils das erwartete/richtige Ergebnis

   (Bsp.: für alle natürliche Zahlen `a` und `b` terminiert `mult(a,b)` und liefert das Produkt `a * b`)

1.1.8 Beweisprinzip der vollständigen Induktion

Der (informelle!) Terminierungsbeweis vom Abschnitt 1.1.3 wendet das Beweisprinzip der vollständigen Induktion an. Aufgrund der Wichtigkeit von Terminierungsbeweisen nimmt dieses Beweisprinzip einen zentralen Platz in der Informatik ein.

Definition Definition (Beweisprinzip der vollständigen Induktion):
Sei $f : \mathbb{N} \to M$ eine (totale) Funktion von den natürlichen Zahlen in eine Menge M. Sei B die Menge $\{f(n) \mid n \in \mathbb{N}\}$.
Um zu zeigen, dass jedes Element b von B eine Eigenschaft E besitzt, genügt es zu zeigen:

1. Induktionsbasis: $f(0)$ besitzt die Eigenschaft E.
2. Induktionsschritt: Sei n eine beliebige natürliche Zahl. Wenn $f(n)$ die Eigenschaft E besitzt, so auch $f(n+1)$.

1.1.9 Programm

Eine formale Spezifikation eines Algorithmus, wie im Abschnitt 1.1.6 gegeben, ist aus den folgenden Gründen kein Programm:

1. Manche verwendete Funktionen, wie `integer_add`, sind nur (formal oder informell) spezifiziert, aber nicht implementiert.
2. Die verwendeten Datenstrukturen sind nur spezifiziert, nicht implementiert, d.h. nicht unter Anwendung einer Programmiersprache auf Speicherstrukturen eines Computers abgebildet.

Aus den folgenden Gründen sind formale Spezifikationen wichtig und deren Entwicklung voreiligen Programmentwürfen vorzuziehen:

1. Jede Programmiersprache besitzt Eigenschaften, die in manchen Fällen die Entwicklung eines einfachen Algorithmus erschweren.
2. Die Datentypen einer Anwendung (hier: `digit`, `integer`) können oft in einer Programmiersprache erst dann besser implementiert werden, wenn ihre Verwendung einmal vollständig und präzis festgelegt wurde.
3. Programmiersprachen wechseln, Algorithmen bleiben.

1.1.10 Eigenschaften eines Algorithmus: Zeit- und Speicherplatzkomplexität

Die Komplexität eines Algorithmus wird nach seinem Zeit- und Speicherbedarf ermittelt. Der Multiplikationsalgorithmus `integer_digit_mult` aus Abschnitt 1.1.1 verläuft wie folgt:

```
multiplicator x multiplicand
      (digit)      (integer)
              --------------
                    product1  = integer_digit_mult(multiplicator,
                                                   tail(multiplicand))
                              [ product2 =
                                  digit_mult(multiplicator,
                                             head(multiplicand)) ]
 + shift(tail(multiplicand),
                   product2)
  -------------------------
                     result
```

Der benötigte Speicherplatz für die einzelnen Werte entspricht der Länge der jeweiligen Werte:

Länge von `product1` $\leq$ Länge von `tail(multiplicand)` + 1
= Länge von `multiplicand`
Länge von `product2` ≤ 2

Länge von `shift(tail(multiplicand), product2)`
$\leq$ 2 + Länge von `multiplicand` - 1
$<$ 2 + Länge von `multiplicand`
Länge von `result` $\leq$ 2 + Länge von `multiplicand`
max. Speicherbedarf: $\leq$ 1 + 2 * Länge von `multiplicand`

Der Speicherbedarf der Funktion `integer_digit_mult` wächst also linear $(1 + 2x)$ in der Länge $(= x)$ ihres 2. Arguments.

Die Ermittlung des Zeitbedarfs verlangt, dass Zeiteinheiten festgelegt werden. z.B.:

1 Look-up („Nachschlagen“) in der Multiplikationstabelle	=	1 Zeiteinheit
Addition zweier digits	=	1 Zeiteinheit

Dann ergibt sich für die Funktion `integer_digit_mult` der folgende Zeitbedarf:

- so viele Look-ups in der Multiplikationstabelle wie `multiplicand` lang ist
- Wegen der Überträge:
  Anzahl der Additionen zweier digits $\leq$ (Länge von `multiplicand` - 1)

Der Zeitbedarf der Funktion `integer_digit_mult` wächst also höchstens linear $(2x - 1)$ in der Länge $(= x)$ ihres 2. Arguments.

1.2 Was ist Informatik?

Die Frage ist schwer zu beantworten, weil die Informatik sich nicht durch ihre Anwendungsgebiete definieren lässt. Sie erschließt immer neue Anwendungsgebiete — insbesondere derzeit.
Das Wort „Informatik“ wurde 1962 von dem (französischen) Ingenieur Philippe Dreyfus vorgeschlagen. Die Wortschöpfung wurde aus „Information“ und „Elektronik“ gebildet. Im Englischen werden die Bezeichnungen „computing science“ und „computer science“ verwendet. „Computing science“ ist sicherlich eine zutreffende Bezeichnung, wenn „Rechnen“ im allgemeinen mathematischen Sinne verstanden wird — also nicht auf Numerik eingeschränkt.
Einige mögliche Definitionen für Informatik:

Definition (Informatik) [DUDEN Informatik]:

Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Computern.

Definition (Informatik)
[Gesellschaft für Informatik e.V., www.gi-ev.de;
Studien- und Forschungsführer Informatik, Springer-Verlag]:

Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.

Definition (Computer Science)
[Association for Computing Machinery, www.acm.org]:

Computer Science is the systematic study of algorithms and data structures, specifically (1) their formal properties, (2) their mechanical and linguistic realizations, and (3) their applications.

Es ist wichtig zu betonen, dass Informatik und Programmierung nicht gleich sind. Die Informatik setzt die Programmierung voraus, ist aber viel mehr als nur Programmierung.

Während des Grundstudiums werden die Hauptbereiche der Informatik eingeführt, so dass die Frage „Was ist Informatik?“ immer präziser zu beantworten sein wird.

Die Informatik verändert sich extrem schnell — wie kaum ein anderes Fach. Was die Informatik ist, wird auch von Ihnen bestimmt.

1.3 Die Programmiersprache der Vorlesung

Diese Vorlesung ist eine Einführung in die Informatik und in die Programmierung. Dazu ist eine gut konzipierte Programmiersprache vorteilhaft. Da die Programmiersprachen aus der Industrie erhebliche Mängel haben, die das Erlernen der Programmierung wesentlich erschweren, wurde für diese Vorlesung eine Programmiersprache aus der Forschung gewählt: SML (Standard ML).

Die ersten Entwürfe, die zu SML führten, stammen vom Ende der 70er Jahre. SML in der Form, die in dieser Vorlesung verwendet wird, wurde erst 1986 entwickelt. Es handelt sich also um eine Programmiersprache, die alt genug ist, um von Mängeln bereinigt zu sein, aber auch jung genug ist, um eine zeitgemäße Ausbildung zu ermöglichen.

SML ist eine sogenannte Funktionale Programmiersprache, d.h. dass ein SML-Programm als Sammlung von Gleichungen verstanden werden kann. Nicht alle Programmiersprachen sind funktional. Weitere sogenannte Berechnungsparadigmen, worauf Programmiersprachen beruhen, sind:

- das imperative Paradigma,
- das logische Paradigma,
- das objektorientierte Paradigma.

Manche Programmiersprachen kombinieren Aspekte aus verschiedenen Paradigmen.

Es ist wünschenswert, dass Sie während des Grundstudiums möglichst allen Paradigmen und mehreren Programmiersprachen begegnen. In der Vorlesung Informatik 2 werden Sie die imperative und objekt-orientierte Programmierung mit der (aus der Industrie stammenden) Programmiersprache JAVA kennenlernen. Es empfiehlt sich, dass jeder Student eventuell selbständig mindestens eine weitere Programmiersprache während des Grundstudiums lernt.

Es wird oft gefragt, wie viele Programmiersprachen es gibt. Die Antwort lautet: Tausende! Man kann sie also nicht alle lernen. Ein Informatiker muss dazu in der Lage sein, sich neue Programmiersprachen in kürzester Zeit selbst anzueignen. Hierzu ist es essentiell, die immer wiederkehrenden Konzepte zu kennen, die praktisch allen Programmiersprachen zugrunde liegen.

1.4 Inhaltsverzeichnis der Vorlesung

1. Einleitung
2. Einführung in die Programmierung mit SML
3. Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)
4. Prozedur zur Abstraktionsbildung
5. Die vordefinierten Typen von SML
6. Typprüfung
7. Abstraktionsbildung mit Prozeduren höherer Ordnung
8. Abstraktionsbildung mit neuen Datentypen
9. Pattern Matching
10. Auswertung und Ausnahmen
11. Bildung von Abstraktionsbarrieren mit abstrakten Typen und Moduln
12. Imperative Programmierung in SML
13. Formale Beschreibung der Syntax und Semantik von Programmiersprachen

1.5 Literatur

Diese Vorlesung wird in Anlehnung an die Vorlesung „Informatik 1“ von M. Wirsing im Wintersemester 1999/2000 und insbesondere an die von F. Bry im Wintersemester 2000/2001 gestaltet, um etwaige Prüfungswiederholungen zu erleichtern. Sie ist aber sicherlich nicht vollständig identisch. Für eine Prüfungswiederholung ist aber in jedem Fall eine aktive Teilnahme — vor allem an den Übungen — empfehlenswert.

Die folgenden Schriften können als Grundliteratur zur Programmiersprache SML verwendet werden:

- Robert Harper:

  Programming in Standard ML.
  http://www.cs.cmu.edu/People/rwh/introsml/index.htm

- Michael R. Hansen, H. Rischel:

  Introduction to Programming using SML, Addison-Wesley, 1999
  ISBN 0-201-39820-6 (paperback)
  http://www.it.dtu.dk/introSML/

- Lawrence Paulson:

  ML for the Working Programmer (Second Edition), MIT Press, 1996
  ISBN 0 521 57050 6 (hardback)
  ISBN 0 521 56543 X (paperback)
  http://www.cl.cam.ac.uk/users/lcp/MLbook/

  [Studenten mit wenig Programmiererfahrung sollten das Kapitel 1 überspringen.]

  Die Programme aus dem Buch finden sich unter:
  http://www.cl.cam.ac.uk/users/lcp/MLbook/programs/

Weitere Literaturhinweise insbesondere auch zur Einführung in die Informatik finden sich auf der Webseite der Vorlesung:

http://www.dbs.informatik.uni-muenchen.de/Lehre/Info1/

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 2

Einführung in die Programmierung mit SML

In diesem Kapitel wird am Beispiel der funktionalen Programmiersprache SML in die Programmierung eingeführt. Ziel ist es, dass jeder Student sich die für ein Informatikstudium unabdingbaren Programmierfertigkeiten schnell aneignet. Das Programmieren kann als Handwerkzeug der Informatik betrachtet werden. Programmierung und Programmiersprachen können nur mit viel Übung gelernt werden. Die Vorlesung kann nur die zentralen Konzepte und Herangehensweisen vermitteln; üben muss aber jeder Student für sich selbst.

SML bedeutet „Standard ML“. Wie die meisten Programmiersprachen entstand das heutige SML aus Verfeinerungen einer ursprünglichen Programmiersprache. Die ursprüngliche Programmiersprache, aus der SML entstand, hieß ML (was ursprünglich für Meta Language stand). Heute bezeichnen beide Namen, SML und ML, dieselbe Programmiersprache. Dazu gibt es ein paar ML-Dialekte, die in einigen Aspekten von SML abweichen.

2.1 Antipasti

In einem italienischen Essen sind die Antipasti eine mehr oder weniger große Sammlung von kleinen schmackhaften Appetitanregern, die am Anfang des Essens angeboten werden. In ähnlicher Weise werden in diesem Abschnitt einige — hoffentlich schmackhafte — Aspekte von SML vermittelt.

SML wird mit dem Linux-Kommando `sml` aufgerufen. Eine SML-Sitzung wird mit `^D` (`Ctrl` + `D` bzw. `Strg` + `D` gleichzeitig) beendet.

SML bietet eine interaktive, d.h. dialogorientierte, Benutzerschnittstelle, die auf einer Treiberschleife beruht. Wenn die Treiberschleife das Zeichen „`-`“ am Anfang einer Zeile anzeigt, dann kann der Benutzer einen Ausdruck angeben, das Ende des Ausdrucks mit dem Zeichen „`;`“ kennzeichnen und die Auswertung des Ausdruckes mit „enter“ (auch „return“ genannt) anfordern. SML wertet dann den Ausdruck unter Verwendung der zu der Zeit bekannten Definitionen aus und liefert den ermittelten Wert in einer neuen Zeile.

Beispiel einer SML-Sitzung:

```
linux% sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
```

```
- 2;
val it = 2 : int

- ~1;
val it = ~1 : int

- ~(~2);
val it = 2 : int

-  2 * 3 + 1;
val it = 7 : int

- (2 * 3) + 1;
val it = 7 : int

- 2 * (3+1);
val it = 8 : int

- 4 div 2;
val it = 2 : int

- 5 div 2;
val it = 2 : int

- 100 mod 4;
val it = 0 : int

- 012;
val it = 12 : int

- Ctrl+D
linux%
```

2.1.1 Der Datentyp „ganze Zahl“

„`int`“ bezeichnet den (Daten-)Typ „integer“ oder „ganze Zahl“. In SML wie in den meisten modernen Programmiersprachen besitzt jeder Ausdruck einen (Daten-)Typ. Typen sind wichtige Bestandteile von Programmiersprachen. Die vordefinierte Typen von SML werden im Kapitel 6 eingeführt.

„`it`“ bezeichnet den unbenannten Wert des Ausdrucks, dessen Auswertung angefordert wird. Wir werden sehen, dass Werte von Ausdrücken an weitere Namen gebunden werden können.

In diesem Kapitel werden vorwiegend Ausdrücke behandelt, deren Werte ganze Zahlen sind, also Ausdrücke vom Typ „ganze Zahl“. Neben dem Typ „ganze Zahl“ wird auch der Typ „boolean“, Boole'scher Wert, in diesem Kapitel eingeführt. Der Typ „reelle Zahl“ wird auch in diesem Kapitel kurz erwähnt, wird aber erst im Kapitel 6 ausführlich behandelt.

In SML sind bei ganzen Zahlen führende Nullen zulässig. Zum Beispiel ist `012` eine andere

Notation für `12`, `~0012` eine andere Notation für `~12`.

SML bietet die folgenden vordefinierten Operationen über natürlichen Zahlen: `+`, `-`, `*`, `div` und `mod`, die alle infix notiert werden.

Auf die sogenannten „Präzedenzen“ der vordefinierten Operationen muss geachtet werden: `2 * 3 + 1` steht z.B. für `(2 * 3) + 1`.

Vorsicht: `~` (Vorzeichen für negative ganze Zahlen) und `-` (Subtraktion) sind in SML nicht austauschbar.

`~` ist ein unärer (d.h. einstelliger) Operator (oder Operation). `-`, `+`, `*`, `div` und `mod` sind binäre (d.h. zweistellige) Operatoren.

2.1.2 Gleichheit für ganze Zahlen

Zum Vergleich von ganzen Zahlen bietet SML die vordefinierte Funktion „`=`“:

```
- 2 = 2;
val it = true : bool

- 2 = 3;
val it = false : bool
```

Eine Funktion, die wie `=` als Wert entweder `true` oder `false` liefert, wird *Prädikat* oder auch *Test* genannt.

2.1.3 Der Datentyp „Boole'scher Wert“

Die Werte der Ausdrücke `2 = 2` und `2 = 3` sind sogenannte „Wahrheitswerte“ oder Boole'sche Werte. Es gibt zwei Boole'sche Werte: „`true`“ (wahr) und „`false`“ (falsch).

SML bietet die folgenden Operationen über Boole'schen Ausdrücken: `not` (präfix notiert), `andalso` (infix notiert) und `orelse` (infix notiert).

```
- true;
val it = true : bool

- not true;
val it = false : bool

- not (not false);
val it = false : bool

- true andalso not false;
val it = true : bool

- false orelse true;
val it = true : bool

- not (2 = 3);
val it = true : bool
```

Der Ausdruck `not not false` kann nicht ausgewertet werden, weil er von SML wie `(not not) false` verstanden wird. `(not not) false` ist aus zwei Gründen inkorrekt:

1. Die Teilausdrücke `(not not)` und `false` sind nicht mit einer Operation verbunden. `(not not) false` ist also genauso sinnlos wie etwa `2 4`.
2. Der Teilausdruck `(not not)` ist inkorrekt gebildet, weil die erste Negation auf keinen Booles'schen Ausdruck angewandt wird. `(not not)` ist genauso sinnlos wie etwa (~ ~). (Übrigens ist `~ ~ 5` aus den gleichen Gründen inkorrekt.)

`not` ist ein unärer Operator (oder Operation). `orelse` (Disjunktion) und `andalso` (Konjunktion) sind binäre Operatoren (oder Operationen).

2.1.4 Gleichheit für Boole'sche Werte

Boole'sche Ausdrücke können mit der vordefinierten Funktion `=` verglichen werden:

```
- true = not (not true);
  val it = true : bool

- false = (false andalso not true);
  val it = true : bool
```

Bemerkung:
Allerdings ist der Vergleich von Wahrheitswerten mit `=` fast immer schlechter Programmierstil und unnötig kompliziert. Besonders Anfänger neigen oft zu Konstruktionen wie:

```
if Bedingung = true then Ausdruck else Ausdruck'
```

ohne zu beachten, dass `Bedingung = true` immer genau denselben Wert hat wie `Bedingung`. Es ist einfacher und übersichtlicher,

```
if Bedingung then Ausdruck else Ausdruck'
```

zu schreiben. In ähnlicher Weise lässt sich

```
if Bedingung = false then Ausdruck else Ausdruck'
```

stets vereinfachen zu

```
if not Bedingung then Ausdruck else Ausdruck'
```

oder noch besser (?) zu

```
if Bedingung then Ausdruck' else Ausdruck
```

2.1.5 Überladen

Obwohl die Gleichheit für Boole'sche Ausdrücke und die Gleichheit für ganze Zahlen identisch geschrieben werden, handelt es sich um grundverschiedene Funktionen, weil ihre Argumente verschiedene Typen besitzen. Zur Feststellung der Gleichheit, um die es sich in einem Ausdruck handelt, zieht das SML-System die Typen der Operanden in Betracht.

Wenn derselbe Name oder dasselbe Symbol, wie hier `=`, zur Bezeichnung unterschiedlicher Operationen oder Funktionen verwendet wird, die vom System unterschieden werden, dann sagt man, dass der Name oder das Symbol „überladen" (overloaded) ist.

Das Überladen von Bezeichnern ist nicht ungefährlich und wird deswegen nur selten angewandt.

Weitere Fälle von Überladen in SML sind `+` und `*`, die die arithmetischen Operationen Addition und Multiplikation sowohl für ganze Zahlen als auch für reelle Zahlen bezeichnen:

```
- 2 + 3;
val it = 5 : int

- 2.1 + 3.3;
val it = 5.4 : real

- 2 * 3;
val it = 6 : int

- 2.1 * 3.3;
val it = 6.93 : real
```

Es ist wichtig zu bemerken, dass in SML sowie in den meisten Programmiersprachen ganze Zahlen und reelle Zahlen Zahlen unterschiedlicher Typen sind. Im Gegensatz dazu sind in der Mathematik ganze Zahlen ein Untertyp der reellen Zahlen. Der Unterschied kommt daher dass Programmiersprachen völlig unterschiedliche Darstellungen für ganze Zahlen und für reelle Zahlen benutzen (und dass diese „reellen Zahlen" im Sinne von Programmiersprachen auch gar nicht die Menge der „reellen Zahlen" im mathematischen Sinn repräsentieren, sondern nur eine Teilmenge der rationalen Zahlen). In SML wie in den meisten Programmiersprachen kennt die Ganzzahlarithmetik keine Rundung und damit auch keine Ungenauigkeit, wogegen die Genauigkeit der Arithmetik mit reellen Zahlen von der Gleitkommahardware des Computers abhängt.

Der Versuch, eine ganze Zahl und eine reelle Zahl zu addieren, führt folglich zu einer Fehlermeldung:

```
- 2 + 4.83;
stdIn:10.1-10.9 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    2 + 4.83
```

Wegen des ersten Operanden `2` interpretiert SML das Zeichen `+` als die Addition für ganze Zahlen. Da `4.83` keine ganze Zahl ist, wird ein Typfehler gemeldet.

2.1.6 Weitere Typen

SML bietet weitere häufig benötigte Typen wie z.B. „Zeichen“ (wie `a`, `b`, `c`, usw.) und „Zeichenfolge“ (wie diese Klammer) — siehe Kapitel 6.

Ferner ermöglicht SML die Definition von Typen, die für ein praktisches Problem maßgeschneidert werden können (wie etwa eine beliebige Notenskala oder die Tage der Woche in einer beliebigen Sprache) — siehe Kapitel 9.

2.1.7 Vergleichsfunktionen für ganze Zahlen und für reelle Zahlen

Für ganze Zahlen und für reelle Zahlen bietet SML die folgenden vordefinierten überladenen Prädikate an:

`<`	(echt kleiner)
`>`	(echt größer)
`<=`	(kleiner gleich)
`>=`	(größer gleich)

Für ganze Zahlen bietet SML das vordefinierte Prädikat `<>` (Negation der Gleichheit) an.

Vorsicht: `=` und `<>` sind für reelle Zahlen nicht zulässig. Für reelle Zahlen bietet SML die Funktion `Real.compare(x, y)` an:

```
- Real.compare(1.0,7.0);
val it = LESS : order

- Real.compare(100.0,1.0);
val it = GREATER : order

- Real.compare(1.0,1.0);
val it = EQUAL : order
```

Man beachte, dass die Funktion `Real.compare` kein Prädikat ist, weil sie keine Boole'schen Werte, sondern Werte eines bisher nicht behandelten Typs namens `order` liefert.

SML bietet auch die Gleichheitsfunktion `Real.==` für reelle Zahlen, die den Typ `order` nicht verwendet:

```
- Real.==(2.5, 2.5);
val it = true : bool

- Real.==(2.5, 3.0);
val it = false : bool
```

2.1.8 Weitere nützliche Funktionen für ganze Zahlen

`Int.abs`: Betrag einer ganzen Zahl

```
- Int.abs(~4);
val it = 4 : int
```

`Int.min`: Minimum zweier ganzen Zahlen

```
- Int.min(5,2);
val it = 2 : int
```

`Int.max`: Maximum zweier ganzen Zahlen

```
- Int.max(3,5);
val it = 5 : int
```

`Int.sign`: „Vorzeichen“ einer ganzen Zahl

```
- Int.sign(0);
val it = 0 : int

- Int.sign(~5);
val it = ~1 : int

- Int.sign(6);
val it = 1 : int
```

2.2 Ausdrücke, Werte, Typen und polymorphe Typüberprüfung

2.2.1 Ausdrücke, Werte und Typen

Wie bereits erwähnt, wertet SML Ausdrücke aus. Ein Ausdruck kann atomar wie „`2`“, „`4`“ oder „`false`“ sein, oder zusammengesetzt wie `2 + 4` und `not (false andalso true)`.

Jeder korrekt gebildete Ausdruck hat einen Typ. Ein Typ ist eine Menge von Werten, zum Beispiel die Menge der ganzen Zahlen. Meistens hat ein Ausdruck auch einen Wert, und wenn der Ausdruck einen Wert hat, ist der Wert ein Element des Typs des Ausdrucks. Manche Ausdrücke wie `1 div 0`, in denen nicht-totale Funktionen verwendet werden, haben keinen Wert.

Atomare Ausdrücke wie `true` und `false` kann man oft mit ihren Werten identifizieren. Aber in vielen, auch ziemlich einfachen Fällen ist diese Betrachtungsweise problematisch. So lässt SML zum Beispiel bei ganzen Zahlen (Typ `int`) führende Nullen zu. Also sind `02` und `2` verschiedene atomare Ausdrücke, die aber beide denselben Wert haben. Hier sind atomare Ausdrücke und Werte nicht identisch. Ein anderes Beispiel sind atomare Ausdrücke wie `div` oder `orelse`, deren Werte selbstverständlich Funktionen sind, also nicht mit den atomaren Ausdrücken identisch. Zusammengesetzte Ausdrücke sind mit ihren Werten auf keinen Fall identisch. Aus all diesen Gründen wird üblicherweise strikt zwischen Ausdrücken und Werten unterschieden.

Jeder korrekt gebildete Ausdruck hat einen Typ (aber nicht immer einen Wert). Bisher sind wir nur Ausdrücken der Typen „ganze Zahl“, „Boole'scher Wert“ und „reelle Zahl“ begegnet.

Auch Operationen und allgemein Funktionen haben Typen: `+` z.B. ist eine Funktion, die als Argumente zwei (atomare oder zusammengesetzte) Ausdrücke vom Typ „ganze

Zahl“ erhält und einen Wert ebenfalls vom Typ „ganze Zahl“ liefert. Die Gleichheit für ganze Zahlen ist eine Funktion, die als Argumente zwei (atomare oder zusammengesetzte) Ausdrücke vom Typ „ganze Zahl“ erhält und einen Wert vom Typ „Boole'scher Wert“ liefert. Man schreibt:

```
+  :  (int, int)  →  int
=  :  (int, int)  →  bool
```

Bei der Bildung von zusammengesetzten Ausdrücken muss immer auf die Typen der verwendeten Operationen oder Funktionen und der eingesetzten Teilausdrücke geachtet werden.

2.2.2 Typen in Programmiersprachen

Es gibt zwei grundlegende Ansätze, was Typen in Programmiersprachen angeht:

- Schwach typisierte Programmiersprachen (z.B. Prolog, Lisp)
- Stark (oder streng) typisierte Programmiersprachen (z.B. Pascal, Modula)

Eine schwach typisierte Programmiersprache würde einen Ausdruck wie `8.0 + 1` (Summe einer reellen Zahl und einer ganzen Zahl) akzeptieren und bei der Auswertung die natürliche Zahl `1` in eine reelle Zahl automatisch umwandeln — man spricht von einer (automatischen) „Typanpassung“.

Eine stark typisierte Programmiersprache verlangt vom Programmierer, dass er für jeden Namen (oder Bezeichner) einen Typ explizit angibt und jede notwendige Typanpassung selbst programmiert. In SML kann eine Typanpassung zwischen reellen und ganzen Zahlen unter Verwendung der vordefinierten Funktion `real` und `round` wie folgt programmiert werden:

```
- real(1);
val it = 1.0 : real

- round(8.12);
val it = 8 : int

- round(8.99);
val it = 9 : int

- round(8.0);
val it = 8 : int
```

Man beachte, dass der Zweck von `round` nicht nur die Typanpassung ist, sondern zudem das Auf- bzw. Abrunden ist.

SML verfolgt einen Mittelweg zwischen schwach und stark typisierten Programmiersprachen, den Weg der sogenannten „polymorphen Typüberprüfung“ (*polymorphic type checking*): Anstatt vom Programmierer die explizite Angabe von Typen (wie etwa in der Programmiersprache Pascal) zu verlangen, ermittelt SML wenn möglich selbst, was die Typen der Bezeichner sind.

2.3 Präzedenz- und Assoziativitätsregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken

Wir haben gesehen, dass der Ausdruck `2 * 3 + 1` für `(2 * 3) + 1` steht und dass der Ausdruck `not not false` für den (inkorrekt gebildeten) Ausdruck `(not not) false` steht.

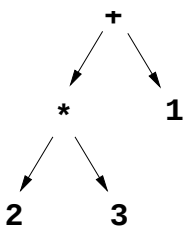
Dahinter stehen zwei Begriffe: die Präzedenzen und die Assoziativitätsregeln für Operatoren.

Präzedenzen von Operatoren legen fest, welche implizite Klammerung bei unzureichend oder gar nicht geklammerten Ausdrücken gemeint sein soll. Man sagt, dass `*` stärker bindet als `+`, was bedeutet, dass z.B. `2 * 3 + 1` für `(2 * 3) + 1` steht. Obwohl diese Annahme üblich ist, könnte eine Programmiersprache genauso auf der Annahme beruhen, dass `*` weniger stark als `+` bindet.

Die Assoziativitätsregeln legen fest, ob fehlende Klammerungen von links oder rechts her einzusetzen sind, d.h. ob `2 + 3 + 4` für `(2 + 3) + 4` oder für `2 + (3 + 4)` steht. In SML sind übrigens `+` und `*` linksassoziativ, d.h. `2 + 3 + 4` steht für `(2 + 3) + 4`. Auch wenn in vielen Fällen beide Klammerungen denselben Wert liefern, ist der Wert im allgemeinen von der Assoziativitätsregel abhängig: `10 - 7 - 1` hat den Wert `2`, wenn `-` wie in SML als linksassoziativ behandelt wird. Würde `-` als rechtsassoziativ behandelt, hätte der Ausdruck den Wert `4`. In manchen Programmiersprachen sind die Assoziativitätsregeln auch deshalb wichtig, weil sie die Reihenfolge der Auswertung bestimmen.

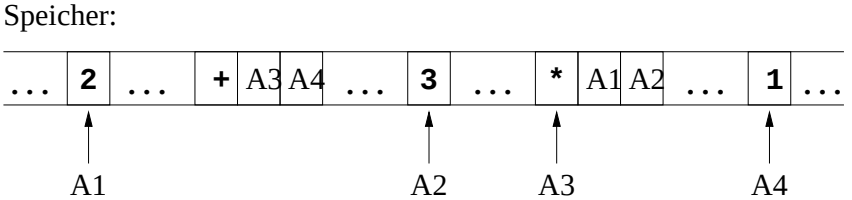
Ausdrücke, die SML zur Auswertung weitergereicht werden, sind linear, weil sie aus einer Folge von Zeichen bestehen. Beispiele von linearen Ausdrücke sind `(2 * 3) + 1` und `2 * 3 + 1`. Die Syntax solcher Ausdrücke wird von SML analysiert, bevor die Ausdrücke ausgewertet werden. Die Syntaxanalyse des linearen Ausdrucks `(2 * 3) + 1` führt zur Bildung einer baumartigen Struktur im Speicher wie:

(*)



Dabei stellen die gerichteten Kanten Zeiger, d.h. Speicheradressen, dar (siehe Informatik 3). Im (linear angeordneten) Speicher ist der obige Baum wie folgt repräsentiert (die Ai stellen Speicheradressen dar):

(**)



Die Syntaxanalyse ist aus zwei Gründen notwendig:

1. Sie ermöglicht die Auslegung (Interpretation) von unvollständig geklammerten Ausdrücken — wie etwa `4 + 5 + 6`.

2. Sie ersetzt die sogenannte „konkrete Syntax“ von Ausdrücken, d.h. die vom Programmierer verwendete Darstellung, durch die sogenannte „abstrakte Syntax“, d.h. die Repräsentation im Speicher durch „Bäume“ à la (**), die von SML zur Auswertung verwendet wird.

Man beachte, dass eine Baumdarstellung à la (*) genauso wie die lineare konkrete Syntax eine abstrakte Wiedergabe der abstrakten Syntax ist.

Da `2 * 3 + 1` in SML für `(2 * 3) + 1` steht, führt die Syntaxanalyse von `2 * 3 + 1` zur Bildung desselben Baumes wie die Syntaxanalyse von `(2 * 3) + 1`!.

Da Computer eine baumartige Repräsentation von Ausdrücke verwenden, kann man sich manchmal fragen, ob manche Überlegungen oder Überprüfungen, die Menschen durchführen, nicht ebenfalls auf Bäumen statt (linearen) Ausdrücken beruhen, also unter Verwendung der abstrakten statt konkreten Syntax stattfinden sollten.

Die abstrakte Syntax ist nur dann wünschenswert, wenn die interne Repräsentation der Ausdrücke im Speicher bei der Untersuchung eine Rolle spielt. Sonst ist die konkrete Syntax vom Vorteil, weil sie für Menschen einfacher (vor allem zu schreiben) ist. Man beachte zudem, dass eine Baumdarstellung von Ausdrücken wie (**) keine geringere Abstraktion der Speicherdarstellung als die konkrete (lineare) Syntax ist.

2.4 Namen, Bindungen und Deklarationen

Mit einer „Deklaration“ kann ein Wert an einen „Namen“ gebunden werden. Mögliche Werte, die an Namen gebunden werden können, sind unter anderem Konstanten und Funktionen. Eine Deklaration, die eine Konstante (bzw. Funktionen) an einen Namen bindet, wird Konstantendeklaration (bzw. Funktionsdeklaration) genannt.

2.4.1 Konstantendeklaration — Wertdeklarationen

```
- val zwei = 2;
val zwei = 2 : int
```

Nach dieser Konstantendeklaration kann der Namen `zwei` genauso wie die Konstante `2` verwendet werden:

```
- zwei + zwei;
val it = 4 : int

- zwei * 8;
val it = 16 : int
```

Anstatt von einer Konstantendeklaration spricht man auch von einer Wertdeklaration, daher das Wort `val` (englisch „value“). Alle Konstantendeklarationen sind Wertdeklarationen, aber nicht alle Wertdeklarationen sind Konstantendeklarationen — siehe unten.

2.4.2 Funktionsdeklaration

```
- fun zweimal(x) = 2 * x;
val zweimal = fn : int -> int
```

Anstelle des eigentlichen Wertes der Funktion, die an den Namen `zweimal` gebunden wird, gibt SML „`fn`“ (für Funktion) an. Dabei handelt es sich lediglich um eine Kurzmitteilung. Der Wert des Namens `zweimal` ist die Funktion, die als Eingabe eine ganze Zahl erhält und das Doppelte dieser Zahl als Ausgabe liefert.

Nachdem eine Funktion deklariert wurde, kann sie aufgerufen werden:

```
- zweimal(8);
val it = 16 : int

- zweimal(zweimal(8));
val it = 32 : int
```

Neben dem Kürzel „`fn`“, das als Platzhalter für den Wert des (Funktions-)Namens steht, liefert SML den Typ der deklarierten Funktion, hier: `int -> int`. Dieser Typ wurde wie folgt ermittelt: Da `2` eine ganze Zahl ist, steht die überladene Operation `*` für die Multiplikation ganzer Zahlen. Folglich muss `x` vom Typ „ganze Zahl“ sein (daher `int ->`). Da `*` die Multiplikation ganzer Zahlen ist, ist der von `zweimal` berechnete Wert eine ganze Zahl (daher `-> int`).

Die Ermittlung des Typs `int -> int` der Funktion `zweimal` ist ein Beispiel der „polymorphen Typüberprüfung“ (siehe 2.2) von SML.

Anstatt von *Funktionsdeklaration* spricht man auch von *Funktionsdefinition.*

Die folgenden Deklarationen sind gleichwertig:

```
- fun zweimal (x) = 2 * x;

- fun zweimal x = 2 * x;
```

2.4.3 Funktion als Wert — Anonyme Funktion

Für SML ist eine Funktion ein Wert. Folglich kann auch das Deklarationskonstrukt `val` verwendet werden, um einen Namen an eine Funktion zu binden. Dazu wird eine besondere Notation verwendet, die die Definition von anonymen Funktionen ermöglicht.

Die Funktion `zweimal` kann z.B. wie folgt definiert werden:

```
val zweimal = fn x => 2 * x;
```

Diese Deklaration kann wie folgt in Worte gefasst werden: An den Namen `zweimal` wird die anonyme Funktion gebunden, die eine Zahl `x` als Argument erhält und `2 * x` liefert. Der Teil `fn x => 2 * x` definiert eine anonyme Funktion. `fn` wird oft „lambda“ (λ) ausgesprochen.

Vorsicht: Verwechseln Sie die Konstrukte `fn` und `fun` von SML nicht!

2.4.4 Formale und aktuelle Parameter einer Funktion

In der Funktionsdeklaration

```
fun zweimal(x) = 2 * x;
```

wird `x` formaler Parameter (der Funktionsdeklaration oder Funktionsdefinition) genannt.
Im Funktionsaufruf `zweimal(8)` wird `8` der aktuelle Parameter (des Funktionsaufrufes) genannt.
Formale Parameter haben in Funktionsdeklarationen eine ähnliche Bedeutung wie Pronomen in natürlichen Sprachen. Die Deklaration der Funktion `zweimal` kann wie folgt paraphrasiert werden: Um zweimal von ETWAS zu berechnen, multipliziere ES mit 2.

2.4.5 Rumpf oder definierender Teil einer Funktionsdeklaration

Der Rumpf oder definierende Teil einer Funktionsdeklaration ist der Teil nach dem Zeichen „=“. Im Falle der folgenden Deklaration der Funktion `zweimal`

```
fun zweimal(x) = 2 * x;
```

ist der Rumpf `2 * x`.

2.4.6 Namen, Variablen und Bezeichner

Diese drei Begriffe haben dieselbe Bedeutung.

2.4.7 Typ-Constraints

Da `x + x = 2 * x` ist, hätte man die Funktion `zweimal` wie folgt definieren können:

```
- fun zweimal(x) = x + x;
```

Eine solche Deklaration wird nicht von allen SML-Systemen als korrekt angenommen, weil es nicht eindeutig ist, ob der formale Parameter `x` den Typ ganze Zahl oder den Typ reelle Zahl besitzt. Manche Systeme nehmen an, dass `x` den Typ ganze Zahl besitzt, weil sie im Zweifel annehmen, dass `+` für die Addition von ganzen Zahlen steht. Andere SML-Systeme machen keine solche Annahme und verwerfen die vorangehende Funktionsdeklaration als inkorrekt.
Typ-Constraints (auch Typisierungsausdrücke genannt) ermöglichen, die fehlende Information anzugeben, z.B. wie folgt:

```
- fun zweimal x: int = x + x;
```

womit der Typ des Ergebnisses (des berechneten und gelieferten Wertes) angegeben wird; oder wie folgt:

```
- fun zweimal(x: int) = x + x;
```

womit der Typ des Parameters angegeben wird; oder sogar wie folgt:

```
- fun zweimal(x: int): int = x + x;
```

womit sowohl der Typ des Ergebnis als auch der Typ des Parameters angegeben werden.
Mit einem Typ-Constraint kann die folgende Funktion für reelle Zahlen definiert werden:

```
- fun reell_zweimal x:real = x + x;
val reell_zweimal = fn : real -> real

- val pi = 3.1416;
val pi = 3.1416 : real

-  reell_zweimal pi;
val it = 6.2832 : real
```

Man beachte, dass vor und nach dem Zeichen „:“ in einem Typ-Constraint ein oder mehrere Leerzeichen zulässig sind.

2.4.8 Syntax von Namen

SML unterscheidet zwischen „alphabetischen“ und „symbolischen Namen“, je nach dem, wie sie gebildet sind. Diese Unterscheidung betrifft nicht die Verwendung der Namen: sowohl symbolische wie alphabetische Namen können in Konstanten- und Funktionsdeklarationen verwendet werden.

Alphabetische Namen fangen mit einem (kleinen oder großen) Buchstaben an, dem endlich viele (auch null) Buchstaben (`a ...z A ...Z`), Ziffern (`0 1 2 ...9`), Underscore (`_`), Hochkommata (single quote: `'`) folgen.

Symbolische Namen sind (endliche) Folgen der folgenden Zeichen:

`!` `%` `&` `$` `#` `+` `-` `*` `/` `:` `<` `=` `>` `?` `@` `\` `~` `‘` `^` und `|`.

```
- val @#!@@@ = 12;
val @#!@@@ = 12 : int

- fun $@#? x = 5 * x;
val $@#? = fn : int -> int
```

Vorsicht: Die folgenden symbolischen Namen haben in SML eine vordefinierte Bedeutung:

`:` `|` `=` `=>` `->` `#`

2.4.9 Dateien laden (einlesen)

Es ist empfehlenswert, Funktionsdeklarationen in einer Datei zu speichern und dann das Laden, d.h. Einlesen, dieser Deklarationen anzufordern. Heißt die Datei meine_datei.sml, dann geschieht dies wie folgt:

```
- use("meine_datei.sml");
val it = () : unit
```

Dabei ist `()` (gesprochen „unity“) der einzige Wert eines besonderen Datentyps namens `unit`, dessen Zweck es ist, einen Wert für Funktionsaufrufe zu liefern, die eigentlich keinen Wert berechnen, sondern wie die vordefinierte Funktion `use` einen Nebeneffekt bewirken, im Falle von `use` das Laden (oder Einlesen) einer Datei.

2.5 Fallbasierte Definition einer Funktion

2.5.1 if-then-else

SML ermöglicht fallbasierte Funktionsdefinitionen. Eine Funktion `Vorzeichen`, die der vordefinierten Funktion `Int.sign` (siehe Abschnitt 2.1) entspricht, kann zum Beispiel wie folgt definiert werden:

```
fun Vorzeichen(x : int) = if x > 0 then 1
                          else if x < 0 then ~1
                               else 0;
```

Das Konstrukt `if Test then E1 else E2` stellt die Anwendung einer wie folgt definierten Funktion auf `Test` dar:

```
(fn true => E1 | false => E2)
```

`if Test then E1 else E2` steht also für `(fn true => E1 | false => E2)(Test)`.
Im Gegensatz zu (imperativen) Programmiersprachen wie Pascal ist in SML der `else`-Teil von if-then-else-Ausdrücken nicht abdingbar. Der Grund ist, dass ein Ausdruck ohne `else`-Teil wie `if B then A` keinen Wert hätte, wenn die Bedingung `B` den Wert `false` hätte, was in der funktionalen Programmierung unmöglich ist. In einer imperativen Programmiersprache hat ein if-then-else-Ausdruck wie `if B then A` die Bedeutung eines bedingten Befehls.
In einem SML-Ausdruck `if B then A1 else A2` müssen `A1` und `A2` denselben Typ haben.

2.5.2 Pattern Matching („Musterangleich“)

In der Definition der obigen anonymen Funktion sind zwei Aspekte bemerkenswert:

1. „`|`“ drückt eine Alternative aus.
2. Die Ausdrücke `true` und `false` stellen sogenannte Muster (Patterns) dar. „Matcht“ der Wert des aktuellen Parameters mit dem ersten Muster, so wird der Wert des Ausdrucks `E1` geliefert. Ansonsten wird getestet, ob der Wert des aktuellen Parameters mit dem zweiten Muster „matcht“.

Es können mehr als zwei Fälle vorkommen. In solchen Fällen werden die Muster sequenziell in der Reihenfolge der Definition probiert, bis einer mit dem Wert des aktuellen Parameters „matcht“. Das Muster `_` (wildcard) stellt einen Fangfall dar, d.h. matcht mit jedem möglichen Wert des aktuellen Parameters. Das Wildcard-Symbol wird nicht im Rumpf eines Falles (also hinter `=>`) verwendet. Das folgende Prädikat liefert `true`, wenn es auf eine ganze Zahl angewandt wird, die eine (nicht-negierte) Ziffer ist:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
```

```
  | 5 => true
  | 6 => true
  | 7 => true
  | 8 => true
  | 9 => true
  | _ => false;
```

Vorsicht: Ein Muster ist kein Test wie etwa (`x > 0`), sondern repräsentiert mögliche Werte des Parameters.
Das Pattern Matching wird auf Deutsch auch „Angleich“ oder „Musterangleich“ genannt.

2.6 Definition von rekursiven Funktionen

2.6.1 Rekursive Berechnung der Summe der n ersten ganzen Zahlen

Es sei die Funktion `summe` zu programmieren, die zu jeder natürlichen Zahl n die Summe aller natürlichen Zahlen von 0 bis einschließlich n liefert. `summe` kann unter anderem wie folgt definiert werden:

$$\texttt{summe}(\texttt{n}) = \begin{cases} \texttt{0} & \text{falls } \texttt{n} = \texttt{0} \\ \texttt{n} + \texttt{summe}(\texttt{n} - \texttt{1}) & \text{falls } \texttt{n} > \texttt{0} \end{cases}$$

was in SML in einer der folgenden Weisen programmiert werden kann:

```
fun summe(n) = if n = 0 then 0 else n + summe(n-1);
```

oder

```
val rec summe = fn 0 => 0 | n => n + summe(n-1);
```

Man beachte das Wort `rec`, das dazu dient hervorzuheben, dass diese Wertdefinition die Definition einer rekursiven Funktion ist. Das Hinzufügen von `rec` nach `val` ist unabdingbar, weil `summe` rekursiv ist.

2.6.2 Effiziente Berechnung der Summe der n ersten ganzen Zahlen

`summe` kann auch wie folgt definiert werden:

$$\texttt{summe}(\texttt{n}) = \texttt{n} * (\texttt{n} + \texttt{1})/\texttt{2}$$

Diese Definition führt zu wesentlich effizienteren Berechnungen, weil sie für jedes n nur drei Grundoperationen verlangt. Diese Definition kann in SML wie folgt programmiert werden:

```
fun summe(n) = n * (n + 1) div 2;
```

2.6.3 Induktionsbeweis

Warum gilt diese Definition? Ihre Korrektheit kann wie folgt induktiv bewiesen werden.

Beweis:

Induktionsbasis:

Für $n = 0$ gilt die Gleichung $summe(n) = n*(n+1)/2$ offenbar, weil $0*(0+1)/2 = 0$.

Induktionsschritt:

Induktionsannahme:
Sei angenommen, für eine natürliche Zahl k gilt $summe(k) = k * (k + 1)/2$.
Zeigen wir, dass sie auch für die Nachfolgerzahl $k + 1$ gilt.
$summe(k + 1) = k + 1 + summe(k)$.
Nach Induktionsannahme: $summe(k) = k * (k + 1)/2$.
Also
$summe(k + 1) = k + 1 + summe(k) =$
$k + 1 + (k * (k + 1)/2) =$
$[2(k + 1) + (k * (k + 1)]/2 =$
$[(k + 2) * (k + 1)]/2$.

qed.

Die Technik, die in diesem Beweis angewendet wurde, heißt „vollständige Induktion“. Induktionsbeweise gehören zu den unabdingbaren Techniken der Informatik.

2.6.4 Alternativer Beweis

Beweis:

Sei $n \in \mathbb{N}$.

Fall 1: n ist gerade.
Die ganzen Zahlen von 1 bis n können in Paaren $(n, 1)$, $(n-1, 2)$, $(n-2, 3)$, ... gruppiert werden. Das letzte solcher Paare ist $((n/2)+1, n/2)$ (*), weil kein weiteres solches Paar (a, b) die beiden Eigenschaften: $a + b = n + 1$ und $a \geq b$ besitzt. Die Summe der Zahlen jedes Paares ist $n + 1$ und es gibt $n/2$ solche Paare, also $summe(n) = n * (n + 1)/2$.

Fall 2: n ist ungerade.
Die ganzen Zahlen von 0 bis n können in Paaren $(n, 0)$, $(n-1, 1)$, $(n-2, 2)$, ... gruppiert werden. Das letzte solcher Paare ist $(\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor)$ (**). Die Summe der Zahlen jedes Paares ist n und es gibt $(n + 1)/2$ solche Paare, also $summe(n) = n * (n + 1)/2$.

qed.

Bemerkung: Die Aussagen (*) und (**) im vorangehenden Beweis verlangen im Grunde (einfache) Induktionsbeweise, die hier der Übersichtlichkeit halber ausgelassen wurden.

2.6.5 Terminierungsbeweis

Es kommt häufig vor, dass bei der Programmierung von rekursiven Funktionen ein Denk- oder Programmierungsfehler zu einer nichtterminierenden Funktion führt. Dies ist z.B. der Fall bei der folgenden Funktion:

```
fun s(n) = n + s(n+1);
```

Die Terminierung einer rekursiven Funktion wie `summe` kann unter Anwendung der Beweistechnik der vollständigen Induktion gezeigt werden. Zeigen wir, dass für alle (positiven) ganzen Zahlen `n` der Aufruf `summe(n)` terminiert.

Beweis:

Basisfall: `summe(0)` terminiert, weil nach Funktionsdeklaration der Aufruf `summe(0)` den Wert `0` liefert.

Induktionsschritt:
Induktionsannahme: Sei angenommen, dass für eine natürliche Zahl `k` der Aufruf `summe(k)` terminiert.

Zeigen wir, dass der Aufruf `summe(k + 1)` terminiert. Nach Funktionsdeklaration liefert der Aufruf `summe(k + 1)` den Wert von `k + 1 + summe(k)`. Nach Induktionsannahme terminiert der Aufruf `summe(k)`. Folglich terminiert auch der Aufruf `summe(k + 1)`.

qed.

Dieser Beweis stellt exemplarisch dar, wie Terminierungsbeweise sowie Beweise anderer Eigenschaften von rekursiven Funktionen unter Anwendung der Beweistechnik „vollständige Induktion“ geführt werden können.

Nur in den seltensten Fällen kann man sich durch Tests von der Korrektheit eines Programms überzeugen, weil es in der Regel wie bei der rekursiven Funktion `summe` unendlich — oder zu viele — mögliche Aufrufparameter gibt. Durch Testen kann man zwar Fehler finden, allerdings ohne Garantie, dass alle Fehler gefunden werden. So sind Beweise unabdingbare Bestandteile der Programmentwicklung.

2.7 Wiederdeklaration eines Namens — Statische Bindung — Umgebung

2.7.1 Wiederdeklaration eines Namens

Betrachten wir die folgende Sitzung:

```
- val zwei = 2;
val zwei = 2 : int

- fun zweimal(n) = zwei * n;
val zweimal = fn : int -> int

- zweimal(9);
val it = 18 : int

- val zwei = 0;
val zwei = 0 : int

- zweimal(9);
val it = 18 : int

- fun zweimal'(n) = zwei * n;
val zweimal' = fn : int -> int

- zweimal'(9);
val it = 0 : int
```

Es ist zulässig, die Bindung eines Wertes, sei es eine Konstante wie im obigen Beispiel oder eine Funktion, an einen Namen durch eine neue Deklaration zu ändern (Wiederdeklaration, engl. redeclaration/redefinition). Wird der Name in einer Deklaration verwendet, dann gilt seine letzte Bindung an einen Wert.

2.7.2 Statische und dynamische Bindung

Die Wiederdeklaration eines Namens gilt jedoch nicht für Deklarationen, die diesen Namen vor der Wiederdeklaration verwendet haben. So steht `zwei` für `2` in der Deklaration der Funktion `zweimal`, für `0` in der Deklaration der Funktion `zweimal'`. Man sagt, dass die

Bindung in SML eine „statische Bindung“ (oder „lexikalische Bindung“) ist. Würde die Wiederdeklaration eines Namens `N` Einfluss auf Funktionen haben, deren Rümpfe sich auf `N` beziehen, so würde man von einer „dynamischen Bindung“ sprechen.

Die Wiederdeklaration von Namen und ihre Behandlung durch SML ist eine große Hilfe bei der Entwicklung von Programmen, die viele Namen verwenden.

2.7.3 Umgebung

Das SML-System verwaltet mit jeder Sitzung und jeder eingelesenen Datei, d.h. Programm, eine geordnete Liste von Gleichungen der Gestalt Name = Wert (dargestellt als Paare (Name, Wert)), die Umgebung heißt. Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N = W` am Anfang der Umgebung. Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. So gilt immer als Wert eines Namens `N` derjenige Wert, der bei der letzten Deklaration von `N` angegeben wurde.

Kommt ein Name `A` im Wertteil `W` einer Deklaration `val N = W` oder `val rec N = W` oder `fun N = W` vor, so wird der Wert von `A` ermittelt und in `W` anstelle von `A` eingefügt, bevor der Eintrag für `N` in der Umgebung gespeichert wird. So verändert eine spätere Wiederdeklaration von `A` den Wert von `N` nicht.

2.8 Totale und partielle Funktionen (Fortsetzung)

Der Unterschied zwischen totalen und nichttotalen (also partiellen) Funktionen ist für die Programmierung vom Belang. Die rekursive Funktion `summe` mit Typ `int` → `int` aus Abschnitt 2.6

```
fun summe(n) = if n = 0 then 0 else n + summe(n-1);
```

ist über den ganzen Zahlen nicht total, weil ein Aufruf von `summe` mit einem nichtpositiven Eingabeparameter wie etwa `summe(~25)` nicht terminiert. Über den natürlichen Zahlen ist diese Funktion aber total.

Es ist wichtig zu ermitteln, über welchem Bereich eine programmierte Funktion total ist. Dies benötigt man, um sicherzustellen, das die Funktion auch nur mit entsprechenden Parametern aufgerufen wird. Sehr oft werden zudem weitere Eigenschaften der Funktion nur bezüglich dieses Bereiches angegeben.

2.9 Kommentare

In SML sind Kommentare beliebige Texte, die mit den Zeichen `(*` anfangen und mit den Zeichen `*)` enden. SML lässt geschachtelte Kommentare zu. Im folgenden Beispiel ist also die ganze Funktionsdeklaration „auskommentiert”:

```
(*
fun Vorzeichen(x : int) = if x > 0 then 1
                          else if x < 0 then ~1
                               else (* x = 0 *) 0;
*)
```

Klare und präzise Kommentare sind in jedem Programm unabdingbar. Es ist immer naiv anzunehmen, dass ein Programm, in welcher Programmiersprache auch immer, selbsterklärend sei. Programme werden zunächst für Menschen und danach für Computer geschrieben. Es ergibt keinen Sinn, Programme von Computern ausführen zu lassen, die nicht von Menschen verstanden werden.

2.10 Die Standardbibliothek von SML

Manche der vordefinierten Funktionen von SML wie `Real.compare` sind in sogenannten Modulen programmiert, d.h. in Programmen, andere wie `+` sind Teile des SML-Systems. Die SML-Bezeichnung für Module ist „Struktur“ (structure).

Eine Funktion `F`, die in einem Modul `M` definiert ist, wird außerhalb dieses Moduls als `M.F` bezeichnet — und aufgerufen.

Die Standardbibliothek stellt eine Sammlung von Modulen (Strukturen) für herkömmliche Typen wie reelle Zahlen dar. Die Module der Standardbibliothek werden vom SML-System automatisch geladen. Das Laden von anderen Modulen muss aber vom Programmierer explizit angefordert werden — siehe Kapitel 12.

Siehe „The Standard ML Basis Library“ unter
http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/

2.11 Beispiel: Potenzrechnung

2.11.1 Einfache Potenzrechnung

Es sei die folgende Funktion in SML zu programmieren:

$$\begin{array}{lccc} \texttt{potenz:} & \mathbb{Z}\times\mathbb{N} & \to & \mathbb{Z} \\ & (a,b) & \mapsto & a^b \end{array}$$

Die Potenz ist übrigens keine vordefinierte Funktion in SML.

Die folgenden Gleichungen liefern die Grundlage für ein rekursives Programm:

$$\begin{array}{ll} a^b = 1 & \text{falls } b = 0 \\ a^b = a * a^{b-1} & \text{andernfalls} \end{array}$$

Daraus folgt die folgende Implementierung in SML:

```
fun potenz(a, b) = if b = 0 then 1 else a * potenz(a, b - 1);
```

2.11.2 Terminierungsbeweis für die einfache Potenzrechnung

Wir beweisen induktiv, dass für alle $(a, b) \in (\mathbb{Z}\times\mathbb{N})$ der Aufruf `potenz(`a`, `b`)` terminiert.

Beweis:

Sei a eine beliebige ganze Zahl.

Basisfall: $b = 0$. Nach Funktionsdeklaration terminiert der Aufruf und liefert 1.

Induktionsschritt: Sei angenommen, dass für ein gegebenes $b \in \mathbb{N}$ der Aufruf `potenz(`a`, `b`)` terminiert (Induktionshypothese). Nach Definition berechnet der Aufruf `potenz(`a`, `$b+1$`)` den Wert von a `* potenz(`a`, `b`)`. Er terminiert also nach Induktionsannahme.

qed.

2.11.3 Zeitbedarf der einfachen Potenzberechnung

Der Zeitbedarf wird als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt. Diese Schätzung stellt eine Vergröberung dar, weil Multiplikationen kleiner Zahlen weniger Zeit verlangen als die Multiplikation großer Zahlen. Solche vergröbernden Annahmen sind bei Schätzungen des Zeitbedarfs üblich.

Die Berechnung von `potenz(a, b + 1)` bedarf einer Multiplikation mehr als die Berechnung von `potenz(a, b)`, die Berechnung von `potenz(a, 0)` bedarf keiner Multiplikation. Also bedarf die Berechnung von `potenz(a, b)` insgesamt `b` Multiplikationen.

Man sagt, dass der Zeitbedarf der Funktion `potenz` linear im zweiten Argument ist.

2.11.4 Effizientere Potenzrechnung

Ist b gerade mit $b = 2k$, so gilt: $a^b = a^{2k} = (a^k)^2$. Es ist also möglich für gerade natürliche Zahlen b die b-Potenz einer ganzen Zahl a mit weniger als b Multiplikationen zu berechnen. Diese Beobachtung führt zur folgenden Funktionsdeklaration:

```
fun potenz'(a, b) = if b = 0
                    then 1
                    else if gerade(b)
                         then quadrat(potenz'(a, b div 2))
                         else a * potenz'(a, b - 1);
```

wobei die Hilfsfunktionen `gerade` und `quadrat` wie folgt deklariert werden:

```
fun gerade(a) = (a mod 2 = 0);

fun quadrat(a : int) = a * a;
```

2.11.5 Zeitbedarf der effizienteren Potenzberechnung

Der Zeitbedarf der Funktion `potenz'` wird wie bei der Funktion `potenz` als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt. Nach dieser Annahme werden die Rechenzeiten für die Aufrufe des Prädikats `gerade` vernachlässigt.

So geschätzt ist die Rechenzeit abhängig von `b` und unabhängig von `a`. Sei also $rz(\texttt{b})$ die Rechenzeit eines Aufrufes `potenz'(a, b)` (für eine beliebige ganze Zahl `a` und für eine natürliche Zahl `b`). Es gilt:

$$
\begin{array}{lll}
(*) & rz(2^{\texttt{b}}) & = \; rz(2^{\texttt{b}-1}) + 1 \\
(**) & rz(0) & = \; 0
\end{array}
$$

Es gilt also:

$$(***) \quad rz(2^{\mathtt{b}}) = \mathtt{b}$$

Auf die Potenzen von 2 ist also rz die Umkehrung der Funktion $\mathtt{b} \mapsto 2^{\mathtt{b}}$, d.h. der Logarithmus zur Basis 2, genannt $\log_2$. Diese Beobachtung liefert keinen präzisen Wert für Zahlen, die keine Potenzen von 2 sind.

Für große Zahlen ist der Zeitbedarf von `potenz'` viel geringer als der Zeitbedarf von `potenz`. Zum Beispiel bedarf `potenz'(a,1000)` nur 14 Multiplikationen anstatt der 1000 Multiplikationen von `potenz(a, 1000)`. Für wachsende Werte von `b` vergrößert sich sehr schnell der Berechnungszeitabstand zwischen `potenz'` und `potenz`:

`b`	`potenz(a,b)`	`potenz'(a,b)`
1	1 Multiplikation	1 Multiplikation
10	10 Multiplikationen	5 Multiplikationen
100	100 Multiplikationen	9 Multiplikationen
1000	1000 Multiplikationen	14 Multiplikationen
⋮	⋮	⋮

2.11.6 Bessere Implementierung der effizienteren Potenzrechnung

Die folgende Implementierung der effizienteren Potenzrechnung ist auch möglich:

```
fun potenz''(a, b) = if b = 0
                     then 1
                     else if gerade(b)
                          then potenz''(quadrat(a), b div 2)
                          else a * potenz''(a, b - 1);
```

In Abschnitt 4.3.4 werden wir sehen, dass der `then`–Fall der Funktion `potenz''` „endrekursiv“ ist, d.h. dass der rekursiver Aufruf außer im `if-then-else`–Ausdruck in keinem weiteren zusammengesetzten Ausdruck vorkommt. Man beachte, dass der `else`–Fall der Funktion `potenz''` nicht endrekursiv ist. In diesem Abschnitt wird dann erläutert, warum Funktionen mit nur endrekursiven Aufrufen (wie im `then`–Fall der Funktion `potenz''`) gegenüber Funktionen mit nicht-endrekursiven Aufrufen (wie im `else`–Fall der Funktion `potenz''`) vorzuziehen sind.

Es ist leicht zu überprüfen, dass die Zeitbedarfsanalyse aus Abschnitt 2.11.5 ebenfalls auf die Funktion `potenz''` zutrifft.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 3

Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)

Dieses Kapitel ist der Auswertung von Ausdrücken gewidmet. Es führt ein einfaches abstraktes Modell ein, das sogenannte Substitutionsmodell, womit verschiedene Formen der Auswertung von Ausdrücken in funktionalen Programmiersprachen definiert werden können. Dann wird unterschieden zwischen zwei Arten von Variablen, funktionalen Variablen und Zustandsvariablen, die in Programmiersprachen vorkommen. Abschließend wird gezeigt, dass das Substitutionsmodell nicht ermöglicht, die Auswertung von Ausdrücken zu definieren, in denen Zustandsvariablen vorkommen.

3.1 Auswertung von Ausdrücken

3.1.1 Arten von Ausdrücken

Nicht alle Ausdrücke haben denselben Zweck: Konstanten- und Funktionsdeklaration wie etwa

```
val zwei = 2;
```

und

```
fun quadrat(x: int) = x * x;
```

binden Werte, das sind Konstanten oder Funktionen, an Namen. Funktionsanwendungen wie etwa `quadrat(3 + zwei)` und `quadrat(3) + quadrat(2)` wenden Funktionen auf Werte an. Von dem Zweck eines Ausdrucks hängt ab, wie der Ausdruck ausgewertet wird. Zunächst unterscheiden wir verschiedene Arten von Ausdrücken.

- Funktionale Ausdrücke sind Konstanten und Funktionsanwendungen wie etwa `3`, `2`, `true`, `false`, `zwei`, `quadrat(3 + 2)`, `3 * 2`, `not false`, `not false <> true`, usw. Ein funktionaler Ausdruck kann atomar sein wie etwa `3`, `2`, `true`, `false`, `zwei` oder zusammengesetzt wie etwa `quadrat(3 + 2)`, `3 * 2`, `not false`, `not false <> true`.

- Sonderausdrücke werden die Ausdrücke genannt, die keine funktionalen Ausdrücke sind. Darunter versteht man in SML Ausdrücke, die mit den folgenden vordefinierten Konstrukten gebildet sind:
  - `val` und `fun`, die zur Wertdeklaration dienen
  - `if-then-else`, `case` und das *Pattern Matching* zur Fallunterscheidung
  - die Boole'schen Operatoren `andalso` und `orelse`

Es mag zu Recht überraschen, das Ausdrücke, die mit `andalso` bzw. `orelse` gebildet sind, in SML keine funktionalen Ausdrücke sind. Der Grund dafür wird später erläutert.

Im Lauf des Semesters werden wir weitere SML-Konstrukte zur Bildung von Sonderausdrücken kennenlernen.

3.1.2 Die Auswertung von Ausdrücken als Algorithmus

Zunächst stellt sich die Frage, wie — d.h. mit welchen Mitteln oder in welchem Formalismus — die Auswertung von Ausdrücken spezifiziert werden soll. Jeder Formalismus, der ausreichend präzise und verständlich ist, mag als geeignet erscheinen. Es ist aber wichtig einzusehen, dass die Auswertung von Ausdrücken als Algorithmus formalisiert werden muss:

- zum einen, weil es sich um eine (symbolische) Berechnung handelt, die prinzipiell derselben Art ist, wie die Multiplikation zweier natürlicher Zahlen.
- zum anderen, weil die Auswertung von Ausdrücken auf einem Computer durchgeführt werden soll.

Es ist in der Wissenschaft zweckmäßig, nicht mehr Begriffe, Methoden und Annahmen als zwingend erforderlich einzuführen. Da uns der Algorithmusbegriff zu Verfügung steht und zur Spezifikation der Auswertung von Ausdrücken zweckmäßig ist, ist es also angebracht, ihn dazu zu verwenden.

Die Einschränkung auf so wenige Begriffe, Methoden und Annahmen wie möglich wurde schon im 15. Jhdt. von dem englischen Franziskaner-Mönch und Philosoph Wilhelm von Occam — der übrigens eine Zeit lang in München weilte — als philosophisches und wissenschaftliches Prinzip formuliert. Seitdem wird dieses Prinzip „Occam'sches Messer“ genannt. Sinnbild dieser Bezeichnung ist das Herausschneiden nutzloser Begriffe, Methoden oder Annahmen. (Der Roman „Der Name der Rose“ von Umberto Eco, dessen Hauptfigur von Occam inspiriert wurde, führt leicht verständlich in die Occam'sche philosophische Lehre ein.)

Es ist wichtig zu verstehen, dass die Durchführung der Auswertung auf einem Computer nicht der Hauptgrund ist, weswegen zur Spezifikation der Auswertung von Ausdrücken der Algorithmusbegriff verwendet wird. Es sei daran erinnert, dass Algorithmen und Programme zunächst für Menschen verfasst werden und nur nebenbei für Computer. Dies trifft besonders auf die Auswertung von Ausdrücken zu: Was wäre der Nutzen einer Auswertung, die Menschen nicht gut verstehen könnten? Die Entwicklung von Programmen, die einer solchen Auswertung unterliegen, wäre schwierig und sehr fehleranfällig.

3.1.3 Die Auswertung von Ausdrücken als rekursive Funktion

Soll die Auswertung von Ausdrücken als Algorithmus spezifiziert werden, dann stellt sich die Frage, welche Art von Algorithmus sich dazu eignet. Als rekursive Funktion lässt sich die Auswertung von Ausdrücken wie folgt besonders elegant, d.h. einfach und prägnant, informell spezifizieren.

Skizze des Auswertungsalgorithmus

Zur Auswertung eines Ausdrucks A gehe wie folgt vor:

1. Werte die Teilausdrücke von A aus.
2. Wende die Funktion, die sich als Wert des am weitesten links stehenden Teilausdrucks ergibt, auf die Werte an, die sich als Werte aus der Auswertung der restlichen Teilausdrücke ergeben.

Diese Skizze eines Algorithmus ist rekursiv, weil er die Auswertung von Teilausdrücken anfordert. Diese unvollständige und unpräzise Algorithmusskizze muss jetzt noch verfeinert werden.

Annahmen des Auswertungsalgorithmus

Wir nehmen (nur für den Auswertungsalgorithmus) der Einfachheit halber an, dass für alle Funktionen die Präfixschreibweise statt der Infixschreibweise benutzt wird, also zum Beispiel `+(1,2)` statt `1 + 2` usw., auch wenn das keine korrekte SML-Syntax ist.
Gewisse Funktionen wie die Addition oder die Multiplikation stehen zur Verfügung, ohne dass man sie definieren muss. Solche Funktionen nennen wir Systemfunktionen. Andere Funktionen können definiert werden, wie zum Beispiel:

```
fun quadrat(x) = *(x, x)
```

Diese Definition ist nur „syntaktischer Zucker“ für die Definition

```
val quadrat = fn(x) => *(x, x)
```

In der aktuellen Umgebung hat der Name `quadrat` als Wert also die Funktion `fn(x) => *(x, x)`.

Auswertungsalgorithmus

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Falls `A` atomar ist, dann:
   (a) Falls `A` vordefiniert ist, liefere den vordefinierten Wert von `A` (dieser kann auch eine Systemfunktion sein.)
   (b) Andernfalls (der Wert von `A` ist in der Umgebung definiert) sei $\texttt{A} = \texttt{W}$ die Gleichung in der Umgebung, die den Wert von `A` definiert. Liefere `W` als Wert von `A`.
   (`W` kann auch eine Funktion der Form $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_k) \Rightarrow \texttt{R}$ sein.)
2. Andernfalls (`A` ist zusammengesetzt) hat `A` die Form $\texttt{B}(\texttt{A}_1, \ldots, \texttt{A}_n)$ mit $\texttt{n} \geq \texttt{0}$. Werte die Teilausdrücke $\texttt{B}, \texttt{A}_1, \ldots, \texttt{A}_n$ aus. Seien $\texttt{W}_1, \ldots, \texttt{W}_n$ die Werte der Teilausdrücke $\texttt{A}_1, \ldots, \texttt{A}_n$.
   (a) Falls der Wert von `B` eine Systemfunktion ist, dann: Wende sie auf $(\texttt{W}_1, \ldots, \texttt{W}_n)$ an. Liefere den dadurch erhaltenen Wert als Wert von `A`.
   (b) Falls der Wert von `B` eine Funktion der Form $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_n) \Rightarrow \texttt{R}$ ist, dann: Werte `R` in der erweiterten Umgebung aus, die aus der aktuellen Umgebung und den zusätzlichen Gleichungen $\texttt{F}_1 = \texttt{W}_1, \ldots, \texttt{F}_n = \texttt{W}_n$ besteht. Liefere den dadurch erhaltenen Wert als Wert von `A` (die Umgebung ist nun wieder die ursprüngliche).

Muss im Fall 2 der Teilausdruck `B` tatsächlich erst scheinbar kompliziert ausgewertet werden? Betrachten wir die verschiedenen Unterfälle.

Die Existenz des Typs `unit` in SML ermöglicht es, dass $\texttt{n} = \texttt{0}$ im Fall 2 vorkommen kann. Die folgende Funktionsdeklaration ist z.B. möglich:

```
- fun f() = 1;
val f = fn : unit -> int
```

`f()` ist die Anwendung der Funktion namens `f` auf `()` (unity). Für andere funktionale Programmiersprachen kann im Fall 2 des Auswertungsalgorithmus die Einschränkung $\texttt{n} \geq \texttt{1}$ statt $\texttt{n} \geq \texttt{0}$ notwendig sein.

Beginnt ein zusammengesetzter Ausdruck `A` mit `abs`, ist also `B` der Ausdruck `abs`, dann liefert die Auswertung von `abs` die einstellige Systemfunktion, die eine ganze Zahl als Argument nimmt und deren Absolutbetrag als Wert liefert (Fall 2.a). Dabei ist `abs` nur der Name dieser Systemfunktion, aber nicht die Funktion selbst. Also muss `B` zunächst ausgewertet werden. Systemfunktionen werden übrigens häufig durch die Maschinensprache des zu Grunde liegenden Computers zur Verfügung gestellt (siehe die Grundstudiumsvorlesung „Informatik III“ und die Hauptstudiumsvorlesung „Übersetzerbau“).

Sei `quadrat` wie oben definiert, so dass der Name `quadrat` in der aktuellen Umgebung die Funktion `fn(x) => *(x, x)` als Wert hat. Ist nun `A` der zusammengesetzte Ausdruck `quadrat(2)`, ist also `B` der Ausdruck `quadrat`, so muss `B` zunächst ausgewertet werden, um diese Funktion `fn(x) => *(x, x)` zu erhalten (Fall 2.b), die dann auf die natürliche Zahl `2` angewandt wird.

Schließlich darf man nicht vergessen, dass `B` selbst ein zusammengesetzter Ausdruck sein kann. Ist zum Beispiel `A` der Ausdruck `(if n>0 then quadrat else abs)(5)`, also `B` der Ausdruck `(if n>0 then quadrat else abs)`, so ist offensichtlich, dass `B` zunächst ausgewertet werden muss, um die Funktion zu erhalten, die dann auf den Wert `5` angewandt wird.

Der obige Auswertungsalgorithmus definiert eine Funktion, die als Eingabeparameter einen Ausdruck und eine Umgebung erhält und als Wert den Wert des Ausdrucks liefert. Die Auswertungsfunktion ist rekursiv, weil in den Schritten 2 und 2.b die Auswertungsfunktion aufgerufen wird.

3.1.4 Unvollständigkeit des obigen Algorithmus

Die obige Spezifikation des Auswertungsalgorithmus ist nicht vollständig, weil sie Programmfehler und Typen nicht berücksichtigt, einige Konstrukte von SML gar nicht behandeln kann und die Behandlung der Umgebung nur unpräzis erläutert.

Offenbar können die folgenden Fehler auftreten:

- In 1(b) tritt ein Fehler auf, wenn `A` ein Name ist, der nicht deklariert wurde, d.h., für den es keine Gleichung in der Umgebung gibt.
- In 2(a) und auch in 2(b) tritt ein Fehler auf, wenn die Anzahl `n` der aktuellen Parameter $\mathtt{A}_1, \ldots, \mathtt{A}_n$ mit der Stelligkeit des Wertes von `B` nicht übereinstimmt. Das ist z.B. der Fall, wenn in der oben beschriebenen Umgebung der Ausdruck

  ```
  quadrat(3, 5)
  ```

  ausgewertet werden soll.

Die Ergänzung des obigen Auswertungsalgorithmus, damit solche Fehler erkannt und gemeldet werden, ist nicht schwierig. Sie wird in diesem Kapitel nicht weiter behandelt.

In der obigen Spezifikation des Auswertungsalgorithmus werden ferner Typen nicht berücksichtigt. Dieser Aspekt der Auswertung soll im Kapitel 6 behandelt werden.

Zu den Konstrukten, die der Auswertungsalgorithmus in der obigen Form nicht behandeln kann, gehören insbesondere alle Sonderausdrücke, aber auch Funktionen, deren formale Parameter komplexe Pattern sind, sowie einige andere Konstrukte.

Wir haben im Kapitel 2 (Abschnitt 2.7) gesehen, dass die Umgebung als geordnete Liste verwaltet wird. Diese Verwaltung ist im vorangehenden Auswertungsalgorithmus nicht näher spezifiziert.

3.1.5 Zweckmäßigkeit des obigen Algorithmus

Ist es zulässig, den Algorithmus als rekursive Funktion zu spezifizieren, wo er doch selbst unter anderem zur Auswertung von rekursiven Funktionen dienen soll? Dreht sich eine solche Spezifikation nicht im Kreis, so dass sie keinen Sinn ergibt?

Sicherlich muss unsere Annäherung an den Begriff „Algorithmus“ zu einfacheren „Urbegriffen“ führen, unter deren Verwendung komplexere Begriffe wie der Begriff einer rekursiven Funktion definiert werden können. Es ist aber nicht unzweckmäßig, sich auf dem Weg zu solchen Definitionen des informellen Verständnisses von komplexen Begriffen zu

bedienen. Anders ausgedrückt bedienen wir uns unserer Intuition von der Auswertung von (rekursiven) Funktionen, um zu verstehen, wie (rekursive) Funktionen ausgewertet werden. Nicht anders gehen z.B. Linguisten vor, wenn sie in einer Sprache die Grammatik derselben Sprache erläutern.

Die Durchführbarkeit des obigen Algorithmus (siehe unten) ist ein Zeichen dafür, dass wir uns mit seiner Spezifikation nicht im Kreis gedreht haben.

3.1.6 Beispiel einer Durchführung des Auswertungsalgorithmus

Seien folgende Deklarationen gegeben:

```
val zwei = 2;
fun quadrat(x) = *(x, x);
```

so dass die Umgebung also aus den beiden Gleichungen `quadrat = fn(x) => *(x, x)` und `zwei = 2` besteht.

Sei `A` der Ausdruck `quadrat(zwei)`, der in dieser Umgebung ausgewertet werden soll. (Die im Folgenden angegebenen Nummern beziehen sich auf die entsprechenden Fälle des Algorithmus.)

2. `A` ist zusammengesetzt: `B` ist `quadrat`, A_1 ist `zwei`, $n = 1$.
   Werte den Teilausdruck `B` aus;
   Nebenrechnung, in der `A` der Ausdruck `quadrat` ist:

   1. `A` ist atomar.

      (b) `A` ist nicht vordefiniert.
      Als Wert von `quadrat` wird aus der Umgebung `fn(x) => *(x,x)` geliefert.

   Ende der Nebenrechnung; Wert von `B` ist `fn(x) => *(x,x)`.
   Werte den Teilausdruck A_1 aus;
   Nebenrechnung, in der `A` der Ausdruck `zwei` ist:

   1. `A` ist atomar.

      (b) `A` ist nicht vordefiniert.
      Als Wert von `zwei` wird aus der Umgebung die natürliche Zahl `2` geliefert.

   Ende der Nebenrechnung; Wert von A_1 ist `2`.

   (b) Der Wert von `B` ist keine Systemfunktion, sondern eine Funktion `fn(x) => *(x,x)`.
   Die erweiterte Umgebung besteht aus der aktuellen Umgebung und der zusätzlichen Gleichung `x = 2`.
   Werte `*(x,x)` in dieser erweiterten Umgebung aus;
   Nebenrechnung, in der `A` der Ausdruck `*(x,x)` ist.

   2. `A` ist zusammengesetzt,
      `B` ist `*`, A_1 ist `x`, A_2 ist `x`, $n = 2$.
      Werte den Teilausdruck `B` aus;
      Nebenrechnung, in der `A` der Ausdruck `*` ist

      1. `A` ist atomar.

(a) `A` ist vordefiniert.
Wert von `*` ist die zweistellige Multiplikationsfunktion, also eine Systemfunktion.

Ende der Nebenrechnung;
Wert von `B` ist die Multiplikationsfunktion.
Werte den Teilausdruck $\texttt{A}_1$ aus;
Nebenrechnung, in der `A` der Ausdruck `x` ist:

1. `A` ist atomar.

(b) `A` ist nicht vordefiniert.
Als Wert von `x` wird aus der (erweiterten) Umgebung die natürliche Zahl `2` geliefert.

Ende der Nebenrechnung; Wert von $\texttt{A}_1$ ist `2`
Genauso: Wert von $\texttt{A}_2$ ist `2`

(a) Der Wert von `B` ist eine Systemfunktion, nämlich die Multiplikationsfunktion.
Wende sie auf `(2, 2)` an
Der dadurch erhaltene Wert ist `4`.

Ende der Nebenrechnung, Wert von `*(x,x)` ist `4`.
Der dadurch erhaltene Wert von `A` ist also `4`
(die Umgebung ist nun wieder die ursprüngliche, ohne die Gleichung `x=2`).

Die Notation `fn(x) => *(x,x)` wird für eine vom Benutzer selbst definierte, einstellige Funktion verwendet, die der eigentliche Wert von `quadrat` ist. Tatsächlich wird in Implementierungen aber nicht diese textuelle Notation benutzt, sondern eine Speicheradresse, an der ein „Funktionsdeskriptor“ zu finden ist. Dieser enthält die formalen Parameter, die Typen und den Rumpf der Funktion. Als „Wert“ von `quadrat` wird dann intern die Speicheradresse dieses Funktionsdeskriptors geliefert. Das SML-System benutzt einen Teil der dort vorhandenen Angaben, um die Ausgabe `fn: int -> int`, d.h. den Typ der Funktion `quadrat`, zu ermitteln.

Das Symbol `fn` (oder die Speicheradresse für einen Funktionsdeskriptor) stellt also nur einen Vermerk dar. Eine solche Vermerktechnik ist unabdingbar, weil ein Programm nur Bezeichner (auch Symbole genannt) bearbeitet. Ein Begriff wie der Funktionsbegriff wird erst durch einen Algorithmus wie den obigen Auswertungsalgorithmus verwirklicht. Auf dem Weg zu dieser Verwirklichung können nur symbolische Berechnungen stattfinden, d.h. eine Bearbeitung von Bezeichnern wie `fn`.

Dies stellt einen wesentlichen Unterschied zu Berechnungen in der Mathematik dar. Ein Mathematiker führt Teilberechnungen wie etwa Teilbeweise durch, die nicht notwendigerweise durch einen ausformulierten Algorithmus spezifiziert sind, der von dem Mathematiker penibel Schritt für Schritt durchgeführt wird — wie im obigen Beispiel der Auswertungsalgorithmus durchgeführt wurde —, sondern die von dem Mathematiker einigermaßen intuitiv durchgeführt werden.

Obwohl Mathematiker so rechnen und beweisen, heißt das nicht, dass präzis ausformulierte Berechnungs- und Beweisalgorithmen in der Mathematik verzichtbar seien. Präzise Algorithmen werden tatsächlich in der Mathematik ausformuliert und untersucht. Was Beweise angeht, stellt die präzise Formulierung der Algorithmen eine der zentralen Aufgaben des Teilgebiets der mathematischen Logik dar.

3.1.7 Substitutionsmodell

Der Auswertungsalgorithmus kann auf einer höheren Abstraktionsebene wie folgt erläutert werden:

> Um einen Ausdruck A auszuwerten, werden alle Teilausdrücke von A durch ihre Definitionen ersetzt, wobei die formalen Parameter von Funktionsdefinitionen durch die aktuellen Parameter der Funktionsanwendung ersetzt werden, vordefinierte Konstanten gemäß ihrer Definition durch ihre Werte ersetzt werden und vordefinierte Funktionen gemäß ihrer Definition ersetzt werden.

Diese sehr abstrakte Beschreibung der Auswertung wird *Substitutionsmodell* genannt.

Das Substitutionsmodell ist weder falsch noch nutzlos. Es ist aber viel weniger präzis als der vorangehende Algorithmus.

3.2 Auswertung in applikativer und in normaler Reihenfolge

3.2.1 Auswertungsreihenfolge

Vergleicht man den (präzisen) Auswertungsalgorithmus mit dem (abstrakteren) Substitutionsmodell, so stellt man fest, dass der Algorithmus die Reihenfolge der Auswertung der Teilausdrücke eines zusammengesetzten Ausdruck festlegt, das Substitutionsmodell aber diese Reihenfolge nicht näher definiert.

Betrachten wir z.B. den Ausdruck `quadrat(2 + 1)`.

Der Auswertungsalgorithmus „merkt“ sich mit dem Vermerk „fn“, dessen er sich als Pseudo-Wert eines Funktionsnamens bedient, dass quadrat eine in der Umgebung definierte Funktion ist. Dann wertet er den Teilausdruck `2 + 1` aus. Erst dann wird die Funktion namens `quadrat` auf den so ermittelten Wert `3` von `2 + 1` angewandt, was zur Auswertung des Ausdrucks `3 * 3` führt.

Der Auswertungsalgorithmus wertet also zunächst die aktuellen Parameter (oder Operanden) einer Funktionsanwendung aus, bevor er die Funktion auf die Werte dieser Parameter anwendet. Diese Reihenfolge ist im Algorithmus dadurch vorgegeben, dass in Schritt 2 die Teilausdrücke $\mathtt{B}, \mathtt{A_1}, \ldots, \mathtt{A_n}$ ausgewertet werden und erst dann der Wert von `B` (der eine Funktion sein muss) auf die Werte von $\mathtt{A_1}, \ldots, \mathtt{A_n}$ angewandt wird.

Achtung: Was der Auswertungsalgorithmus dabei nicht festlegt, ist, in welcher Reihenfolge die Teilausdrücke $\mathtt{B}, \mathtt{A_1}, \ldots, \mathtt{A_n}$ ausgewertet werden, zum Beispiel in der Reihenfolge des Aufschreibens von links nach rechts oder von rechts nach links oder anders.

Im Falle des Ausdrucks `quadrat(quadrat(2))` kann diese Auswertungsreihenfolge wie folgt wiedergegeben werden:

```
quadrat(quadrat(2))

quadrat(2 * 2)

quadrat(4)

4 * 4

16
```

Eine andere Reihenfolge ist aber auch möglich, wie anhand desselben Beispiels leicht zu sehen ist:

```
quadrat(quadrat(2))

quadrat(2) * quadrat(2)

(2 * 2) * (2 * 2)

4 * 4

16
```

3.2.2 Auswertung in applikativer Reihenfolge

Die Auswertung entsprechend dem obigen Auswertungsalgorithmus — also Parameterauswertung vor Funktionsanwendung — hat die folgenden Namen, die alle dasselbe bezeichnen:

- Auswertung in applikativer Reihenfolge
- Inside-out–Auswertung
- Call-by-value–Auswertung
- strikte Auswertung

Die Bezeichnung Call-by-value ist unter SML-Experten verbreitet, aber für andere Programmiersprachen (wie Pascal und Modula) ist dieselbe Bezeichnung für eine andere Bedeutung gebräuchlich. (Für Programmiersprachen wie Pascal bedeutet „call by value“ eine Form des Prozeduraufrufes, bei der die Werte der Aufrufparameter an die Prozeduren weitergegeben werden, nicht deren Speicheradresse, so dass der Prozeduraufruf die Parameter nicht verändert. Als Alternative von „call by value“ kennen Programmiersprachen wie Pascal das „call by reference“, bei dem die Speicheradressen, d.h. die Referenzen, der Aufrufparameter weitergegeben werden.)

3.2.3 Auswertung in normaler Reihenfolge

Die Auswertung in der anderen Reihenfolge — also Funktionsanwendung vor Parameterauswertung — hat die folgenden Namen:

- Auswertung in normaler Reihenfolge
- Outside-in–Auswertung
- Call-by-name–Auswertung

Die Bezeichnung Call-by-name ist unter SML-Experten verbreitet, aber für andere Programmiersprachen ist dieselbe Bezeichnung für eine andere Bedeutung gebräuchlich. (In einigen alten Programmiersprachen bezeichnete „call by name“ die Weitergabe eines Bezeichners ohne Festlegung eines Wertes noch einer Speicheradresse, also eine rein textuelle Zeichenfolgenersetzung.)
Die Auswertung in applikativer Reihenfolge und die Auswertung in normaler Reihenfolge von einem Ausdruck liefern immer dasselbe Ergebnis, wenn alle Funktionsanwendungen des Ausdrucks terminieren. Bei nichtterminierenden Funktionsanwendungen ist dies nicht immer der Fall. Ist zum Beispiel `null` eine einstellige Funktion, die für jede ganze Zahl den Wert `0` liefert, und ist `f` eine einstellige nichtterminierende Funktion, also

```
fun null(x : int) = 0;
fun f(x : int) = f(x + 1);
```

so liefert eine Auswertung in normaler Reihenfolge von `null(f(1))` den Wert `0`, wogegen eine Auswertung in applikativer Reihenfolge desselben Ausdrucks nicht terminiert.

3.2.4 Vorteil der applikativen Reihenfolge gegenüber der normalen Reihenfolge

Am Beispiel des Ausdrucks `quadrat(quadrat(2))` erkennt man einen Vorteil der Auswertung in applikativer Reihenfolge gegenüber der Auswertung in normaler Reihenfolge:
Ein Parameter (oder Operand) wird bei einer Auswertung in applikativer Reihenfolge nur einmal ausgewertet, bei einer Auswertung in normaler Reihenfolge jedoch unter Umständen mehrmals.

3.3 Verzögerte Auswertung

3.3.1 Vorteil der normalen Reihenfolge gegenüber der applikativen Reihenfolge

Betrachten wir die einstellige konstante Funktion `null`, die zu jeder ganzen Zahl den Wert `0` liefert. In SML kann die Funktion `null` wie folgt deklariert werden:

```
fun null(x : int) = 0;
```

Vergleichen wir nun die Auswertung in applikativer Reihenfolge und in normaler Reihenfolge des Ausdrucks `null(quadrat(quadrat(quadrat(2))))`:

Auswertung in applikativer Reihenfolge

```
null(quadrat(quadrat(quadrat(2))))

null(quadrat(quadrat(2 * 2)))

null(quadrat(quadrat(4)))

null(quadrat(4 * 4))

null(quadrat(16))

null(16 * 16)

null(256)

0
```

Auswertung in normaler Reihenfolge

```
null(quadrat(quadrat(quadrat(2))))

0
```

Die Auswertung der Parameter (oder Operanden) vor der Funktionsanwendung durchzuführen, ist nur dann von Vorteil, wenn alle Parameter von der Funktion „verwendet“ werden, d.h. zur Berechnung des Wertes der Funktionsanwendung herangezogen werden. Sonst führt diese Auswertungsreihenfolge, die applikative Reihenfolge, zu nutzlosen Berechnungen.

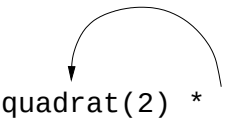
Verzögerte Auswertung

Die verzögerte Auswertung (*lazy evaluation*), auch *Call-by-need*–Auswertung genannt, ist eine Form der Auswertung, die Aspekte der Auswertung in applikativer Reihenfolge und der Auswertung in normaler Reihenfolge zusammenbringt.
Die Grundidee der verzögerten Auswertung ist einfach:

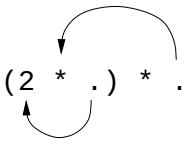
- Wie bei der Auswertung in normaler Reihenfolge führt die verzögerte Auswertung die Funktionsanwendung vor der Auswertung der Parameter (oder Operanden) durch.
- Bei der Funktionsanwendung werden aber alle bis auf ein Vorkommen eines Parameters durch einen Verweis auf ein einziges dieser Vorkommen ersetzt, damit dieser Parameter nicht mehrmals ausgewertet wird.

Die verzögerte Auswertung des Ausdrucks `quadrat(quadrat(2))` ist wie folgt:

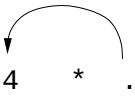
```
quadrat(quadrat(2))
```



```
quadrat(2) * .
```



```
(2 * .) * .
```



```
4   *   .

16
```

Dank des Verweises gilt der Wert `4` für beide Operanden der letzten Multiplikation, sobald das innere Produkt `2 * 2` berechnet wird.

Die verzögerte Auswertung des Ausdrucks `null(quadrat(quadrat(quadrat(2))))` ist wie folgt:

```
null(quadrat(quadrat(quadrat(2))))

0
```

Wie die Auswertung in applikativer Reihenfolge vermeidet die verzögerte Auswertung die mehrfache Auswertung von Parametern. Wie die Auswertung in normaler Reihenfolge vermeidet die verzögerte Auswertung die Auswertung von Parametern, die zur Auswertung einer Funktionsanwendung nicht notwendig sind.

Die verzögerte Auswertung hat also die Vorteile der beiden Grundansätze zur Auswertung und keinen ihrer Nachteile. Die verzögerte Auswertung scheint also die beste Auswertung zu sein.

Man beachte, dass das Substitutionsmodell zur Formalisierung der verzögerten Auswertung nicht passt, weil die verzögerte Auswertung auf einer Datenstruktur, den Zeigern oder dem Graph, beruht, die im Substitutionsmodell nicht vorhanden ist.

3.3.2 Verweis

Was ist aber ein Verweis (auch Zeiger, pointer oder Referenz genannt)? Oft sagt man: eine Adresse. Im Grunde liegt der Verwendung von Verweisen ein Berechnungsmodell zugrunde. Nach diesem Berechnungsmodell wird bei der Auswertung (von Ausdrücken) der Ausdruck, auf den ein Verweis zeigt, an der Stelle des Ursprungs des Verweises eingefügt.

3.3.3 Auswertungsreihenfolge von SML

SML verwendet die Auswertung in applikativer Reihenfolge. Der Grund dafür ist, dass die verzögerte Auswertung die folgenden Nachteile hat:

- Die verzögerte Auswertung verlangt eine ziemlich komplizierte — also zeitaufwendige — Verwaltung von Verweisen (Zeigern).
- Die verzögerte Auswertung lässt imperative (oder prozedurale) Befehle wie Schreibbefehle nur schwer zu, weil sie wie die Auswertung in normaler Reihenfolge den Zeitpunkt der Ausführung solcher Befehle für den Programmierer schwer vorhersehen lässt. (Auf diesen Aspekt soll zurückgekommen werden, wenn imperative Befehle von SML eingeführt werden.)
- In manchen Fällen verlangt die verzögerte Auswertung viel mehr Speicherplatz als die Auswertung in applikativer Reihenfolge.

Effiziente Implementierungsmöglichkeiten der verzögerten Auswertung werden seit einigen Jahren untersucht. Viele hoch interessante Ergebnisse sind erzielt worden. Auf dem Gebiet wird intensiv geforscht — siehe u.a. die Forschung um die Programmiersprache Haskell http://www.haskell.org/ und das Buch Simon L. Peyton Jones: The implementation of functional programming languages, Prentice-Hall, ISBN 0-13-453333-X, ISBN 0-13-453325-9 Paperback, 1987).

3.4 Auswertung der Sonderausdrücke

3.4.1 Wertdeklarationen (`val` und `fun`)

Die Auswertung einer Deklaration der Gestalt `val N = A` fügt die Gleichung `N = A` zu der Umgebung hinzu.

Als Spezialfall davon: Die Auswertung einer Deklaration der Gestalt `val N = fn P => A` fügt die Gleichung `N = fn P => A` zu der Umgebung hinzu. Eine Deklaration der Gestalt `val rec N = fn P => A` wird ebenso behandelt, aber mit zusätzlichen Vorkehrungen, damit die Umgebung von `A` auch diese Gleichung für `N` enthält.

Die Auswertung einer Deklaration der Gestalt `fun N P = A` fügt die Gleichung `N = fn P => A` zu der Umgebung hinzu, mit den gleichen Vorkehrungen wie bei `val rec N = fn P => A`.

Wird eine Gleichung zu der Umgebung hinzugefügt, so wird sie auf Korrektheit überprüft. Eine Deklaration wie z.B. `val zwei = zwei` wird dabei abgelehnt. Gleichungen aus Deklarationen werden in der Praxis in einen anderen Formalismus übersetzt, bevor sie in die Umgebung hinzugefügt werden. Diese Übersetzung ist aber zum Verständnis der Auswertung unwesentlich.

3.4.2 if-then-else

Die Schreibweise eines Ausdrucks `if A1 then A2 else A3` entspricht nicht der üblichen Schreibweise von Funktionen. Bei der besonderen Schreibweise von if-then-else-Ausdrücken handelt es sich lediglich um sogenannten „syntaktischen Zucker“, d.h. um

einen Zusatz, der den Kuchen schöner und (optisch) schmackhafter macht, den Kuchen jedoch prinzipiell nicht verändert. Man könnte einen if-then-else-Ausdruck `if A1 then A2 else A3` genauso gut in Präfixschreibweise `if-then-else(A1, A2, A3)` schreiben. Zur Verdeutlichung werden wir diese Schreibweise in diesem Abschnitt manchmal verwenden, auch wenn das keine korrekte SML-Syntax ist.

Nach dem Prinzip des Occam'schen Messers empfiehlt es sich, if-then-else-Ausdrücke ähnlich wie herkömmliche Funktionsanwendungen auszuwerten. Da SML auf der Auswertung in applikativer Reihenfolge beruht, hieße das, zur Auswertung eines Ausdrucks `if A1 then A2 else A3` zunächst alle drei Teilausdrücke `A1`, `A2` und `A3` auszuwerten.

Dieser Ansatz ergibt aber wenig Sinn, wie man sich am Beispiel der Fakultätsfunktion überzeugen kann:

$$(*)\quad \begin{array}{ll} n! = 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{andernfalls} \end{array}$$

Die Fakultätsfunktion ist auf den natürlichen Zahlen definiert. Sie kann also als eine partielle Funktion auf den ganzen Zahlen gesehen werden. In SML kann die Fakultätsfunktion wie folgt implementiert werden:

(**) `fun fak(n) = if n = 0 then 1 else n * fak(n-1);`

oder, in der Präfixschreibweise,

(***) `fun fak(n) = if_then_else( n=0, 1, n*fak(n-1) );`

Dass die Funktion nicht total ist, äußert sich darin, dass sie für gewisse Argumente keinen Wert liefert. Während zum Beispiel der Ausdruck `fak(0)` den Wert `1` hat, hat der Ausdruck `fak(~1)` keinen Wert. Die Auswertung von `fak(~1)` terminiert nicht.

Wenn ein Ausdruck `if A1 then A2 else A3`, oder `if_then_else(A1, A2, A3)` in Präfixschreibweise, in applikativer Reihenfolge genau so ausgewertet würde wie funktionale Ausdrücke, müssten zunächst alle drei Teilausdrücke `A1`, `A2` und `A3` ausgewertet werden. Das hätte zur Folge, dass zur Auswertung von `fak(0)` der Ausdruck `fak(~1)` ausgewertet werden muss und die Auswertung nicht terminieren würde, obwohl `fak(0)` ja den Wert `1` hat. Davon kann man sich mit Hilfe des Substitutionsmodells überzeugen:

```
fak(0)
if_then_else(0=0,  1, 0*fak(0-1) )
if_then_else(true, 1, 0*fak(~1) )
if_then_else(true, 1, 0*if_then_else(~1=0 , 1, ~1*fak(~1-1) ) )
if_then_else(true, 1, 0*if_then_else(false, 1, ~1*fak(~2) ) )
if_then_else(true, 1, 0*if_then_else(false, 1, ~1*if_then_else(...) ) )
...
```

Bei einer Auswertung in applikativer Reihenfolge könnte überhaupt keine rekursive Definition ausgewertet werden, weil immer zunächst alle darin vorkommenden Teilausdrücke ausgewertet werden müssten. Deshalb ist es notwendig, bestimmte Ausdrücke in anderer Weise auszuwerten. Diese Ausdrücke sind die Sonderausdrücke. Jede Programmiersprache mit Auswertung in applikativer Reihenfolge muss Sonderausdrücke enthalten.

Das besondere Verfahren zur Auswertung eines Sonderausdrucks der Form `if_then_else(A1, A2, A3)`, oder `if A1 then A2 else A3` in der üblichen Schreibweise, lautet wie folgt:

Werte von den drei Teilausdrücken `A1`, `A2`, `A3` zuerst nur `A1` aus.
Hat `A1` den Wert `true`, dann (und nur dann) wird `A2` ausgewertet (und `A3` wird nicht ausgewertet). Der Wert von `A2` wird als Wert des gesamten Ausdrucks geliefert.
Hat `A1` den Wert `false`, dann (und nur dann) wird `A3` ausgewertet (und `A2` wird nicht ausgewertet). Der Wert von `A3` wird als Wert des gesamten Ausdrucks geliefert.

Bei dieser Spezifikation handelt es sich um einen sehr präzisen (und auch sehr einfachen) Algorithmus. SML wertet wie die meisten Programmiersprachen if-then-else-Ausdrücke in dieser Weise aus. Der Grund dafür ist, dass rekursive Definitionen sonst nicht möglich wären, obwohl sie wie im Fall der Fakultätsfunktion oft wünschenswert sind. Im Fall der Funktion `fak` entspricht die Deklaration (**) bzw. (***) genau der mathematischen Definition (*).

3.4.3 Pattern Matching

Im Kapitel 2 haben wir das „Pattern Matching“ im Zusammenhang mit Deklarationen kennengelernt:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
              | 5 => true
              | 6 => true
              | 7 => true
              | 8 => true
              | 9 => true
              | _ => false;

(fn true => E1 | false => E2)
```

SML bietet das folgende Konstrukt an, das `if_then_else` in gewissem Sinn verallgemeinert und wie if-then-else-Ausdrücke in anderen Kontexten als Deklarationen verwendet werden kann:

```
case A of A1 => B1
        | A2 => B2
          ...
        | An => Bn
```

Die Auswertung eines Ausdrucks dieser Gestalt geschieht wie folgt.

Zunächst wird nur der Ausdruck `A` ausgewertet. Der Wert von `A` wird dann nacheinander mit den Mustern `A1`, ..., `An` „gematcht“. Ist `Ai` das erste Muster (*pattern*), das mit dem Wert von `A` „matcht“, so wird `Bi` ausgewertet und dessen Wert als Wert des `case`-Ausdrucks geliefert.

Das Pattern Matching in Deklarationen wird ähnlich ausgewertet.

In der obigen Beschreibung der Auswertung von *Pattern Matching*–Ausdrücken ist der Begriff „Pattern Matching“ nicht präzise definiert worden. Wir möchten es aber zunächst bei dieser intuitiven Erklärung belassen.

Das `case`-Konstrukt könnte als Grundlage zur Auswertung von anderen Sonderausdrücken dienen. Zum Beispiel könnte ein Ausdruck `if A1 then A2 else A3` auch so ausgewertet werden:

> Werte von den drei Teilausdrücken `A1`, `A2`, `A3` zunächst gar keinen aus. Konstruiere einen neuen Ausdruck:
>
> ```
> case A1 of true => A2 | false => A3
> ```
>
> Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des if-then-else-Ausdrucks.

Andere Sonderausdrücke, können in ähnlicher Weise auf `case` zurückgeführt werden. Dieses Zurückführen von Sonderausdrücken auf wenige andere Sonderausdrücke ist ein Beispiel für die Anwendung des Occam'schen Messers. In der Praxis werden Sonderausdrücke allerdings seltener durch Zurückführen auf andere Sonderausdrücke behandelt, sondern meistens durch maßgeschneiderte Auswertungsalgorithmen.

3.4.4 Die Boole'schen Operatoren `andalso` und `orelse`

Boole'sche Konjunktionen $A_1 \wedge A_2$ und Disjunktionen $A_1 \vee A_2$ können prinzipiell in zwei verschiedenen Weisen ausgewertet werden:

1. Die Teilausdrücke A_1 und A_2 werden zunächst ausgewertet und der Wert der Konjunktion oder Disjunktion wird entsprechend der folgenden Wahrheitstabelle aus den Werten von A_1 und von A_2 ermittelt:

A_1	A_2	$A_1 \wedge A_2$	$A_1 \vee A_2$
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

2. - Fall „$\wedge$“:
   Zunächst wird nur der erste Teilausdruck A_1 ausgewertet. Ist der Wert von A_1 true, so wird auch A_2 ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert. Ist der Wert von A_1 false, so wird false als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert (und A_2 wird nicht ausgewertet).
   - Fall „$\vee$“:
   Zunächst wird nur der erste Teilausdruck A_1 ausgewertet. Ist der Wert von A_1 true, so wird true als Wert des Ausdrucks $A_1 \vee A_2$ geliefert (und A_2 wird nicht ausgewertet). Ist der Wert von A_1 false, so wird A_2 ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \vee A_2$ geliefert.

Den zweiten Ansatz kann man wieder durch Zurückführen auf andere Sonderausdrücke realisieren. Zur Auswertung eines Ausdrucks der Form $A_1 \wedge A_2$ könnte man so vorgehen:

> Werte von den Teilausdrücken A_1, A_2 zunächst gar keinen aus. Konstruiere einen neuen Ausdruck
>
> `if` A_1 `then` A_2 `else false`
>
> Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des ursprünglichen Ausdrucks.

Auf welche der beiden möglichen Weisen sollten Konjunktion und Disjunktion am besten ausgewertet werden? Einiges spricht für den einen Ansatz, einiges für den anderen, und verschiedene Programmiersprachen beantworten diese Frage verschieden. SML wählt wie viele Programmiersprachen den 2. Ansatz. Die SML-Bezeichnungen „`andalso`“ und „`orelse`“ statt „`and`“ und „`or`“ sollen unterstreichen, dass diese Operatoren auf die zweite Weise ausgewertet werden.

Wie kann man erkennen, wie eine Programmiersprache die Boole'sche Konjunktion auswertet? Dazu reicht ein Aufruf wie der folgende in SML:

```
fun endlose_berechnung(n) : bool = endlose_berechnung(n + 1);
false andalso endlose_berechnung(0);
```

Terminiert er, dann wertet die Programmiersprache die Konjunktion nach dem 2. Ansatz aus, andernfalls nach dem 1. Ansatz.

3.4.5 Infixoperator-Deklarationen und Präzedenzen

Auch wenn man die Boole'schen Operatoren selbst definiert, hätte man vermutlich gern die Möglichkeit, sie in Infixschreibweise zu verwenden. Zu diesem Zweck bietet SML sogenannte Infixoperator-Deklarationen:

```
infix /\;
fun A1 /\ A2 = if A1 then A2 else false;

infix \/;
fun A1 \/ A2 = if A1 then true else A2;
```

Eine Infixoperator-Deklaration kann mit einer Präzedenz von 0 bis 9 ergänzt werden:

```
infix 6 /\;
```

Fehlt die Präzedenz, dann gilt der voreingestellte (default) Wert 0. Je größer die Präzedenz, desto stärker bindet der Operator.

Eine Infixoperator-Deklaration `infix n Op;` definiert den Operator `Op` als linksassoziativ. Eine Infixoperator-Deklaration `infixr n Op;` definiert den Operator `Op` als rechtsassoziativ.

Beispiel:

Die Funktion `potenz'` aus Abschnitt 2.11 kann wie folgt als rechtsassoziativer Infixoperator deklariert werden:

```
infixr 8 **;
fun a ** b = if b = 0 then 1
                      else if gerade(b)
                              then quadrat(a ** (b div 2))
                              else a * (a ** (b - 1));
```

3.4.6 Erweiterter Auswertungsalgorithmus mit Behandlung von Sonderausdrücken

In Abschnitt 3.4 wurde für einige Klassen von Sonderausdrücken, zum Beispiel die mit `if-then-else` gebildeten und die mit `andalso` gebildeten, jeweils ein Verfahren beschrieben, mit dem Sonderausdrücke dieser Klasse ausgewertet werden. Wenn wir alle diese Verfahren mit dem Sammelbegriff „Sonderalgorithmus“ bezeichnen und annehmen, dass die Sonderalgorithmen einfach die Werte der Bezeichner `if-then-else` usw. sind, kann der zu Beginn dieses Kapitels beschriebene Auswertungsalgorithmus wie folgt um eine Behandlung von Sonderausdrücken erweitert werden.

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Falls `A` atomar ist, dann:
   (a) Falls `A` vordefiniert ist, liefere den vordefinierten Wert von `A`. (Dieser kann auch ein Sonderalgorithmus oder eine Systemfunktion sein).
   (b) Andernfalls (der Wert von `A` ist in der Umgebung definiert) sei $\texttt{A} = \texttt{W}$ die Gleichung in der Umgebung, die den Wert von `A` definiert. Liefere `W` als Wert von `A` (wobei `W` auch eine Funktion $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_\texttt{k})$ `=> R` sein kann).
2. Andernfalls (`A` ist zusammengesetzt) hat `A` die Form $\texttt{B}(\texttt{A}_1, \ldots, \texttt{A}_\texttt{n})$ mit $\texttt{n} \geq 0$. Werte den Teilausdruck `B` aus.
   (a) Falls der Wert von `B` ein Sonderalgorithmus ist, dann: Wende ihn auf die Teilausdrücke $\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}$ an. Liefere den dadurch erhaltenen Wert als Wert von `A`.
   (b) Andernfalls (der Wert von `B` ist kein Sonderalgorithmus), werte die Teilausdrücke $\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}$ aus. Seien $\texttt{W}_1, \ldots, \texttt{W}_\texttt{n}$ die Werte der Teilausdrücke $\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}$.
      i. Falls der Wert von `B` eine Systemfunktion ist, dann: Wende sie auf $(\texttt{W}_1, \ldots, \texttt{W}_\texttt{n})$ an. Liefere den dadurch erhaltenen Wert als Wert von `A`.
      ii. Falls der Wert von `B` eine Funktion der Form $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_\texttt{n})$ `=> R` ist, dann: Werte `R` in der erweiterten Umgebung aus, die aus der aktuellen Umgebung und den zusätzlichen Gleichungen $\texttt{F}_1 = \texttt{W}_1$, …, $\texttt{F}_\texttt{n} = \texttt{W}_\texttt{n}$ besteht. Liefere den dadurch erhaltenen Wert als Wert von A (die Umgebung ist nun wieder die ursprüngliche).

3.5 Funktionale Variablen versus Zustandsvariablen

Die Namen (bzw. Bezeichner oder funktionale Variablen), die wir bisher verwendet haben, sind nur eine Art von Variablen, die Programmiersprachen anbieten. Variablen dieser Art werden wir von nun an „funktionale Variablen“ nennen. Andere geläufige Bezeichnungen für solche Variablen sind „logische Variablen“ oder kurz „Variablen“, wenn der Kontext eindeutig macht, welche Art von Variablen gemeint ist.

Viele Programmiersprachen verwenden eine andere Art von Variablen, die „Zustandsvariablen“. SML bietet sowohl funktionale wie Zustandsvariablen an.

In diesem Abschnitt wird zunächst an die Merkmale der funktionalen Variablen erinnert. Dann werden die Zustandsvariablen und die damit verbundenen Begriffe und Operationen erläutert. Abschließend wird die Programmierung mit Zustandsvariablen in SML eingeführt.

3.5.1 Funktionale Variablen

Hauptmerkmal der funktionalen Variablen ist, dass die Bindung eines Werts an eine funktionale Variable (in der bisher verwendeten Terminologie: die Bindung eines Werts an einen Namen) nicht verändert werden kann. Eine funktionale Variable kann also als eine benannte Speicherzelle (oder ein benannter Speicherbereich) angesehen werden, die zur Speicherung eines einzigen Wertes verwendet werden kann. Die folgende Sitzung schafft z.B. drei verschiedene solcher Speicherzellen:

```
- val z = 2;
val z = 2 : int

- fun quadrat(x : int) = x * x;

- val z = 5;
val z = 5 : int
```

In der vorangehenden SML-Sitzung ist die zweite Deklaration einer Variablen namens `z` ein Fall von „Wiederdeklaration eines Namens“ (oder „Wiederdeklaration einer Variable“) (siehe Abschnitt 2.7). Dabei bleibt die erste Speicherzelle namens `z`, die den Wert `2` beinhaltet, erhalten und eine zweite Speicherzelle erhält den Namen `z` und als Inhalt den Wert `5`.

Namenskonflikte nach Wiederdeklarationen einer Variable werden durch die folgende einfache Regel gelöst, deren Wirkung auf Variablen „Überschatten“ heißt:

> Nach einer Wiederdeklaration einer Variablen mit Namen `N` gilt der Name `N` nur noch für die zuletzt deklarierte Variable mit Namen `N`.

Liegt der Programmiersprache eine statische (oder lexikalische) Bindung (siehe Abschnitt 2.7) zu Grunde, dann sind Deklarationen, in deren definierenden Teilen der Namen `N` vorkommt und die vor der Wiederdeklaration ausgewertet wurden, von der Wiederdeklaration der Variable mit Namen `N` unbeeinflusst.

Zusammen machen das Überschatten und die statische (oder lexikalische) Bindung eine funktionale Programmiersprache „referenztransparent”: Der Wert eines Ausdrucks (dieser Sprache) wird nicht verändert, wenn „das Gleiche durch das Gleiche“, d.h. ein Teilausdruck durch seine Definition, ersetzt wird. Anders ausgedrückt: Eine Sprache ist genau

dann referenztransparent, wenn syntaktisch gleiche Ausdrücke in der gleichen Umgebung stets auch gleiche Werte haben.

3.5.2 Zustandsvariablen

Eine andere Art von Variablen, die zur Beschreibung von Algorithmen verwendet werden und in Programmiersprachen vorkommen, sind die „Zustandsvariablen“. Die grundlegende Idee einer Zustandsvariablen ist die einer benannten Speicherzelle (oder eines Speicherbereiches) mit *veränderbarem* Inhalt. Das Bild einer etikettierten Schublade trifft auf eine Zustandsvariable gut zu: Der Name der Variable entspricht dem Etikett, der Schubladeninhalt lässt sich verändern.

Ein Anwendungsbeispiel für eine Zustandsvariable ist ein Bankkonto. Eine Zustandsvariable `kto` habe als Inhalt zunächst den Wert 100 (EURO). Operationen ermöglichen, z.B. den (Zu-)Stand des Kontos `kto` um 25 EURO auf 125 EURO zunächst zu erhöhen (`einzahlen 25`), dann um 55 EURO auf 70 EURO zu verringern (`abheben 55`). Mit funktionalen Variablen ist eine solche Veränderung von Inhalten nicht möglich. Offenbar können Zustandsvariablen, d.h. Variablen mit veränderbaren Inhalten, zur Modellierung gewisser Rechenvorgänge nützlich, sogar unabdingbar, sein.

Zustand, Zustandsänderung und Zuweisung

Da Zustandsvariablen veränderbar sind, hängen ihre Inhalte vom Zeitpunkt ab, zu dem sie ermittelt werden. Ein „Zustand“ einer Menge von Zustandsvariablen, oder allgemeiner eines Systems, das auf Zustandsvariablen beruht, ist eine Menge von Bindungen (Variablenname, Variableninhalt), die zu einem Zeitpunkt gelten. Eine „Zustandsänderung“ ist eine Veränderung des Inhalts einer Zustandsvariablen, also auch des Zustandes.

Zustandsänderungen erfordern eine Operation, mit der der Inhalt einer Zustandsvariablen verändert werden kann. Diese Operation heißt üblicherweise „Zuweisung“. In vielen Programmiersprachen wird sie „`:=`“ notiert, auch in SML.

Zwei Arten von Zustandsvariablen

So gesehen ist der Begriff einer Zustandsvariable ziemlich natürlich und einfach. Eine Komplizierung liegt darin, dass Operationen auf Zustandsvariablen in zwei verschiedenen Weisen ausgedrückt werden, was zur Unterscheidung zwischen

- „Zustandsvariablen mit expliziter Dereferenzierung“ und
- „Zustandsvariablen ohne explizite Dereferenzierung“

führt. Zustandsvariablen mit expliziter Dereferenzierung werden auch „Referenzen“ genannt.

Referenzen und Dereferenzierung

Bei solchen Zustandsvariablen steht der Variablennamen stets für eine (symbolische) Speicheradresse (also für eine Art Schubladenetikett). Also ist ein Sprachkonstrukt notwendig, um den Inhalt der Schublade zu bezeichnen. In SML sind Zustandsvariablen Referenzen,

und der Operator „`!`“ ist dieses Konstrukt. Ist `v` eine SML-Referenz, so bezeichnet `!v` den Inhalt der Referenz.

Die Operation, die in SML mit „`!`“ ausgedrückt wird, heißt „Dereferenzierung“. Dereferenzieren bedeutet, den Inhalt einer Speicherzelle wiedergeben.

In SML wird `kto` als Referenz mit anfänglichem Wert von `100` wie folgt deklariert:

```
- val kto = ref 100;
```

Den Kontostand kann man dann so erhalten:

```
- !kto;
val it = 100 : int
```

Achtung: In diesem Text wird bewusst zwischen den Bezeichnungen „Wert“ und „Inhalt“ unterschieden. Der Ausdruck `!kto` hat den Wert `100`. Der Ausdruck `kto` dagegen hat nicht den Wert `100`. Der Wert von `kto` ist eine (symbolische) Referenz auf eine Speicherzelle, die den Inhalt `100` hat. Der Wert von `kto` ist das Etikett der Schublade, der Wert von `!kto` ist das, was in der Schublade drin ist. Dies äußert sich auch in den Typen: `!kto` hat den Typ `int`, aber `kto` hat den Typ `int ref`.

Unter Verwendung des Dereferenzierungsoperators „`!`“ und der Zuweisung „`:=`“ (ausgesprochen: *becomes*) wird der Konto(zu)stand wie folgt verändert:

```
- kto := !kto + 25;
val it = () : unit

- !kto;
val it = 125 : int

- kto := !kto - 55;
val it = () : unit

- !kto;
val it = 70 : int
```

Sequenzierung

Da der Zeitpunkt einer Zustandsveränderung für den Zustand von Belang ist, muss die Reihenfolge von Zustandsveränderungen festgelegt werden. Das Konstrukt „`;`“ drückt in SML wie in vielen Programmiersprachen eine Reihenfolge oder Sequenz aus. Der Ausdruck `(A1; A2)` wird so ausgewertet: zuerst wird `A1` ausgewertet, aber sein Wert wird ignoriert. Danach wird `A2` ausgewertet und sein Wert wird als Wert des Ausdrucks `(A1; A2)` geliefert. Zwischen den Klammern können beliebig viele durch „`;`“ getrennte Ausdrücke stehen, die dann in der gegebenen Reihenfolge ausgewertet werden, wobei nur der Wert des letzten zurückgeliefert wird.

Zustandsvariablen ohne explizite Dereferenzierung

In den frühen Jahren der Informatik haben sich Zustandsvariablen etabliert, die keiner expliziten Dereferenzierung bedürfen. Je nach Kontext, in dem ein Variablenname `v` vorkommt, steht er mal für eine Referenz (d.h. eine symbolische Speicheradresse, intuitiv: ein Schubladenetikett) mal für den Wert, der Inhalt der Speicheradresse `v` ist.

Ist `kto` eine Zustandsvariable ohne explizite Dereferenzierung, so wird in vielen Programmiersprachen der Konto(zu)stand wie folgt verändert:

```
kto := kto + 25;
kto := kto - 55;
```

Links vom Zuweisungsoperator „`:=`“ bezeichnet `kto` hier eine Speicheradresse, rechts einen Wert: Die Dereferenzierung auf der rechten Seite ist implizit. Viele Programmiersprachen verwenden Zustandsvariablen ohne explizite Dereferenzierung; z.B. Scheme, Pascal, Java.

Gleichheit

Was die Gleichheit zwischen Zustandsvariablen bedeutet, hängt davon ab, was für eine Art von Zustandsvariablen betrachtet wird.

Für Referenzen (also für Zustandsvariablen mit expliziter Dereferenzierung) bedeutet die Gleichheit natürlich die Gleichheit der Referenzen (also der Speicherzellen). Die folgende SML-Sitzung zeigt diese Interpretation der Gleichheit für Referenzen.

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref 5;
val w = ref 5 : int ref

- v = w;
val it = false : bool

- !v = !w;
val it = true : bool
```

Für Zustandsvariablen ohne explizite Dereferenzierung bezieht sich die Gleichheit üblicherweise auf die Inhalte. Die entsprechende Sitzung wäre (in einer Phantasiesprache) also:

```
- declare v = pointer to 5;
v = pointer to 5 : pointer to int;

- declare w = pointer to 5;
w = pointer to 5 : pointer to int;

- w = v;
val it = true : bool
```

Viele Sprachen mit expliziter Dereferenzierung verwenden die Bezeichnung „pointer“ („Zeiger“) für (symbolische) Speicheradressen.

Alias-Problem

Referenzen und Zeiger ermöglichen, dieselbe Zustandsvariable in verschiedener Weise zu benennen. In SML etwa:

```
- val w = ref 5;
val w = ref 5 : int ref

- val z = w;
val z = ref 5 : int ref
```

Was mit funktionalen Variablen kein Problem ist, kann mit Zustandsvariablen zu Unübersichtlichkeit führen — etwa, wenn die vorangehende Sitzung wie folgt fortgesetzt wird:

```
- w := 0;
val it = () : unit;

- w;
val it = ref 0 : int ref
```

Da `z` dieselbe Referenz wie `w` bezeichnet, verändert die Zuweisung `w := 0` nicht nur `w`, sondern auch `z` (genauer: den Inhalt). In großen Programmen kann die Verwendung solcher „Aliase“ für die Programmierer zu Problemen aufgrund der Unübersichtlichkeit führen.

Zyklische Referenzierung

Betrachten wir die SML-Sitzung:

```
-  val a = ref (fn(x : int) => 0);
val a = ref fn : (int -> int) ref

- !a(12);
val it = 0 : int
```

`!a` ist eine Funktion, die jede ganze Zahl auf `0` abbildet.

```
- val b = ref (fn(x : int) => if x = 0 then 1 else x * !a(x - 1));

- !b(0);
val it = 1 : int

- !b(3);
val it = 0 : int
```

`!b` ist eine Funktion, die `0` auf `1` und jede andere ganze Zahl auf `0` abbildet.

```
- a := !b;
val it = () : unit
```

```
- !b(0);
val it = 1 : int

- !b(3);
val it = 6 : int
```

Nach der Zuweisung `a := !b` ist nun `!b` die rekursive Fakultätsfunktion (total auf natürlichen Zahlen).
In Zusammenhang mit komplexen Datenstrukturen (siehe Kapitel 8) können zyklische Referenzierungen nützlich sein.

3.5.3 Zustandsvariablen in SML: Referenzen

In SML sind Zustandsvariablen Referenzen, also (symbolische) Speicheradressen.
Ist `v` eine SML-Referenz eines Objektes vom Typ `t`, so ist `t ref` der Typ von `v` (siehe die vorangehenden Beispielsitzungen).
Der Referenzierungsoperator von SML ist „`ref`":

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

Mit dem Dereferenzierungsoperator „`!`“ und den vorangehenden Deklarationen lässt sich der folgende Ausdruck bilden und auswerten:

```
- !w(!v);
val it = 0 : int
```

Der Sequenzierungsoperator von SML ist „`;`“. Der Wert einer Sequenz von Ausdrücken `(A1 ; A2 ; ...; An)` ist der Wert des letzten Ausdrucks, d.h. der Wert von `An`.
Der Zuweisungsoperator von SML ist „`:=`“ (becomes). Der Wert einer Zuweisung ist „`()`“ (unity).
SML liefert Referenzen, druckt sie aber nicht aus. Anstelle einer Referenz druckt SML das Symbol `ref` gefolgt von ihrem Inhalt:

```
- val v = ref 5;
val v = ref 5 : int ref

- v;
val it = ref 5 : int ref

-  val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- w;
val it = ref fn : (int -> int) ref
```

Dieses Druckverhalten von SML sollte nicht dazu verleiten, die Referenzen von SML für Zustandsvariablen ohne explizite Dereferenzierung zu halten!

Die Referenz einer Referenz ist selbstverständlich möglich:

```
-  val w = ref (fn x:int => 0);
val w = ref fn : (int -> int) ref

- val z = ref w;
val z = ref (ref fn) : (int -> int) ref ref

- !(!z)(9);
val it = 0 : int
```

Die Gleichheit für Referenzen in SML ist „=“.

3.6 Funktionale Programmierung versus Imperative Programmierung

Die Programmierung mit Zustandsvariablen nennt man imperative Programmierung. Die imperative Programmierung stellt ein Programmierparadigma dar, das sich wesentlich von der funktionalen Programmierung unterscheidet. Programmiersprachen wie SML bieten die Möglichkeit, beide Programmierparadigmen zu vermischen. Jedoch ist das Hauptparadigma solcher Sprachen nur eines von beiden Paradigmen. In diesem Abschnitt werden einige Unterschiede zwischen funktionaler und imperativer Programmierung angesprochen.

3.6.1 Überschatten versus Zustandsänderung

Überschatten und Zustandsänderungen sind zwei unterschiedliche Techniken. Das Überschatten verändert den Wert einer bereits existierenden Variable nicht, sondern verwendet deren Name für eine neue Speicherzelle. Eine Zustandsänderung erstellt keine neue Variable, sondern verändert den Inhalt einer bereits vorhandenen Speicherzelle.

Man beachte, dass moderne imperative Programmiersprachen auch das Überschatten kennen.

3.6.2 Funktion versus Prozedur

Eine Funktion kann auf Argumente angewandt werden und liefert einen Wert als Ergebnis, ohne dabei Zustandsveränderungen zu verursachen. Eine Funktion ist also referenztransparent.

Die Bezeichnung Prozedur ist ein Oberbegriff, der sowohl Funktionen umfasst als auch Programme, die Zustandsveränderungen verursachen. Eine Prozedur, die keine Funktion ist, die also Zustandsveränderungen verursacht, liefert je nach Programmiersprache entweder überhaupt keinen Wert als Ergebnis oder einen uninteressanten Wert wie `()` in SML.

In vielen Programmiersprachen kann man Prozeduren definieren, die keine Funktionen sind, weil sie Zustandsveränderungen verursachen, die aber Werte als Ergebnis liefern, als

seien sie Funktionen. Das ist oft schlechter Programmierstil. Man stelle sich ein Programm `ln` vor, das den Logarithmus zu einer Basis $b = e$ berechnet (wobei e die Euler'sche Zahl ist), das beim Aufruf `ln(`e`)` nicht nur den Wert `1.0` liefert, sondern nebenbei noch den Wert der Basis b verändern würde!

Prozeduren, die Werte liefern, die also in Ausdrücken an derselben Stelle wie Funktionen vorkommen können, werden häufig — fälschlich bzw. irreführend — Funktionen genannt.

3.6.3 Unzulänglichkeit des Substitutionsmodells zur Behandlung von Zustandsvariablen

Mit Zustandsvariablen und Zustandsveränderungen ist die Referenztransparenz durchbrochen. Der Wert eines Ausdrucks hängt nun vom Zeitpunkt ab, zu dem er ausgewertet wird. Das Substitutionsmodell ergibt keinen Sinn mehr. In der folgenden Sitzung hat der Ausdruck `!kto` einmal den Wert `150`, einmal den Wert `200`. Obwohl die Umgebung gleich bleibt, haben die syntaktisch gleichen Ausdrücke `!kto` und `!kto` verschiedene Werte.

```
- val kto = ref 100;
val kto = ref 100 : int ref

- kto := ! kto + 50;
val it = () : unit

- kto;
val it = ref 150 : int ref

-  kto := ! kto + 50 ;
val it = () : unit

- kto;
val it = ref 200 : int ref
```

Um Zustandsvariablen beschreiben zu können, ist ein anderes, wesentlich komplizierteres Berechnungsmodell nötig als das Substitutionsmodell. Dieses andere Berechnungsmodell heißt „Umgebungsmodell“. Es hat nicht die Einfachheit und die Eleganz des Substitutionsmodells.

3.6.4 Rein funktionale Programme und Ausdrücke

Funktionale Programme und Ausdrücke ohne Zustandsvariablen werden „rein funktional“ genannt. Mit Zustandsvariablen erweist sich der Beweis von Programmeigenschaften (wie z.B. Terminierung) als viel schwieriger als mit rein funktionalen Programmen, weil zusätzlich zu den Deklarationen die Veränderungen des Programmzustands berücksichtigt werden müssen. Dafür müssen die zeitlichen Programmabläufe berücksichtigt werden, wozu sogenannte „temporallogische“ Formalismen verwendet werden (siehe die Lehrveranstaltungen im Hauptstudium zu den Themen „Temporallogik“ und „Model Checking“).

3.6.5 Nebeneffekte

Die Zustandsveränderungen, die sich aus der Auswertung von nicht rein funktionalen Ausdrücken oder Programmen ergeben, werden „Nebenwirkungen“ oder „Nebeneffekte“ (*side effects*) genannt.

3.6.6 Reihenfolge der Parameterauswertung

Der Auswertungsalgorithmus und das abstraktere Substitutionsmodell, die am Anfang dieses Kapitels eingeführt wurden, legen die Reihenfolge der Auswertung der aktuellen Parameter $\mathtt{A_1}, \ldots, \mathtt{A_n}$ eines Ausdrucks $\mathtt{B}(\mathtt{A_1}, \ldots, \mathtt{A_n})$ nicht fest. Die aktuellen Parameter können von links nach rechts (also in der Reihenfolge $\mathtt{A_1}$, $\mathtt{A_2}$ bis $\mathrm{A_n}$) oder von rechts nach links (also in der Reihenfolge $\mathtt{A_n}$, $\mathtt{A_{n-1}}$, …, $\mathtt{A_1}$) oder in irgendeiner anderen Reihenfolge ausgewertet werden. Für rein funktionale Programme beeinflusst die Reihenfolge der Auswertung der aktuellen Parameter das Ergebnis nicht. Dies ist aber anders, wenn einige der aktuellen Parameter nicht rein funktional sind, weil ihre Auswertung Nebeneffekte hat, so dass der Zustand am Ende der Parameterauswertung von der Reihenfolge dieser Auswertung abhängt.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 4

Prozeduren zur Abstraktionsbildung

Dieses Kapitel ist dem Begriff „Prozedur“ gewidmet. Prozeduren sind Programmkomponenten, die die Definition von Teilberechnungen ermöglichen. Dank der sogenannten „Blockstruktur“ von modernen Programmiersprachen können Variablen lokal zu einer Prozedur deklariert werden. Mit lokalen Variablen zeigt das Überschatten (siehe Kapitel 3) seinen Nutzen. In diesem Kapitel wird auch der Begriff „(Berechnungs-) Prozess“ eingeführt, und Programme und Prozesse werden verglichen. Es wird gezeigt, dass die Rekursion die abstrakte Kontrollstruktur eines Programms oder einer Prozedur sein kann, ohne dass deshalb die Durchführung dieses Programmes oder dieser Prozedur ein rekursiver (Berechnungs-) Prozess sein muss. Schließlich werden die Vergleichsmaße (sogenannte Größenordnungen) zur Beschreibung des Ressourcenverbrauchs von Algorithmen eingeführt.

4.1 Die „Prozedur”: Ein Kernbegriff der Programmierung

4.1.1 Prozeduren zur Programmzerlegung

Die formale Spezifikation des Multiplikationsalgorithmus vom Abschnitt 1.1.6, d.h. die Definition der Funktion `integer_mult`, nimmt Bezug auf weitere Funktionen, nämlich auf die Funktionen `integer_digit_mult` (Multiplikation einer natürlichen Zahl mit einer einstelligen natürlichen Zahl), `one_digit_integer` (Test, ob eine natürliche Zahl einstellig ist), `digit_mult` (die Multiplikationstabelle), etc. Diese Funktionen stellen Teilalgorithmen dar, die in der Spezifikation des Multiplikationsalgorithmus verwendet werden.

Die Verwendung von Teilalgorithmen zur Spezifikation eines Algorithmus ist eine natürliche Vorgehensweise: Zur Lösung eines Problems lohnt es sich in der Regel, Teilprobleme zu erkennen, zu formalisieren und getrennt vom Hauptproblem zu lösen.

Aus diesem natürlichen Ansatz ist der Begriff „Prozedur“ entstanden, der ein Kernbegriff der Programmierung ist. Prozeduren sind Teilprogramme, die Teilaufgaben lösen. Jede Programmiersprache ermöglicht die Programmierung von Prozeduren, auch die sogenannten „niederen“ Maschinensprachen (siehe die Grundstudiumsvorlesung „Informatik 3“ und die Hauptstudiumsvorlesung „Übersetzerbau“), die einem Programmierer äußerst wenig Abstraktionsmöglichkeiten bieten.

Die Vorteile von Prozeduren sind vielseitig. Sie werden im Folgenden an Hand der formalen Spezifikation des Multiplikationsalgorithmus erläutert. (Zur Erläuterung der Vorteile

von Prozeduren spielt es keine Rolle, dass die formale Spezifikation des Multiplikationsalgorithmus kein Programm ist.)

4.1.2 Vorteile von Prozeduren

1. Prozeduren ermöglichen die Zerlegung eines Programms in übersichtliche Teilprogramme.

   Durch die Verwendung der Funktion `integer_digit_mult` wird die Spezifikation der Funktion `integer_mult` übersichtlicher.

   Jedes einzelne Teilprogramm kann leichter als das Gesamtprogramm spezifiziert, verfasst und auf Korrektheit überprüft werden.

2. Prozeduren ermöglichen die Mehrfachverwendung von identischen Programmteilen an verschiedenen Stellen eines Programms.

   Die Funktion `integer_digit_mult` wird zwei Mal in der Definition der Funktion `integer_mult` verwendet.

   Die Erkennung von mehrfachverwendbaren Teilen erleichtert sowohl die Spezifikation als auch die Implementierung und die Wartung von Programmen.

3. Prozeduren ermöglichen die Verwendung von sogenannten „lokalen Variablen“ mit präzise abgegrenzten Geltungsbereichen.

   Die Funktion `integer_digit_mult` verwendet die „lokalen Variablen“ `product1` und `product2` als Namen für Zwischenwerte, die bei der Berechnung des Wertes auftreten, den diese Funktion liefert. Lokale Variablen wie `product1` und `product2` werden „nach außen“ nicht „weitergegeben“.

   Die Verwendung von lokalen Variablen als Namen für konstante Zwischenwerte in Berechnungen trägt oft dazu bei, Berechnungen verständlicher zu machen. Lokale Variablen als Namen für konstante Werte sind auch nützlich, um Mehrfachberechnungen desselben Wertes zu vermeiden (siehe Beispiel unten).

   Lokale Variablen können nicht nur als Namen für konstante Werte dienen, sondern auch als Namen für Funktionen und Prozeduren, die als Hilfsmittel verwendet werden. Käme die Funktion `integer_mult` in einer komplexen Software vor, so wäre es vermutlich sinnvoll, dass die Funktion `integer_digit_mult` lokal zu der Funktion `integer_mult` deklariert würde und nicht global für die gesamte Software.

   Variablennamen, die in einer Prozedur lokal für konstante Werte oder für Prozeduren benutzt werden, stehen zu (anderen) Verwendungen außerhalb dieser Prozedur frei: Die Prozedur, in der lokale Variablen deklariert werden, ist der „Geltungsbereich“ dieser lokalen Variablen. Dadurch reicht es aus, nur über die Namen von nichtlokalen Variablen, die sogenannten „globale Variablen“, Buch zu führen, um Namenskonflikte zu vermeiden. Dadurch wird die Erstellung von komplexer Software und die Zusammenarbeit von mehreren Personen zur Erstellung derselben Software erheblich erleichtert.

4. Prozeduren sind austauschbare Programmkomponenten.

   Wie der Test `one_digit_integer` oder die Funktion `digit_mult`, d.h. die Multiplikationstabelle, implementiert ist, ist für die Definition der Funktion `integer_mult`, die diese Funktionen verwendet, unwichtig. Jede partiell korrekte bzw. total korrekte

Implementierung, die der natürlichsprachlichen Spezifikation dieser beiden Funktionen entspricht, kann verwendet werden: Sie ergibt eine partiell korrekte bzw. total korrekte Implementierung von `integer_mult`. So kann während der Programmentwicklung z.B. zuerst eine „einfache“ Implementierung von Prozeduren verwendet werden, die zu späteren Zeitpunkten durch „bessere“ Implementierungen ersetzt wird, ohne dass solche Änderungen die Korrektheit des Gesamtprogramms in Frage stellen (siehe die Hauptstudiumsvorlesungen über Software-Entwicklung).

4.1.3 Funktion versus Prozedur

Wir erinnern daran, dass nicht alle Prozeduren Funktionen sind (siehe Abschnitt 3.6). Funktionen liefern ein Ergebnis und haben keine Nebeneffekte. Prozeduren, die keine Funktionen sind, verursachen Nebeneffekte und liefern keine oder uninteressante Werte wie `()` in SML.

- Ist die verwendete Programmiersprache rein funktional (siehe Kapitel 3), dann sind alle Prozeduren Funktionen.
- Ist die Programmiersprache nicht rein funktional, dann sind manche Prozeduren keine Funktionen.

Jedoch wird oft bei rein funktionalen und auch bei nicht rein funktionalen Programmiersprachen von Funktionen anstelle von Prozeduren gesprochen, wenn es um Prozeduren (mit Nebeneffekten) geht, die einen Wert liefern. Dafür wird auch die (etwas bessere) Bezeichnung „Funktionsprozedur“ verwendet. Der Sprachgebrauch wechselt von Programmiersprache zu Programmiersprache, von Informatiker zu Informatiker ...

Dass der Unterschied zwischen Funktionen und Nebeneffekte verursachenden Prozeduren im Sprachgebrauch vieler Informatiker undeutlich wiedergegeben wird, zeigt lediglich, dass Eigenschaften von Prozeduren nicht immer formal untersucht und bewiesen werden. Im Falle von Funktionen ist dafür das Substitutionsmodell seiner Einfachheit wegen hervorragend. Um Eigenschaften von Nebeneffekte verursachenden Prozeduren zu beweisen, sind Ansätze notwendig, die viel komplizierter als das Substitutionsmodell sind.

4.1.4 Definition von Funktionen und Prozeduren in SML

SML bietet zwei alternative Schreibweisen zur Definition von Funktionen oder Prozeduren ohne Pattern Matching:

```
val rec fak   =   fn n => if n = 0 then 1 else n * fak(n-1);

fun fak(n)    =   if n = 0 then 1 else n * fak(n-1);
```

Die zweite Schreibweise ist nur „syntaktischer Zucker“ für die erste.

SML bietet ebenfalls zwei alternative Schreibweisen zur Definition von Funktionen oder Prozeduren mit Pattern Matching:

```
val rec fak   =   fn 0 => 1
                   | n => n * fak(n-1);
```

```
fun fak(0)    =   1
  | fak(n)    =   n * fak(n-1);
```

Die zweite Schreibweise ist nur „syntaktischer Zucker“ für die erste.

In SML gibt es Funktionen wie zum Beispiel die Funktion `abs` auf ganzen Zahlen, aber auch einige Prozeduren, die Nebeneffekte haben, wie die Prozedur `use` zum Laden einer Datei. Es ist eine Konvention von SML, dass solche Prozeduren stets den Wert `()` vom Typ `unit` als Ergebnis liefern. Zur Terminologie Funktion/Prozedur siehe auch Abschnitt 3.6.2.

Eine Prozedur, die von der Hardware die Uhrzeit erhält und als Wert liefert, verursacht keinen Nebeneffekt. Sie ist aber wohl keine Funktion, weil sie nicht referenztransparent ist: Der Wert, den sie liefert, ist bei gleichen aktuellen Parameter nicht immer derselbe. Dieses Beispiel zeigt, dass die Unterscheidung zwischen Prozeduren, die keine Funktionen sind, und Funktionen je nach dem, ob die Prozeduren Nebeneffekte verursachen, nicht ganz zutreffend ist. Fälle wie die Uhrzeit liefernde Prozedur sind aber seltene Grenzfälle, so dass diese Unterscheidung ihre Nützlichkeit behält.

4.2 Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen

Prozeduren sind u.a. zur Definition von Zwischenergebnissen und von Teilberechnungen nützlich. Lokale Deklarationen ermöglichen die Verwendung von Variablen als Namen für konstante Werte oder für Funktionen oder Prozeduren innerhalb einer Prozedur, die nur innerhalb dieser Prozedur verwendet werden können.

Lokale Deklarationen sind in (fast) allen modernen „höheren“ Programmiersprachen möglich, d.h. in (fast) allen Programmiersprachen, die heute verwendet werden. Maschinensprachen ermöglichen keine echten lokalen Deklarationen in dem Sinne, dass sie nicht sicherstellen, dass die Geltungsbereiche geschützt sind.

In diesem Abschnitt werden die lokalen Deklarationen von SML eingeführt.

SML bietet zwei syntaktische Möglichkeiten, Namen lokal zu deklarieren:

- Zum einen mit dem Ausdruck „`let`“,
- zum anderen mit dem Ausdruck „`local`“.

4.2.1 Lokale Deklarationen mit „`let`“

`let`-Ausdrücke werden verwendet, um Deklarationen lokal zu einem Ausdruck zu deklarieren, der selbst keine Deklaration ist.

Betrachten wir die folgende Funktion

$$\begin{array}{rccl} f: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & (3x+1)^2 + (5x+1)^2 + (3x)^3 + (5x)^3 \end{array}$$

Eine unmittelbare Implementierung in SML sieht wie folgt aus:

```
fun hoch2(x : int) = x * x;

fun hoch3(x : int) = x * x * x;

fun f(x) = hoch2(3*x + 1) + hoch2(5*x + 1) + hoch3(3*x) + hoch3(5*x);
```

Kommen die Funktionen `hoch2` und `hoch3` nur in der Definition von `f` vor, so bietet es sich an, diese Funktionen wie folgt lokal zu `f` zu deklarieren:

```
fun f(x) = let fun hoch2(x : int) = x * x
               fun hoch3(x : int) = x * x * x
           in
               hoch2(3*x + 1) + hoch2(5*x + 1) + hoch3(3*x) + hoch3(5*x)
           end;
```

Die Syntax (d.h. Schreibweise) des definierenden Teils (oder Rumpfs) einer Funktion- oder Prozedurdeklaration mit lokalen Deklarationen ist also:

```
let ... in ... end;
```

Andere Syntax für mehrere lokale Deklarationen

Zwischen `let` und `in` finden sich eine oder mehrere Deklarationen (in der üblichen Syntax), die mit „`;`“ getrennt werden dürfen (aber nicht müssen). Auch die folgende Syntax ist also zulässig:

```
fun f(x) = let fun hoch2(x : int) = x * x;      (* Zeichen ; beachten *)
               fun hoch3(x : int) = x * x * x
           in
               hoch2(3*x + 1) + hoch2(5*x + 1) + hoch3(3*x) + hoch3(5*x)
           end;
```

Warum SML die Verwendung von „`;`“ zwischen lokalen Deklarationen ermöglicht, wird im Abschnitt 4.2.8 erläutert.

Lokale Deklaration in Definitionen von anonymen Funktionen

Unter Verwendung einer anonymen Funktion (SML-Konstrukt `fn`, gesprochen lambda ($= \lambda$)) kann die Funktion `f` in SML wie folgt implementiert werden:

```
val f = fn (x) =>
           let fun hoch2(x : int) = x * x
               fun hoch3(x : int) = x * x * x
           in
               hoch2(3*x + 1) + hoch2(5*x + 1) + hoch3(3*x) + hoch3(5*x)
           end;
```

In anonymen Funktionen kann `let` genau so benutzt werden wie in benannten Funktionen.

Verbesserungen des Programmbeispiels

Die obigen Implementierungen der Funktion `f` können wie folgt verbessert werden.
Zum einen kann an Stelle der lokalen Funktion `hoch2` eine Funktion `hoch2plus1` verwendet werden, was zu einem verständlicheren, weil kompakteren, Programm führt:

```
fun f(x) = let fun plus1hoch2(x : int) =
                        let  fun hoch2(x) = x * x
                        in   hoch2(x + 1)
                        end
               fun hoch3(x : int)   =  x * x * x
           in
               plus1hoch2(3*x) + plus1hoch2(5*x) + hoch3(3*x) + hoch3(5*x)
           end;
```

Zum anderen kann die mehrfache Berechnung von `3*x` und von `5*x`, die bei komplizierteren Ausdrücken kostspielig sein kann, mit Hilfe von lokalen Variablen für Zwischenergebnisse vermieden werden:

```
(*)
   fun f(x) = let fun plus1hoch2(x : int) = (x + 1) * (x + 1)
                  fun hoch3(x : int)      = x * x * x
                  val x3 = 3*x
                  val x5 = 5*x
              in
                  plus1hoch2(x3) + plus1hoch2(x5) + hoch3(x3) + hoch3(x5)
              end;
```

Anmerkung: Manche Übersetzer erkennen in einigen Fällen mehrfache Berechnungen und führen eine ähnliche Änderung durch wie in diesem Beispiel.

Verschachtelte lokale Deklarationen

Verschachtelte lokale Deklarationen sind selbstverständlich möglich wie etwa in der vorangehenden Deklaration der Funktion `f` oder in der folgenden Deklaration:

```
     fun h(x) = let
                    fun g(y) = let
                                   fun hoch2(z : int) = z * z
                               in
                                   hoch2(y) + hoch2(y + 1)
                               end
                in
                    g(x) * g(x + 1)
                end;
```

4.2.2 Lokale Deklarationen mit „`local`“

`local`-Ausdrücke werden verwendet, um Deklarationen lokal zu Deklarationen zu deklarieren.

Die Funktion

$$\begin{array}{rccl} f: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & (3x+1)^2 + (5x+1)^2 + (3x)^3 + (5x)^3 \end{array}$$

kann ähnlich wie in `(*)` wie folgt deklariert werden:

```
local fun plus1hoch2(x : int) = x * x + 1
      fun hoch3(x : int)      = x * x * x
in
      fun f(x) =
          let val x3 = 3 * x
              val x5 = 5 * x
          in
              plus1hoch2(x3) + plus1hoch2(x5) + hoch3(x3) + hoch3(x5)
          end
end
```

Die Syntax eines local-Ausdrucks ist: `local ... in ... end`

Die lokalen Deklarationen von `x3` und `x5` können *nicht* zwischen `local` und dem ersten `in` stehen, weil dieser Bereich außerhalb des Geltungsbereiches des Parameters `x` liegt: Zwischen `local` und dem ersten `in` ist `x` unbekannt.

Es ist möglich, zwischen `in` und `end` noch eine weitere Funktion `f'` zu deklarieren, die ebenfalls die lokalen Definitionen `plus1hoch2` und `hoch3` verwendet. Bei einer Konstruktion wie in `(*)` ist es nicht möglich, lokale Deklarationen für mehrere Funktionen gemeinsam zu verwenden.

4.2.3 Unterschied zwischen `let` und `local`

`let` ermöglicht Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind, wie etwa in:

```
let val zwei = 2
    val drei = 3
in
    zwei + drei
end;
```

`let` ermöglicht keine Deklarationen lokal zu Ausdrücken, die selbst Deklarationen sind. Die folgenden Ausdrücke sind inkorrekt:

```
let val zwei = 2
    val drei = 3
in
    val fuenf = zwei + drei                    (* inkorrekt *)
end;

let val zwei = 2
    val drei = 3
```

```
in
    fun f(x) = x + zwei + drei                    (* inkorrekt *)
end;
```

In solchen Fällen kann `local` verwendet werden:

```
local val zwei = 2
      val drei = 3
in
      val fuenf = zwei + drei
end;

local val zwei = 2
      val drei = 3
in
      fun f(x) = x + zwei + drei
end;
```

Damit ist nach dem ersten Beispiel die Variable `fuenf` mit Wert `5` deklariert. Die Variablen `zwei` und `drei` sind dagegen nicht deklariert. Sie haben nur als internes Hilfsmittel bei der Deklaration von `fuenf` gedient. Nach dem zweiten Beispiel ist ganz entsprechend nur die Funktion `f` deklariert.

`local` ermöglicht wiederum keine Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind. Der folgende Ausdruck ist inkorrekt:

```
local val zwei = 2
      val drei = 3
in
      zwei + drei                                 (* inkorrekt *)
end;
```

4.2.4 Blockstruktur und Überschatten

Programmiersprachen, in denen Variablen als Namen für konstante Werte oder für Funktionen oder Prozeduren lokal deklariert werden können, besitzen die sogenannte „Blockstruktur“ und werden „Blocksprachen“ genannt.

Die Blockstruktur ist ein wichtiges Mittel zur Strukturierung von Programmen. Sie ermöglicht u.a., dass Programmierer, die gemeinsam an derselben komplexen Software arbeiten, sich nur über wenige gemeinsam benutzte Bezeichner absprechen müssen. Deklariert z.B. ein Programmierer einen Namen `N` lokal zu einem Programmteil, den er implementiert, so kann derselbe Name `N` von einem anderen Programmierer lokal zu einem anderen Programmteil mit einer ganz anderen Bedeutung verwendet werden, wie z.B. in:

```
- fun f(x) = let val a = 2
             in
                 a * x
             end;
val f = fn : int -> int
```

```
- fun g(x) =  let val a = 2000
              in
                 a * x
              end;
val g = fn : int -> int

- f(1);
val it = 2 : int

- g(1);
val it = 2000 : int
```

Ein `let`- oder `local`-Ausdruck stellt einen sogenannten „Block“ dar. In Programmiersprachen mit Blockstruktur erfolgt die Bindung von Werten an Variablen statisch (siehe Abschnitt 2.7).

4.2.5 Festlegung der Geltungsbereiche von Namen — Einführung

In SML ist ein „Block“ einfach ein Ausdruck. Warum bei einer solchen einfachen Definition die Bezeichnung „Block“ verwendet wird, liegt daran, dass der Blockbegriff ursprünglich in Zusammenhang mit imperativen Programmiersprachen eingeführt wurde, bei denen Blöcke sich nicht so einfach definieren lassen.

Ein Block ist also eine rein syntaktische Einheit, die weder von den aktuellen Parametern einer Funktionsanwendung noch vom aktuellen Zustand der Umgebung abhängt.

Der Geltungsbereich eines Namens (oder einer Variablen) in Blocksprachen erfolgt nach den folgenden einfachen Prinzipien:

1. Der Geltungsbereich eines Namens `N` ist der Block, der die Deklaration von `N` enthält, mit Ausnahme aller in dem Block vorkommenden Blöcke, die Deklarationen desselben Namens `N` enthalten.

2. Kommt ein Name `N` in einem Block `B` vor und ist `N` in `B` nicht deklariert, so gilt:

   (a) `N` muss in einem Block `B'` deklariert sein, der `B` umfasst.

   (b) Wenn mehrere Blöcke, die `B` umfassen, Deklarationen für `N` enthalten, so gilt die Deklaration desjenigen Blocks, der `B` am engsten umfasst.

Dass ein Block eine rein syntaktische Einheit ist, entspricht dem Prinzip der statischen Bindung — siehe Abschnitt 2.7. Blocksprachen beruhen also immer auf der statischen Bindung. Die Bezeichnung „lexikalische Bindung“ anstelle von „statische Bindung“ hebt diese Tatsache hervor.

4.2.6 Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung

Die Verwaltung einer Umgebung als eine geordnete Liste von Gleichungen der Gestalt `Name = Wert` (siehe Abschnitt 2.7) ermöglicht eine einfache Implementierung der vorangehenden Regeln:

- Beim Eintritt in einen neuen Block, d.h. bei der Auswertung eines neuen Ausdrucks, führt jede Deklaration eines Namens `N` für einen Wert `W` zu einem Eintrag `N = W` am Anfang der Umgebung.
- Beim Austritt aus einem Block, d.h. bei der Beendigung der Auswertung eines Ausdrucks, werden alle Einträge der Gestalt `N = W` gelöscht, die während der Auswertung des Ausdrucks in die Umgebung eingefügt wurden.
- Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. Die zuerst gefundene Gleichung `N = W` gilt als Definition von `N`. So gilt als Wert eines Namens `N` der Wert, der bei der „letzten“ (also „innersten“) Deklaration von `N` angegeben wurde. Dadurch werden „ältere“ Deklarationen eines Namens `N` „überschattet“ (siehe Abschnitt 2.7).

4.2.7 Überschatten durch verschachtelte lokale Deklarationen

Lokale Deklarationen ermöglichen das Überschatten von Deklarationen.

```
- fun f(x) = let val a = 2
                 fun g(x) = let val a = 3
                            in
                                a * x
                            end
             in
                 a * g(x)
             end;
val f = fn : int -> int

- f(1);
val it = 6 : int
```

Die Anwendung des Substitutionsmodell (siehe Kapitel 2) auf den Ausdruck `f(1)` liefert die Erklärung für den Wert dieses Ausdrucks:

```
f(1)
a * g(1)
2 * g(1)
2 * (a * 1)
2 * (3 * 1)
2 * 3
6
```

Bei einer solchen Anwendung des Substitutionsmodells können sowohl die Regeln zur Festlegung der Geltungsbereiche von Namen aus Abschnitt 4.2.5 als auch der Algorithmus zur Verwaltung der Umgebung aus Abschnitt 4.2.6 verwendet werden. Mit der Einführung von lokalen Variablen verliert also das Substitutionsmodell einen Teil seiner Einfachheit.

Selbstverständlich ist das Überschatten der ersten Deklaration von `a` im vorangehenden Beispiel nicht notwendig. Man hätte zum Beispiel das innerste `a` auch `b` nennen können,

so dass keine Namensgleichheit mit der Deklaration des äußeren Blocks auftritt. Es ist empfehlenswert, das Prinzip des Überschattens nicht so exzessiv zu verwenden, dass Missverständnisse bei Lesern der Programme provoziert werden.

4.2.8 Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung — Fortsetzung

Auswertung von val-Deklarationen

Wird eine Deklaration `val N = A` ausgewertet, so geschieht dies — wie die Auswertung jedes Ausdrucks auch — in einer Umgebung U. Dabei wird zunächst der Ausdruck `A` in dieser Umgebung U ausgewertet. Erst nach der Auswertung des Ausdrucks `A` in U wird die Umgebung U (an ihrem Anfang) um die Gleichung `N = A` erweitert.

Zur Auswertung des Rumpfes (oder definierenden Teils) einer Deklaration gilt also noch die „alte Umgebung“ vor Berücksichtigung dieser Deklaration.

Eintrag einer Funktionsdeklaration in die Umgebung

Zur Auswertung einer Funktionsdeklaration wie

(*) `val N = fn P => W`

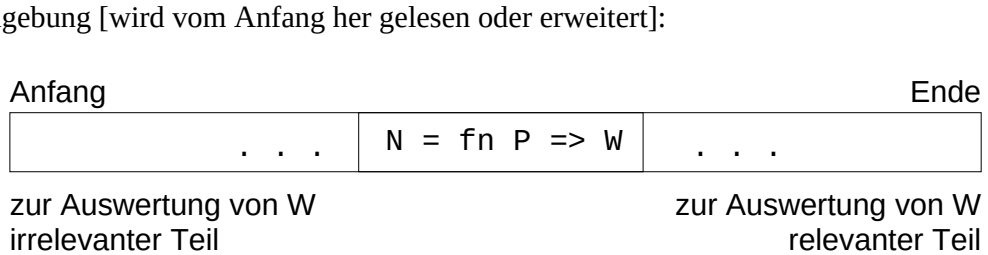
wird eine Gleichung

(**) `N = fn P => W`

zu der Umgebung (am Anfang) hinzugefügt. Zum Zeitpunkt der Auswertung der Funktionsdeklaration (*) wird der Ausdruck `W` nicht ausgewertet. Dies ergäbe keinen Sinn, weil `W` der Rumpf (oder definierende Teil) der Funktion namens `N` ist. Zum Zeitpunkt der Auswertung der Funktionsdeklaration (*) sind keine aktuellen Parameter bekannt, daher ist auch keine Anwendung der Funktion namens `N` durchzuführen.

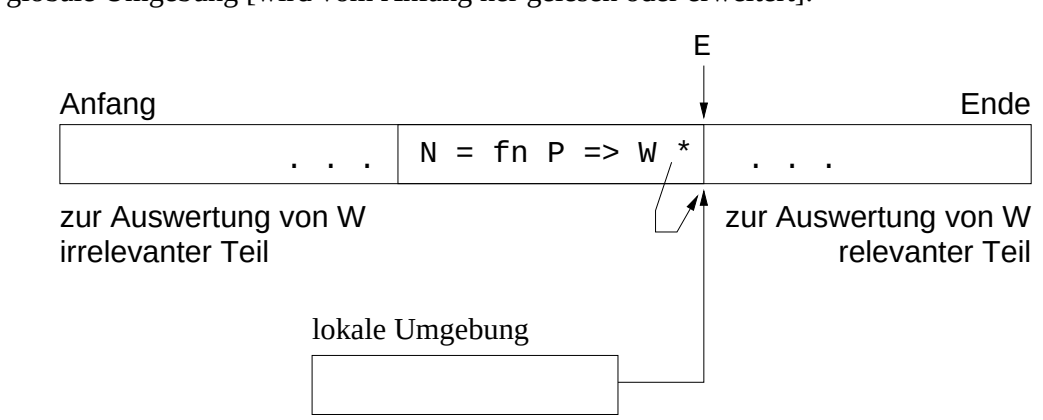
Zur (späteren) Auswertung einer Anwendung der Funktion namens `N` wird aber `W` ausgewertet. Die Frage stellt sich, welche Umgebung für diese Auswertung berücksichtigt werden soll.

Die Umgebung ist eine geordnete Liste, so dass sowohl nach wie vor der Gleichung (**) `N = fn P => W`, die den (Funktions-)Wert des Namens `N` liefert, weitere Gleichungen vorkommen können. Weitere Gleichungen kommen vor (**) vor, wenn weitere (”neuere“) Deklarationen nach der Deklaration von `N` stattgefunden haben. Die Blockstruktur (und das daraus folgende Prinzip der statischen Bindung) verlangt, dass nur die Gleichungen, die nach (**) vorkommen, also die „älter“ als die Deklaration von `N` sind, zur Auswertung von `W` berücksichtigt werden. So ergibt sich das folgende Bild einer Umgebung:



Auswertung einer Funktionsanwendung — Lokale Umgebung

Zur Auswertung einer Anwendung der Funktion namens `N` müssen aber auch die aktuellen Parameter der Funktionsanwendung berücksichtigt werden. Der Teil der Umgebung, der zur Auswertung von `W` relevant ist, wird also um eine sogenannte „lokale Umgebung“ erweitert:



Dabei stellt `*` einen Verweis dar (auf dessen Funktion wir etwas später eingehen werden).

Die Umgebung, die während der Auswertung einer Anwendung der Funktion namens `N` berücksichtigt wird, besteht also aus der lokalen Umgebung gefolgt von dem Teil der (nichtlokalen, auch global genannten) Umgebung, die zur Auswertung von `W` relevant ist. Der Verweis `E` gibt die Stelle an, ab der die Umgebung während der Auswertung einer Anwendung der Funktion mit dem Namen `N` gelesen wird.

Da jede Prozedurdeklaration lokale Deklarationen beinhalten kann, kann während einer Auswertung eine Verkettung von mehreren lokalen Umgebungen entstehen. Diese Verkettung spiegelt die Schachtelung von Ausdrucken wider, in denen Parametern von Funktionsanwendungen oder lokale Variablen deklariert werden.

Sequenzielle Auswertung von lokalen `let`-Deklarationen

Aufeinanderfolgende Deklarationen werden in der gegebenen Reihenfolge ausgewertet.

```
- val x = 2;
val x = 2 : int

- let val x = 3 * x;
      val x = 5 * x
  in
      1 + x
  end;
val it = 31 : int
```

Der `let`-Ausdruck wirkt also genau wie der folgende, in dem alle Namenskonflikte durch Umbenennung beseitigt sind:

```
- val x = 2;
```

```
val x = 2 : int

- let val x3 = 3 * x;
      val x5 = 5 * x3
  in
      1 + x5
  end;
val it = 31 : int
```

Nichtsequenzielle Auswertung von lokalen Deklarationen

Es gibt noch eine Variante von `let`, bei der aufeinanderfolgende Deklarationen gemeinsam behandelt werden. Zuerst werden alle Ausdrücke in den Rümpfen der Deklarationen in derselben äußeren Umgebung ausgewertet, danach werden alle berechneten Werte gleichzeitig an ihre Namen gebunden.

```
- val x = 2;
val x = 2 : int

- let val x  = 3 * x
      and x5 = 5 * x
  in
      x * x5
  end;
val it = 60 : int
```

An die lokale Variable `x5` wird also der Wert `10` gebunden und nicht der Wert `30`, wie es der Fall wäre, wenn `and` durch `val` ersetzt würde.

Das Konstrukt „`and`“ kann selbstverständlich auch in `local`-Ausdrücken verwendet werden.

Die nichtsequenzielle Auswertung von lokalen Deklarationen mit dem `and`-Konstrukt erklärt, warum die lokalen Deklarationen einer Sequenz, die sequenziell ausgewertet werden soll, mit „`;`“ getrennt werden können (siehe Abschnitt 4.2.1): In SML wie in vielen Programmiersprachen drückt „`;`“ die Sequenzierung von Ausdrücken aus (siehe Abschnitt 3.5.2).

Wechselseitige Rekursion

Zwei oder mehrere Funktionen heißen „wechselseitig rekursiv“, wenn sie sich wechselseitig aufrufen. Das `and`-Konstrukt ist für die Deklaration von wechselseitig rekursiven Funktionen unabdingbar, weil die Rümpfe aller Deklarationen in derselben Umgebung ausgewertet werden müssen.

Ein einfaches Beispiel für zwei wechselseitig rekursive Funktionen ist:

```
val rec ungerade = fn 0 => false
                    | 1 => true
                    | n => gerade(n-1)
and
```

```
gerade   = fn 0 => true
           | 1 => false
           | n => ungerade(n-1);
```

oder unter Verwendung des `fun`-Konstrukts:

```
fun ungerade(0) = false
   | ungerade(1) = true
   | ungerade(n) = gerade(n-1)
and
     gerade(0) = true
   | gerade(1) = false
   | gerade(n) = ungerade(n-1);
```

Auswertung von `val-rec`–Deklarationen

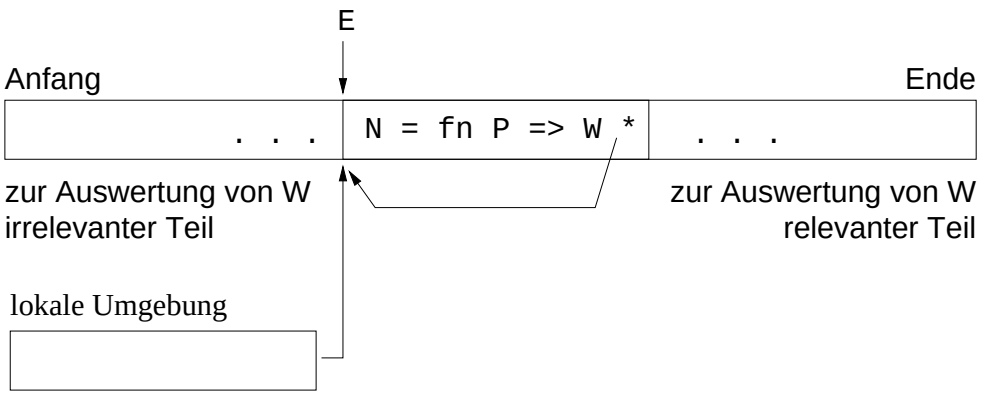
Eine Funktionsdeklaration der Form

```
val rec N = fn P => W
```

wird ähnlich wie eine Funktionsdeklaration der Form

```
val N = fn P => W
```

behandelt mit dem Unterschied, dass der Verweis am Ende der lokalen Umgebung *vor* statt hinter die Gleichung `N = fn P => W` in der globalen Umgebung zeigt:



Der Verweis `E` gibt die Stelle an, ab dem die Umgebung während der Auswertung einer Anwendung der rekursiven Funktion namens `N` gelesen wird.

Auswertung von `fun`-Deklarationen

Ein Funktionsdeklaration der Form

```
fun N P = W
```

wird genauso wie die Funktionsdeklaration

```
val rec N P = W
```

ausgewertet. Der Konstrukt „`fun`“ ist lediglich „syntaktischer Zucker“.

4.3 Prozeduren versus Prozesse

4.3.1 Notwendigkeit der Rekursion

In einer rein funktionalen Programmiersprache ist die Rekursion die einzige Möglichkeit, eine Anzahl von Wiederholungen zu ermöglichen, die von den aktuellen Parametern abhängt. Betrachten wir z.B. die folgende Funktion:

$$\begin{aligned} \text{summe}(0) &= 0 \\ \text{summe}(n) &= n + \ldots + 0 \quad \text{falls } n \in \mathbb{N} \text{ und } n \geq 1 \end{aligned}$$

Sie lässt sich in SML (mit Pattern Matching in der „syntaktisch verzuckerten“ Form) wie folgt rekursiv implementieren:

```
fun summe(0) = 0
  | summe(n) = n + summe(n-1);
```

Ohne Rekursion müsste man die Anzahl der Summanden unabhängig von dem formalen Parameter `n` festlegen, wie etwa in der folgenden unvollständigen und inkorrekten Deklaration:

```
fun summe'(0) = 0
  | summe'(1) = 1 + 0
  | summe'(2) = 2 + 1 + 0
  | summe'(3) = 3 + 2 + 1 + 0
  | summe'(4) = 4 + 3 + 2 + 1 + 0
  | ???
```

Kann man diese Deklaration vollenden, so dass `summe'` auf der unendlichen Menge der natürlichen Zahlen total ist, ohne `summe'` oder irgend eine rekursive Funktion im definierenden Teil der Deklaration von `summe'` zu verwenden? Mit den bisher eingeführten Mitteln ist das unmöglich.

4.3.2 Rekursion versus Iteration: Grundkonstrukte beider Berechnungsmodelle im Vergleich

Die Rekursion ist nicht die einzige Möglichkeit, Wiederholungen zu spezifizieren. Im Abschnitt 1.1.3 wurde ein rekursiver Algorithmus angegeben, mit dem ein Viereck auf den Boden gezeichnet werden kann:

Viereckalgorithmus 1:

> Zeichne eine Seite wie folgt:
>
> Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts.
> Wenn Du nicht am Startpunkt stehst, dann zeichne eine Seite unter Anwendung des oben geschilderte Verfahrens.

Wie man ein Viereck auf den Boden zeichnen kann, lässt sich aber auch so formulieren:

Viereckalgorithmus 2:

Wiederhole 4 Mal: Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts.

oder auch wie folgt:

Viereckalgorithmus 3:

Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts. Wiederhole dies, solange Du nicht am Startpunkt stehst.

Wie ähnlich die drei obigen Viereckalgorithmen auch aussehen mögen, sind sie, was die Wiederholung angeht, grundlegend unterschiedlich:

- Der Viereckalgorithmus 1 ist rekursiv, die Viereckalgorithmen 2 und 3 sind es nicht.
- Auch die Viereckalgorithmen 2 und 3 unterscheiden sich wesentlich voneinander: Im Viereckalgorithmus 2 ist die Anzahl der Wiederholungen durch einen expliziten Wert (4) festgelegt; im Viereckalgorithmus 3 hängt die Anzahl der Wiederholungen von einer Bedingung ab, die nach jeder Wiederholung ausgewertet werden muss.

Wiederholungsformen wie in den Viereckalgorithmen 2 und 3 werden „Iterationsschleifen“ genannt. Algorithmen, die Wiederholungen mittels Schleifen statt Rekursion spezifizieren, werden „iterativ“ genannt. Die Technik, auf der iterative Algorithmen beruhen, heißt „Iteration“.
Die Iterationsschleife des Viereckalgorithmus 3 ist eine sogenannte „While-Schleife“. Die Iterationsschleife des Viereckalgorithmus 2 ist eine sogenannte „For-Schleife“. Wir werden später zusätzlich die „Repeat-Until-Schleife“ kennenlernen.

4.3.3 Rekursion versus Iteration: Komplexitätsaspekte

Betrachten wir die Fakultätsfunktion, die über den natürlichen Zahlen total ist. Eine etwas unpräzise Definition der Fakultätsfunktion sieht wie folgt aus:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1) * \ldots * 1 \text{ falls } n \in \mathbb{N} \text{ und } n \geq 1 \end{aligned}$$

Die folgende rekursive Definition derselben Funktion ist präziser, weil sie nicht von der Schreibweise „. . . “ (d.h. usw.) Gebrauch macht:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \text{ falls } n \in \mathbb{N} \text{ und } n \geq 1 \end{aligned}$$

Diese rekursive Definition lässt sich in SML direkt übertragen:

```
fun fak(0) = 1
  | fak(n) = n * fak(n - 1);
```

Gemäß dem Substitutionsmodell (siehe Abschnitt 3.1) besteht die Auswertung von `fak(4)` aus den folgenden Schritten:

```
fak(4)
4 * fak(3)
4 * (3 * fak(2))
4 * (3 * (2 * fak(1)))
4 * (3 * (2 * (1 * fak(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Diese Auswertung bedarf einigen Speichers für unvollendete Zwischenberechnungen wie etwa `4 * (3 * (2 * fak(1)))`. Damit die unterbrochenen Zwischenberechnungen korrekt weitergeführt werden, sobald es möglich ist, ist zudem eine Buchführung notwendig, die Zeit kostet.

Die Berechnung von `4!` kann auch wie folgt erfolgen, was viel weniger Speicher und einer viel einfacheren Buchführung bedarf, weil es keine unterbrochenen Zwischenberechnungen gibt:

```
(*)     4       1
        4 - 1   1 * 4
        3       4
        3 - 1   4 * 3
        2       12
        2 - 1   12 * 2
        1       24
        1 - 1   24 * 1
        0       24
```

Hier wurde nicht nur ein einziger Ausdruck schrittweise umgewandelt, sondern zwei Ausdrücke. Links wurde der jeweilige Wert von `n` dargestellt, rechts die jeweiligen Teilprodukte. Rechts werden sozusagen die Zwischenergebnisse aufgesammelt oder „akkumuliert“. Eine Variable, die zum Aufsammeln von Zwischenergebnissen dient, nennt man üblicherweise einen „Akkumulator“.

Wenn man sowohl für `n` als auch für den Akkumulator Zustandsvariablen verwendet, kann man den Algorithmus für die obige Berechnung wie folgt mit einer Iterationsschleife spezifizieren:

(**) Iterativer Algorithmus zur Berechnung von `fak(!n)` für `!n`$\in \mathbb{N}$: (in einer imperativen Phantasiesprache, die Zustandsvariablen mit expliziter Dereferenzierung hat. Der Dereferenzierungsoperator wird in dieser Sprache wie in SML mit „`!`“ notiert).

```
akk := 1
while !n > 0 do
   akk := !akk * !n;   n := !n - 1
end-while
return !akk
```

Dieser iterative Algorithmus führt genau zu den gewünschten Schritten aus (*), wobei die linke Spalte von (*) den Inhalten von n und die rechte Spalte den Inhalten von akk entspricht.

Jetzt stellt sich natürlich die Frage, ob die Schritte in (*) auch rein funktional erzielt werden können. Sollte sich herausstellen, dass das nicht möglich ist, dann könnte man meinen, die Iteration sei „besser“ als die Rekursion. Wir werden sehen, dass diese Sichtweise etwas naiv ist. Zunächst zeigen wir, wie die Schritte in (*) rein funktional erzielt werden können:

```
fun fak_iter(n) =
   let fun hilf_fak_iter(n, akk) =  if n = 0 then akk
                                    else hilf_fak_iter(n - 1, akk * n)
   in
       hilf_fak_iter(n, 1)
   end
```

Unter Verwendung des Substitutionsmodells können wir uns vergewissern, dass die Auswertung von fak_iter(4) genau die Schritte (*) durchläuft:

```
fak_iter( 4 )
hilf_fak_iter(4,      1      )
hilf_fak_iter(4 - 1,  1 * 4  )
hilf_fak_iter(3,      4      )
hilf_fak_iter(3 - 1,  4 * 3  )
hilf_fak_iter(2,      12     )
hilf_fak_iter(2 - 1,  12 * 2 )
hilf_fak_iter(1,      24     )
hilf_fak_iter(1 - 1,  24 * 1 )
hilf_fak_iter(0,      24     )
24
```

Die rekursive Funktion fak_iter implementiert also den iterativen Algorithmus (**). Das Zusammenkommen beider Wiederholungsformen, Rekursion und Iteration, mag zu Recht etwas verwirrend wirken. Dieser verwirrende Aspekt wird unten unter dem Stichwort „Endrekursion“ behandelt.

Zuvor wollen wir Iteration und Rekursion am Beispiel der Fakultätsfunktion vergleichen:

- Der Unterschied zwischen der Funktion fak_iter und dem iterativen Algorithmus ist nicht sehr wesentlich. Beide Formalismen scheinen ähnlich anschaulich oder ähnlich unanschaulich zu sein.

- Unter Verwendung eines funktionalen Formalismus ist es viel natürlicher, die Fakultät einer natürlichen Zahl mit `fak` statt mit `fak_iter` zu berechnen. Unter Verwendung von Zustandsvariablen und der Iteration ist aber der Algorithmus `(**)` sehr natürlich. An diesem Beispiel mag also die Iteration der Rekursion überlegen erscheinen, weil dieser Formalismus „natürlich“ zu dem effizienteren Algorithmus führt.

- Auch wenn `fak` weniger effizient als `(**)` bzw. `fak_iter` ist, ist das Programm `fak` viel einfacher, daher leichter zu entwickeln und zu warten als `(**)` bzw. `fak_iter`. Zudem kann `fak` sehr leicht unter Anwendung des Substitutionsmodells überprüft werden, was für `(**)` nicht der Fall ist, aber immerhin auch für `fak_iter`. Unter Anwendung des Substitutionsmodells lässt sich die totale Korrektheit auf den natürlichen Zahlen von `fak` sehr leicht beweisen.

So ähnlich können in vielen praktischen Fällen die Vorzüge von Iteration und Rekursion einander gegenüber gestellt werden. Jeder Ansatz hat seine (oft leidenschaftlichen) Anhänger. Der „Streit“ wird zweifelsohne noch lange andauern ...

4.3.4 Endrekursion

Dass die Funktion `fak_iter` (mit ihrer Hilfsfunktion `hilf_fak_iter`) zu denselben Berechnungen wie der iterative Algorithmus `(**)` führt, liegt daran, dass bei der Auswertung von `fak_iter(n)` keine Zwischenberechnung unterbrochen wird. Sonst würde `fak_iter` (mit `hilf_fak_iter`) zu anderen Berechnungen als der iterative Algorithmus `(**)` führen.

Es ist aus der Syntax einer rekursiven Funktion leicht zu erkennen, ob ihre Auswertung die Unterbrechung von Berechnungen verlangt:

- Kommt der rekursive Aufruf R in einem zusammengesetzten Ausdruck A vor, der keine Fallunterscheidung (if-then-else oder case) ist, dann werden Berechnungen unterbrochen, weil die Auswertung von A erst dann möglich ist, wenn der zugehörige Teilausdruck R ausgewertet wurde.

  Beispiel: `n * fak(n - 1)` im Falle der Funktion `fak`.

- Kommt der rekursive Aufruf nicht in einem zusammengesetzten Ausdruck vor, der keine Fallunterscheidung (if-then-else oder case) ist, dann sind keine Unterbrechungen von Berechnungen nötig.

  Beispiel: die Funktion `hilf_fak_iter`.

Eine rekursive Funktion, deren definierender Teil keinen rekursiven Aufruf enthält, der ein echter Teilausdruck eines zusammengesetzten Ausdrucks A ist, so dass A weder ein if-then-else- noch ein case-Ausdruck ist, nennt man „endrekursiv“ (tail recursive).

Der Ausschluss von if-then-else- und case-Ausdrücken in der Definition der Endrekursion spiegelt wider, dass diese Ausdrücke Sonderausdrücke sind, d.h. Ausdrücke, deren Auswertung auf Sonderalgorithmen beruht (siehe Abschnitt 3.1.1 und Abschnitt 3.4). Würden Fallunterscheidungen in der Definition der Endrekursion wie herkömmliche Ausdrücke behandelt, dann würde die Definition nichttriviale rekursive Funktionen, die Basisfälle von Rekursionsfälle unterscheiden, ausschließen.

Endrekursive Funktionen sind rekursiv. Ihre Auswertung führt aber zu iterativen (Berechnungs-) Prozessen.

Sagt man, dass eine Prozedur (bzw. ein Programm) rekursiv (bzw. endrekursiv) ist, so meint man die Syntax der Prozedur (bzw. des Programms), nicht den (Berechnungs-) Prozess, der sich aus dieser Prozedur (bzw. Programm) ergibt.

Die Funktion `hilf_fak_iter` ist also rein syntaktisch rekursiv, aber auch endrekursiv. Damit ist sichergestellt, dass der Berechnungsprozess, der von dieser Funktion ausgelöst wird, ein iterativer Prozess ist.

Vor Jahrzehnten (bis in die 70-er Jahre hinein) gab es naive Implementierungen von funktionalen Programmiersprachen, die bei endrekursiven Funktionen genauso wie bei nicht-endrekursiven Funktionen vorgingen: also sich unterbrachen, Buch über die Unterbrechung führten und nach der Unterbrechung wieder fortsetzten, wo gar nichts mehr fortzusetzen war, aber dafür natürlich trotzdem die Buchführung aktualisieren mussten. Dadurch waren die Berechnungsprozesse zeitaufwändiger als die Berechnungsprozesse, die durch Iterationsschleifen in imperativen Sprachen ausgelöst werden.

Diese Zeiten sind vorbei. Jede moderne Implementierung einer Programmiersprache erzeugt aus einem endrekursiven Programm einen iterativen (Berechnungs-) Prozess. Es ist sogar so, dass der vom Übersetzer aus einer endrekursiven Funktion erzeugte Code exakt aus denselben Maschinenbefehlen besteht wie der entsprechende Code einer Iterationsschleife, so dass nach der Übersetzung gar nicht mehr erkennbar ist, ob der ursprüngliche Algorithmus syntaktisch rekursiv formuliert war oder nicht.

4.3.5 Lineare und quadratische Rekursion, Rekursion der Potenz n

Betrachten wir die sogenannten „Fibonacci-Zahlen“, die wie folgt definiert sind:

Definition Fibonacci-Zahlen:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \text{ für } n \in \mathbb{N} \text{ und } n \geq 2 \end{aligned}$$

Die Funktion `fib` lässt sich wie folgt in SML rekursiv implementieren:

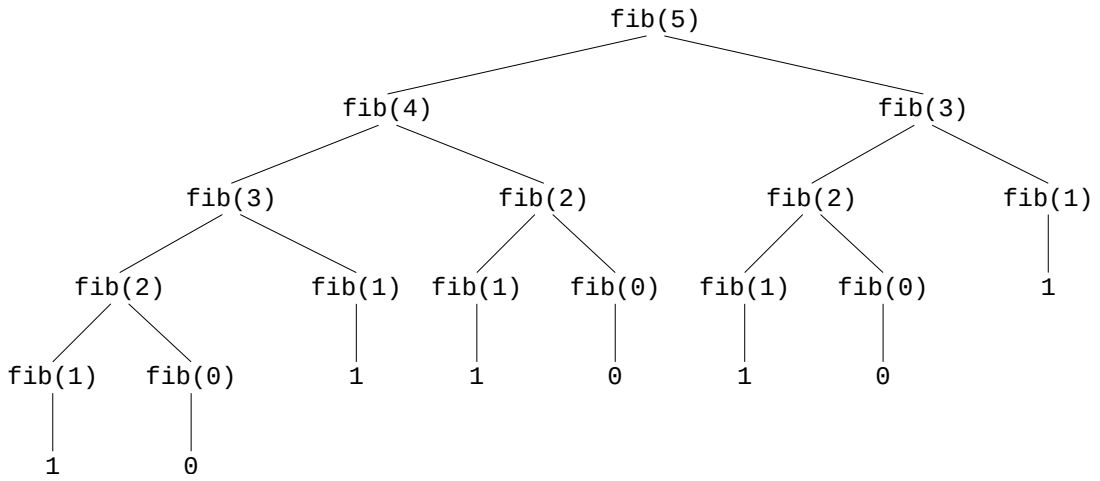
```
fun fib(0) = 0
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2);
```

Bis zu Werten wie etwa `n = 20` erfolgt die Auswertung von `fib(n)` schnell. Für größere Werte wie z.B. `n = 35` oder `n = 40` wird die Rechenzeit für die Auswertung deutlich spürbar. Dies liegt an der besonderer Form der Rekursion in der obigen Definition: im definierenden Teil (oder Rumpf) der Funktion `fib` kommen zwei rekursive Aufrufe vor.

Kommt nur ein rekursiver Aufruf im Rumpf einer Funktion vor (wie z.B. bei `fak` und `hilf_fak_iter`), so heißt die Funktion „linear rekursiv“. Kommen zwei rekursive Aufrufe im Rumpf einer Funktion vor (wie z.B. bei `fib`), so heißt die Funktion „quadratisch rekursiv“. Kommen n rekursive Aufrufe im Rumpf einer Funktion vor, so heißt die Funktion

„rekursiv in der Potenz n“. Achtung: dieses n heißt nur zufällig so wie der Parameter der Funktion `fib`. Man könnte auch einen anderen Namen statt n wählen.

Rekursive Funktionen, die quadratisch rekursiv oder rekursiv der Potenz n sind, heißen auch „baumrekursiv“. Die Anwendung des Substitutionsmodells auf eine Funktion wie `fib` erklärt diese Bezeichnung. Das folgende Bild gibt unter jedem Ausdruck `fib(n)` die zwei rekursiven Aufrufe `fib(n-1)` und `fib(n-2)`, deren Werte addiert werden müssen, um den Wert von `fib(n)` zu erhalten:



Es ist interessant, dass die Fibonacci-Funktion sich asymptotisch wie die Funktion

$$\begin{array}{lcl} \mathbb{N} & \to & \mathbb{N} \\ \mathtt{n} & \mapsto & \phi^{\mathtt{n}} \end{array}$$

verhält. Asymptotisch bedeutet, dass für hinreichend große `n` der Unterschied zwischen `fib(n)` und $\phi^{\mathtt{n}}$ beliebig klein wird. Dabei ist ϕ der „goldene Schnitt“, d.h. die Zahl, die durch die Gleichung $\phi^2 = \phi + 1$ definiert ist, d.h., $\phi = \frac{1+\sqrt{5}}{2} \approx 1,6180$.

Mit Hilfe dieses Zusammenhangs kann man zeigen, dass die Anzahl der Blätter im obigen Baum und damit die benötigte Berechnungszeit exponenziell mit `n` wächst.

An dieser Stelle sei noch auf ein mögliches Missverständnis hingewiesen: Wenn man sagt, dass die Funktion `fib(n)` quadratisch rekursiv ist, heißt das nur, dass im Rumpf der Funktion zwei rekursive Aufrufe vorkommen. Es heißt dagegen nicht, dass die benötigte Berechnungszeit für die Auswertung von `fib(n)` proportional zu $\mathtt{n}^2$ wäre. Wir haben ja gerade gesehen, dass sie tatsächlich proportional zu `exp`(`n`) ist.

4.3.6 Iterative Auswertung der baumrekursiven Funktion fib

Auch bei baumrekursiven Funktionen ist es oft möglich, iterative Auswertungen zu finden. Im Fall von `fib(n)` sind dafür zwei Akkumulatoren `akk1` und `akk2` sowie eine Zwischenvariable `z` nötig:

`(***)` Iterativer Algorithmus zur Berechnung von `fib(!n)` für `!n`$\in \mathbb{N}$ (in einer imperativen Phantasiesprache mit expliziter Dereferenzierung und Dereferenzierungsoperator „`!`“):

```
i := 0
akk1 := 0              (* !akk1: die i-te Fibonaccizahl *)
akk2 := 1              (* !akk2: die (i+1)-te Fibonaccizahl *)
while !i < !n do
   i := !i + 1
   z := !akk2
   akk2 := !akk1 + !akk2
   akk1 := !z
end-while
return !akk1
```

Die Variable `z` dient dazu, den alten Inhalt von `akk2` zu retten, bevor der Inhalt von `akk2` verändert wird, damit der alte Inhalt anschließend zum Inhalt von `akk1` gemacht werden kann. Die Berechnung durchläuft zum Beispiel für `!n = 8` die folgenden Zustände:

`!i`	`!akk1`	`!akk2`
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

Ganz analog zu der Fakultätsfunktion, nur diesmal mit zwei Akkumulatoren, kann dieser iterative Berechnungsprozess auch mit einer endrekursiven Funktion erzielt werden.

4.3.7 Memoisierung

Es ist auffällig, dass eine Funktion wie `fib` wiederholte Auswertungen der gleichen Ausdrücke verlangt: Zur Auswertung von `fib(5)` werden

`fib(4)`	1 mal
`fib(3)`	2 mal
`fib(2)`	3 mal
`fib(1)`	5 mal
`fib(0)`	3 mal

ausgewertet.

Im Folgenden wird noch ein ganz anderer Ansatz dargestellt, der eine effiziente Auswertung der baumrekursiven Funktion `fib` ermöglicht, ohne eine Veränderung ihrer Implementierung zu benötigen.

Der Ansatz besteht in einer Veränderung des Auswertungsalgorithmus. Dabei wird Buch darüber geführt, welche Aufrufe der Funktion `fib` (oder einer anderen Funktion) bereits ausgewertet worden sind:

- Wurde ein rekursiver Aufruf `A` mit Ergebniswert `W` ausgewertet, so wird die Gleichung `A = W` in einer Tabelle T gespeichert ("memoisiert").
- Soll ein rekursiver Aufruf `A` ausgewertet werden, so wird zunächst in der Tabelle T nach einem Eintrag `A = W` gesucht. Gibt es einen solchen Eintrag, so wird der Aufruf `A` nicht wieder ausgewertet, sondern der Wert `W` aus der Tabelle geliefert.

Dieser Ansatz der „Memoisierung" führt im Falle der Funktion `fib` zu keinem iterativen (Berechnungs-) Prozess, er verbessert jedoch die Auswertungszeiten erheblich. Der Preis dafür ist eine Buchführung, die zeitaufwändig ist. Es hängt vom Programm ab, ob diese Buchführung sich lohnt.

Die meisten Implementierungen von Programmiersprachen verwenden keine Memoisierung. Eine Ausnahme bilden die Auswerter der Datenbankanfragesprache SQL. Zur Auswertung von Anfragen über rekursive SQL-Sichten (views) verwenden die modernen SQL-Auswerter die Memoisierung (siehe Hauptstudiumsvorlesungen Datenbanksysteme und Deduktive Datenbanksysteme).

Obwohl die Memoisierung wie die verzögerte Auswertung die Wiederholung der Auswertung mancher Ausdrücke vermeidet, lässt sich die Memoisierung *nicht* auf die verzögerte Auswertung zurückführen. Die verzögerte Auswertung vermeidet lediglich die wiederholten Auswertungen, die sich aus einer Auswertung in normaler Reihenfolge ergeben würden, aber nicht die wiederholten Auswertungen, die sich aus der Definition der Funktion ergeben. Eine verzögerte Auswertung der Fibonacci-Funktion ohne Memoisierung führt zu wiederholten Auswertungen.

4.3.8 Prozedur versus Prozess

Die Beispiele der Endrekursion und der Memoisierung zeigen, dass die Syntax einer Prozedur (oder eines Programms) nicht allein bestimmt, welche Gestalt der zugehörige (Berechnungs-) Prozess hat. Der (Berechnungs-) Prozess hängt sowohl von der Prozedur (oder dem Programm) als auch vom Auswertungsalgorithmus ab.

Die Auswertungsalgorithmen von modernen funktionalen Programmiersprachen erkennen die Endrekursion und erzeugen aus endrekursiven Programmen iterative Prozesse.

4.4 Ressourcenbedarf — Größenordnungen

Prozesse verbrauchen zwei Ressourcen: Rechenzeit und Speicherplatz. Die „Größenordnungen", die im Folgenden eingeführt werden, werden verwendet, um anzugeben, wie viel Rechenzeit oder wie viel Speicherplatz Prozesse verbrauchen.

Zunächst wird für den betrachteten Prozess ein Parameter n festgelegt, der eine natürliche Zahl ist und der die „Größe" des Problems abschätzt. Wie diese Problemgröße abgeschätzt werden kann, hängt vom Problem ab. Sie kann oft in verschiedenen Weisen abgeschätzt werden. Bei der Multiplikation zweier natürlicher Zahlen kann die Problemgröße z.B. die Summe der Längen (d.h. der Anzahl der Ziffern) der beiden natürlichen Zahlen sein, die multipliziert werden sollen. Bei der Berechnung der k-ten Fibonacci-Zahl $fib(k)$ kann die Problemgröße einfach k selbst sein.

Für die benötigte Ressource, Rechenzeit bzw. Speicherplatz, je nach dem, was untersucht wird, wird eine Rechenzeit- bzw. Speicherplatz-Einheit festgelegt.

Der Ressourcenverbrauch eines Prozesses (d.h. der Verbrauch an Rechenzeit oder an Speicherplatz) wird als Funktion $r : \mathbb{N} \to \mathbb{N}$ definiert, die jede Größe n eines Problems auf die Anzahl benötigter Rechenzeit- bzw. Speicherplatz-Einheiten abbildet, die zur Lösung eines Problems der Größe n benötigt werden.

Definition
Definition (Größenordnung)
Seien $f : \mathbb{N} \to \mathbb{N}$ und $s : \mathbb{N} \to \mathbb{N}$ zwei Funktionen (s wie Schranke).
Die Funktion f ist von der Größenordnung $O(s)$, geschrieben f in $O(s)$, wenn es $k \in \mathbb{N}$ und $m \in \mathbb{N}$ gibt, so dass gilt:

Für alle $n \in \mathbb{N}$ mit $n \geq m$ ist $f(n) \leq k * s(n)$.

Die Konstante k in der vorangehenden Definition ist unabhängig von n. In anderen Worten muss es ein und dieselbe Konstante k sein, die für alle $n \in \mathbb{N}$ garantiert, dass $f(n) \leq k * s(n)$ gilt. Wenn eine solche Konstante k nicht existiert, ist f nicht von der Größenordnung $O(s)$. $O(s)$ repräsentiert die Menge aller Funktionen, die bzgl. s die in der Definition angegebene Eigenschaft erfüllen.

Ist f von der Größenordnung $O(s)$ und ist s eine Funktion, die n auf eine Zahl wie z.B. $2n$ bzw. n^4 abbildet, so sagt und schreibt man auch (vereinfachend, aber eigentlich inkorrekt), dass f von der Größenordnung $O(2n)$ bzw. $O(n^4)$ ist. Dabei wird der Ausdruck $2n$ bzw. n^4 als Kurzschreibweise für die folgenden Funktionen verwendet:

$$\begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & 2n \end{array} \quad \text{bzw.} \quad \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n^4 \end{array}$$

Zudem sagt und schreibt man auch $f = O(s)$ statt $f \in O(s)$. Diese andere Schreibweise ist irreführend, weil f nicht gleichzeitig Element von $O(s)$ und gleich zu $O(s)$ sein kann. Sind z.B. die Funktionen $f2$ und $f3$ wie folgt definiert:

$$f2: \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & 2n \end{array} \qquad f3: \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & 3n \end{array}$$

so wird oft geschrieben:

$$f2 = O(n) \text{ und } f3 = O(n)$$

obwohl $f2$ und $f3$ natürlich nicht identisch sind. Die ausführliche und korrektere Schreibweise dafür wäre, zunächst die Funktion

$$id: \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n \end{array}$$

einzuführen und dann zu schreiben $f2 \in O(id)$ und $f3 \in O(id)$. Denn es gibt eine Konstante k, nämlich zum Beispiel $k = 5$, so dass gilt

Für alle $n \in \mathbb{N}$ ist $f2(n) = 2n \leq 5n = k * n = k * id(n)$.

Die gleiche Konstante $k = 5$ ermöglicht den Nachweis, dass $f3$ in $O(id)$ ist. Mit der Konstanten $k = 2$ wäre der Nachweis für $f2$ gelungen, aber nicht für $f3$.

Der Grund, warum die Größenordnungen so definiert sind, wird während der Grundstudiumsvorlesung „Informatik 4“ (Einführung in die Theoretische Informatik) oder in einer Hauptstudiumsvorlesung über die Komplexitätstheorie erläutert werden.

4.5 Beispiel: Der größte gemeinsame Teiler

Der größte gemeinsame Teiler zweier natürlicher Zahlen a und b ist die größte natürliche Zahl, durch die sowohl a als b teilbar (d.h. ohne Rest dividierbar) ist. So ist z.B. $30 = 2*3*5$ der größte gemeinsame Teiler von $60 = (2^2) * 3 * 5$ und $150 = 2 * 3 * (5^2)$.

Die Notation $t|a$ wird im Folgenden verwendet, um auszudrücken, dass t ein Teiler von a ist (das heißt, es gibt ein $k \in \mathbb{N}$ mit $a = t * k$).

Wenn t ein gemeinsamer Teiler von a und von b ist, kann das mit dieser Notation sehr einfach ausgedrückt werden durch $t|a$ und $t|b$.

Der größte gemeinsame Teiler von zwei natürlichen Zahlen a und b kann leicht aus den Zerlegungen in Primzahlen von a und von b ermittelt werden. Er ist der Produkt aller p^n mit:

- die Primzahl p kommt in jeder der beiden Zerlegungen (einmal) vor, einmal mit Exponent n_1, einmal mit Exponent n_2.
- n ist das Minimum von n_1 und n_2.

Die Zerlegung einer natürlicher Zahl in Primzahlen ist eine zeitaufwändige Aufgabe, so dass dieser Ansatz zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen ziemlich ineffizient ist.

Ein effizienterer Ansatz zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen beruht auf der folgenden Eigenschaft:

Satz
Seien $a \in \mathbb{N}$ und $b \in \mathbb{N}$ mit $a \geq b$. Sei r der Rest der Ganzzahldivision von a durch b (d.h. $a = (b * c) + r$ für ein $c \in \mathbb{N}$). Sei $t \in \mathbb{N}$.

$t|a$ und $t|b$ genau dann, wenn $t|b$ und $t|r$.

Beweis:

Seien a, b, c und r wie im Satz definiert.

Notwendige Bedingung (Richtung von links nach rechts; „$\Longrightarrow$“):

Sei angenommen, dass $t|a$ und $t|b$.

Zu zeigen ist, dass $t|b$ und $t|r$ gelten.
Da nach Annahme $t|b$ gilt, reicht es aus, $t|r$ zu zeigen.
Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.
Da $t|a$ gilt, gibt es t_a mit $a = t * t_a$.
Nach Annahme gilt $a = b * c + r$
also $t * t_a = a = b * c + r = t * t_b * c + r$,
also $r = t * t_a - t * t_b * c = t * (t_a - t_b * c)$, d.h. $t|r$.

Hinreichende Bedingung (Richtung von rechts nach links; „$\Longleftarrow$“):

Sei angenommen, dass $t|b$ und $t|r$.

Zu zeigen ist, dass $t|a$ und $t|b$ gelten.
Da nach Annahme $t|b$ gilt, reicht es aus, $t|a$ zu zeigen.
Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.
Da $t|r$ gilt, gibt es t_r mit $r = t * t_r$.
Nach Annahme gilt $a = b * c + r$,
also $a = t * t_b * c + t * t_r = t * (t_b * c + t_r)$, d.h. $t|a$.

qed.

Aus dem Satz folgt, dass der größte gemeinsame Teiler $ggT(a, b)$ zweier natürlicher Zahlen a und b mit $a \geq b$ und $a = b * c + r$ gleich dem größten gemeinsamen Teiler $ggT(b, r)$ von b und r ist.
Diese Beobachtung liefert einen rekursiven Ansatz zur Berechnung des größten gemeinsamen Teilers zweier natürlichen Zahlen a und b:

1. Wenn $a < b$, dann vertausche a und b.
2. Andernfalls (d.h. $a \geq b$) ist der größte gemeinsame Teiler von a und b der größte gemeinsame Teiler von b und r, wobei r der Rest der Ganzzahldivision von a durch b (d.h. in SML **r = a mod b**) ist.

Die aktuellen Parametern (a, b) werden paarweise bei jedem rekursiven Aufruf (außer beim ersten, falls $a < b$) echt kleiner, weil der Rest einer Ganzzahldivision durch b echt kleiner b ist.

Was ist aber der Basisfall der Rekursion?
Man beachte zuerst, dass $ggT(a, 0) = a$ ist, weil $0 = 0 * a = 0 * a + 0$. Der Rest der Ganzzahldivision von 0 durch a ist 0 oder anders ausgedrückt für jede natürliche Zahl a gilt: $a|0$.
Führt aber der oben geschilderte rekursive Ansatz zwangsläufig zu einem rekursiven Aufruf mit aktuellen Parametern der Form $(a, 0)$? Ja, dies lässt sich wie folgt beweisen.

Beweis:

(informell / Beweisidee:)

Bei jedem rekursiven Aufruf ist der zweite aktuelle Parameter (d.h. der Wert des formalen Parameters b) eine natürliche Zahl, die echt kleiner als der zweite aktuelle Parameter des vorherigen rekursiven Aufrufs ist. Zudem ist diese natürliche Zahl größer gleich 0. Da es zwischen dem Wert des formalen Parameters b beim ersten rekursiven Aufruf und 0 nur endlich viele natürlichen Zahlen gibt, muss nach endlich vielen rekursiven Aufrufen der formale Parameter b den Wert 0 haben.

qed.

Diese Bemerkung zusammen mit der Folgerung aus dem Satz liefert die folgende endrekursive Funktion zur Berechnung des größten gemeinsamen Teilers zweier natürlichen Zahlen:

```
fun ggT(a, b) =
     if a < b then ggT(b, a)
              else if b = 0 then a
                            else ggT(b, a mod b);
```

Der Algorithmus, den die Funktion `ggT` implementiert, konvergiert sehr schnell, wie die folgende Anwendung des Substitutionsmodells zeigt:

```
ggT(150, 60)

ggT(60, 30)

ggT(30, 0)

30
```

Dieser Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen wird Euklid (ca. 3. Jhdt. vor Christus) zugeschrieben, weil er in Euklids „Elemente der Mathematik“ erwähnt ist. Er gilt als der älteste bekannte Algorithmus[1], weil er im Gegensatz zu anderen überlieferten Algorithmen aus älteren oder sogar jüngeren Zeiten in Euklids „Elemente der Mathematik“ nicht mittels Beispielen, sondern abstrakt (mit Redewendungen anstelle von Variablen) spezifiziert ist.

Die vorangehende Spezifikation des Euklid'schen Algorithmus ist eine endrekursive Funktion. Die Auswertung dieser Funktion löst also einen iterativen (Berechnungs-) Prozess aus.

Ein Satz des französischen Mathematikers Gabriel Lamé (19. Jhdt.) ermöglicht es, die Rechenzeit des Euklid'schen Algorithmus abzuschätzen:

[1] nach dem Informatiker D.E. Knuth, siehe „The art of computer programming“, volume 2, Seminumerical Algorithms, Addison-Wesley, 1969

Satz (Lamé)

Seien $a \in \mathbb{N}$ und $b \in \mathbb{N}$, so dass $a \geq b$.
Benötigt der Euklid'sche Algorithmus zur Berechnung von $ggT(a, b)$ insgesamt n Rekursionsschritte, so gilt $b \geq fib(n)$, wobei $fib(n)$ die n-te Fibonacci-Zahl ist.

Beweis:

Betrachten wir drei aufeinander folgende rekursive Aufrufe des Algorithmus:

$\texttt{ggT}(\texttt{a}_0, \texttt{b}_0)$
$\texttt{ggT}(\texttt{a}_1, \texttt{b}_1)$
$\texttt{ggT}(\texttt{a}_2, \texttt{b}_2)$

Nach Definition gilt:

$\texttt{a}_1 = \texttt{b}_0$
$\texttt{a}_2 = \texttt{b}_1$
$\texttt{b}_1 = \texttt{a}_0 \texttt{ mod b}$ d.h. $\texttt{a}_0 = \texttt{b}_0 * \texttt{c}_0 + \texttt{b}_1$ (für ein $\texttt{c}_0 \in \mathbb{N}$)
$\texttt{b}_2 = \texttt{a}_1 \texttt{ mod b}_1$ d.h. $\texttt{a}_1 = \texttt{b}_1 * \texttt{c}_1 + \texttt{b}_2$ (für ein $\texttt{c}_1 \in \mathbb{N}$)
Da $\texttt{c}_0 \geq 1$ und $\texttt{c}_1 \geq 1$ ist, folgt:
(*) $\texttt{a}_0 \geq \texttt{b}_0 + \texttt{b}_1$
$\texttt{a}_1 = \texttt{b}_0 \geq \texttt{b}_1 + \texttt{b}_2$

Dieses Ergebnis entspricht gerade dem Bildungsgesetz für die Fibonacci-Zahlen. Es ist also nahe liegend, die Hypothese aufzustellen, dass $\texttt{b}_0 \geq fib(n)$, wenn n die Anzahl der rekursiven Aufrufe der Funktion ggT während der Berechnung von $(\texttt{a}_0, \texttt{b}_0)$ ist (wobei dieser allererste Aufruf nicht mitgezählt wird).

Diese Hypothese kann wie folgt induktiv bewiesen werden.

Wir verschärfen zunächst die Behauptung: Für alle $n \in \mathbb{N}$ gilt: für alle $k \leq n$ und für alle $\texttt{a} \in \mathbb{N}$ und $\texttt{b} \in \mathbb{N}$ mit $\texttt{a} \geq \texttt{b}$, für die die Auswertung von $\texttt{ggT}(\texttt{a}, \texttt{b})$ genau k rekursive Aufrufe benötigt, gilt $\texttt{b} \geq fib(k)$.

Aus dieser verschärften Behauptung folgt die eigentliche Hypothese dann als Spezialfall mit $k = n$.

Basisfall: $n = 0$
Für alle $k \leq 0$ und für alle $\texttt{a} \in \mathbb{N}$ und $\texttt{b} \in \mathbb{N}$ mit $\texttt{a} \geq \texttt{b}$, für die die Auswertung von $\texttt{ggT}(\texttt{a}, \texttt{b})$ genau k rekursive Aufrufe benötigt, gilt $\texttt{b} \geq fib(k)$, weil $k = 0$, $fib(k) = 0$ und $\texttt{b} \in \mathbb{N}$ gilt.

Basisfall: $n = 1$
Für $k = 0$ gilt, wie gerade gezeigt, für alle $\texttt{a} \in \mathbb{N}$ und $\texttt{b} \in \mathbb{N}$ mit $\texttt{a} \geq \texttt{b}$, für die die Auswertung von $\texttt{ggT}(\texttt{a}, \texttt{b})$ genau k rekursive Aufrufe benötigt, dass $\texttt{b} \geq fib(k)$. Für $k = 1$ seien $\texttt{a} \in \mathbb{N}$ und $\texttt{b} \in \mathbb{N}$ mit $\texttt{a} \geq \texttt{b}$, so dass die Auswertung von $\texttt{ggT}(\texttt{a}, \texttt{b})$ genau k rekursive Aufrufe benötigt, also genau einen. Das bedeutet, dass a und b die Bedingungen erfüllen müssen, die in der Definition von $\texttt{ggT}(\texttt{a}, \texttt{b})$ zum letzten else-Fall führen, also insbesondere $\texttt{b} \neq 0$. Also gilt $\texttt{b} \geq 1 = fib(1) = fib(k)$. Also ist die Behauptung für alle $k \leq 1$ gezeigt.

Induktionsfall:
Sei $n \geq 2$ und sei angenommen (Induktionsannahme): für alle $k \leq n$ und für alle

$\mathtt{a} \in \mathbb{N}$ und $\mathtt{b} \in \mathbb{N}$ mit $\mathtt{a} \geq \mathtt{b}$, für die die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau k rekursive Aufrufe benötigt, gilt $\mathtt{b} \geq fib(k)$.

Sei nun $k \leq n+1$ und seien $\mathtt{a} \in \mathbb{N}$ und $\mathtt{b} \in \mathbb{N}$ mit $\mathtt{a} \geq \mathtt{b}$, so dass die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau k rekursive Aufrufe benötigt. Falls $k \leq n$ ist, gilt $\mathtt{b} \geq fib(k)$ nach Induktionsannahme. Es bleibt also nur noch der Fall $k = n + 1$ zu zeigen.

Die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ benötige also $n + 1$ rekursive Aufrufe. Sei $\mathtt{a}_0 = \mathtt{a}$ und $\mathtt{b}_0 = \mathtt{b}$ und seien $\mathtt{ggT}(\mathtt{a}_1, \mathtt{b}_1)$ und $\mathtt{ggT}(\mathtt{a}_2, \mathtt{b}_2)$ die zwei ersten rekursiven Aufrufe. Nach Konstruktion benötigt die Auswertung von $\mathtt{ggT}(\mathtt{a}_1, \mathtt{b}_1)$ genau n rekursive Aufrufe und die Auswertung von $\mathtt{ggT}(\mathtt{a}_2, \mathtt{b}_2)$ genau $n - 1$ rekursive Aufrufe. Es gilt sowohl $n \leq n$ als auch $n - 1 \leq n$, so dass nach Induktionsannahme gilt $\mathtt{b}_1 \geq fib(n)$ und $\mathtt{b}_2 \geq fib(n - 1)$.

Nach (*) gilt: $\mathtt{b}_0 \geq \mathtt{b}_1 + \mathtt{b}_2$.

Also $\mathtt{b} = \mathtt{b}_0 \geq fib(n) + fib(n - 1) = fib(n + 1) = fib(k)$.

Hinweis: für die Anwendbarkeit von (*) wurde die Voraussetzung gebraucht, dass $n \geq 2$ ist. Deshalb musste der Basisfall $n = 1$ getrennt behandelt werden.

qed.

Benötigt der Euklid'sche Algorithmus zur Berechnung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau n rekursive Aufrufe, so gilt nach dem Satz von Lamé:

$$\mathtt{b} \geq fib(n) \approx \phi^n$$

wobei ϕ der „goldene Schnitt“ ist (siehe Abschnitt 4.3.5).
Daraus folgt, dass asymptotisch, also für große n, gilt (mit der Schreibweise $\log_\phi$ für den Logarithmus zur Basis ϕ):

$$\log_\phi \mathtt{b} \geq n$$

Da es aber eine Konstante k gibt mit $\log_\phi \mathtt{b} \leq k * \ln \mathtt{b}$, wobei ln den Logarithmus zur Basis e bezeichnet, gilt asymptotisch auch $n \leq k * \ln \mathtt{b}$, also

$$\mathtt{ggT}(\mathtt{a}, \mathtt{b}) \in O(\ln(\mathtt{b})).$$

Man sagt, dass die Anzahl der Rekursionsschritte, die der Euklid'sche Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen benötigt, höchstens logarithmisch in diesen Zahlen (oder präziser in der kleineren der beiden Zahlen) ist.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 5

Die vordefinierten Typen von SML

In jeder Programmiersprache sind einige „Basisdatentypen“ vordefiniert, üblicherweise die Datentypen „ganze Zahl“, „reelle Zahl“, „Boole'sche Werte“, „Zeichen“ und „Zeichenfolge“. In diesem Kapitel werden zunächst die vordefinierten Basistypen von SML behandelt, die aus früheren Kapiteln und Übungsaufgaben schon bekannt sind. Ziel ist es, einen umfassenden Überblick über die Basistypen von SML zu geben. Ferner wird die Zusammensetzung von Typen in SML eingeführt.

5.1 Was sind Typen?

Ein Typ (oder Datentyp) ist eine Menge von Werten. Mit einem Typ werden Operationen, oder Prozeduren, zur Bearbeitung der Daten des Typs angeboten.

Eine n-stellige Operation über einem Typ T ist eine Funktion $T^n \rightarrow T$, wobei T^n für $T \times \ldots \times T$ (n Mal) steht. Die ganzzahlige Addition ist z.B. eine binäre (d.h. zweistellige) Operation über dem Typ „ganze Zahl“.

Ein Typ kann vordefiniert sein, d.h. von der Programmiersprache als Wertemenge angeboten werden. Mit einem vordefinierten Typ bieten Programmiersprachen die Operationen, Funktionen oder Prozeduren an, die zur Bearbeitung von Daten des Typs üblich sind.

Moderne Programmiersprachen ermöglichen auch, dass der Programmierer selbst Typen definiert wie z.B. einen Typ „Wochentag“ mit Wertemenge

$$\{\text{Montag}, \text{Dienstag}, \ldots, \text{Sonntag}\}$$

einen Typ „Uhrzeit“ mit einer Wertemenge

$$\{h : m : s \mid h \in \mathbb{N}, 0 \leq h < 24, m \in \mathbb{N}, 0 \leq m < 60, s \in \mathbb{N}, 0 \leq s < 60\}$$

einen Typ „komplexe Zahl“ mit Wertemenge

$$\{a + ib \mid a \text{ und } b \text{ vom Typ „reelle Zahlen“ }\}$$

oder einen Typ „Übungsgruppe“ zur Bündelung der folgenden Merkmale einer Übungsgruppe:

- Der Name des Übungsleiters (als Zeichenfolge dargestellt)
- Die Namen der in dieser Übungsgruppe angemeldeten Studenten (als Zeichenfolgen dargestellt)

- Der wöchentliche Termin der Übung (als Paar aus Wochentag und Uhrzeit dargestellt)
- Der Ort, wo die Übungsgruppe sich trifft (als Zeichenfolge dargestellt)

Offenbar verlangt die Spezifikation von Typen wie „komplexe Zahlen“ und „Übungsgruppen“ Mittel zur Zusammensetzung von Typen wie „ganze Zahl“ (zur Bildung des Typs „Uhrzeit“), „reelle Zahl“ (zur Bildung des Typs „komplexe Zahl“), „Zeichenfolge“ und „Uhrzeit“ (zur Bildung des Typs „Übungsgruppe“).
Die Bildung von neuen Typen wird in den Kapiteln 8 und 11 behandelt. In diesem Kapitel werden die vordefinierten Typen von SML behandelt. Die vordefinierten Typen von SML bilden eine sehr „klassische“ Auswahl an vordefinierten Typen, die sich von dem Angebot an vordefinierten Typen in anderen Programmiersprachen nur unwesentlich unterscheidet.

5.2 Die Basistypen von SML

5.2.1 Ganze Zahlen

Der SML-Typ `int` (integer) steht für die „ganzen Zahlen“.
Das Minusvorzeichen der negativen ganzen Zahlen wird in SML `~` geschrieben: z.B. `~89`.
Führende Nullen sind in SML erlaubt: z.B. `007`, `089`, `~002`.
Über dem Typ `int` bietet SML die folgenden binären Operationen an:

`+`	(infix)	Addition
`-`	(infix)	Subtraktion
`*`	(infix)	Multiplikation
`div`	(infix)	ganzzahlige Division
`mod`	(infix)	Rest der ganzzahligen Division

SML bietet über dem Typ `int` die folgenden Vergleichsoperatoren an:

`=`	(infix)	gleich
`<>`	(infix)	nicht gleich
`<`	(infix)	echt kleiner
`<=`	(infix)	kleiner-gleich
`>`	(infix)	echt größer
`>=`	(infix)	größer-gleich

Die Vergleichsoperatoren des Typs `int` sind Funktionen vom Typ `int` $\times$ `int` $\rightarrow$ `bool`.
Die Funktion `real` vom Typ `int` $\rightarrow$ `real` dient zur Konvertierung einer ganzen Zahl in eine Gleitkommazahl mit demselben mathematischen Wert.

5.2.2 Reelle Zahlen

Der SML-Typ `real` bezeichnet die Gleitkommazahlen, die auch inkorrekterweise „reelle Zahlen“ genannt werden. Was Gleitkommazahlen genau sind, wird in der Grundstudiumsvorlesung Informatik 3 erläutert. Im Grunde stellen Gleitkommazahlen rationale Zahlen

dar, aber nur eine endliche Teilmenge davon und mit Gesetzmäßigkeiten der Arithmetik, die stark von den mathematischen Gesetzmäßigkeiten abweichen können.

Zur Darstellung von Gleitkommazahlen in SML können zwei Konstrukte (zusammen oder nicht zusammen) verwendet werden:

- Der Punkt zur Darstellung von Dezimalbruchzahlen: z.B. `31.234`, `123.0`, `012.0`, `~2.459`
- Die Mantisse-Exponent-Notation oder E-Notation zur Darstellung von Zehnerpotenzen: z.B. `123E5`, `123.0E~3`, `123.0e~3`, `~0.899e4`

In der Mantisse-Exponent-Notation kann das „`e`“ sowohl groß als auch klein geschrieben werden.

SML lässt führende Nullen in Dezimalbruchzahlen sowie in Mantissen und Zehnerexponenten zu. Vor dem Punkt einer Dezimalbruchzahl verlangt SML eine Ziffer: z.B. lässt SML die Schreibweise `.89` nicht zu.

Über dem Typ `real` bietet SML die folgenden binären Operationen an:

`+`	(infix)	Addition
`-`	(infix)	Subtraktion
`*`	(infix)	Multiplikation
`/`	(infix)	Division

Achtung: die Arithmetik mit Gleitkommazahlen folgt ihren eigenen Gesetzmäßigkeiten und führt oft zu anderen Ergebnissen als die Arithmetik mit rationalen Zahlen oder gar reellen Zahlen im mathematischen Sinn:

```
- 1234567890.0 + 0.005;
val it = 1234567890.01 : real

- 1234567890.0 + 0.0005;
val it = 1234567890.0 : real
```

Aus diesen Gründen ergeben arithmetische Berechnungen mit Gleitkommazahlen im Allgemeinen bestenfalls Approximationen der tatsächlichen Werte. Oft sind sorgfältige Analysen mit Methoden der numerischen Mathematik erforderlich, um sicherzustellen, dass die Ergebnisse überhaupt in der Nähe der tatsächlichen Werte liegen und nicht einfach völlig falsch sind.

SML bietet über dem Typ `real` die folgenden Vergleichsoperatoren an:

`<`	(infix)	echt kleiner
`<=`	(infix)	kleiner-gleich
`>`	(infix)	echt größer
`>=`	(infix)	größer-gleich

Die Funktion `floor` vom Typ `real` → `int` konvertiert eine Gleitkommazahl in eine ganze Zahl und rundet sie dabei nach unten. Die Funktionen `ceil` vom gleichen Typ rundet nach oben. Die Funktion `trunc` vom gleichen Typ rundet in Richtung Null, indem sie einfach alle Nachkommastellen weglässt.

Der SML-Typ `real` enthält außer den „normalen“ Gleitkommazahlen noch einige spezielle Werte, die als Ergebnis bestimmter Operationen auftreten können:

```
- 1.0 / 0.0;
val it = inf : real

- 0.0 / 0.0;
val it = nan : real

- Math.sqrt(~1.0);
val it = nan : real

- 1.0 + (1.0 / 0.0);
val it = inf : real

- 1.0 + (0.0 / 0.0);
val it = nan : real
```

Dabei steht `inf` für *infinite*, also unendlich, und `nan` für *not-a-number*, also kein Zahlenwert. Wie die letzten beiden Beispiele andeuten, sind alle Operationen des Typs `real` auch definiert, wenn diese speziellen Werte als Argument auftreten. Die Einzelheiten dieser Definitionen folgen einem internationalen Standard für Gleitkommazahlen in Programmiersprachen (IEEE standard 754-1985 und ANSI/IEEE standard 854-1987).

5.2.3 Boole'sche Werte

Der SML-Typ `bool` (Boole'sche Werte) besteht aus der Wertemenge {`true`, `false`}.
Über dem Typ `bool` bietet SML die folgenden Operatoren an:

`not`	(präfix, unär)	Negation
`andalso`	(infix, binär)	Konjunktion
`orelse`	(infix, binär)	Disjunktion

Die Operatoren `andalso` und `orelse` sind in SML keine Funktionen:

- Während der Auswertung von `A1 andalso A2` wird zunächst nur `A1` ausgewertet. Hat `A1` den Wert `false`, so wird `A2` nicht ausgewertet (und `false` als Wert von `A1 andalso A2` geliefert).
- Während der Auswertung von `A1 orelse A2` wird zunächst nur `A1` ausgewertet. Hat `A1` den Wert `true`, so wird `A2` nicht ausgewertet (und `true` als Wert von `A1 andalso A2` geliefert).

5.2.4 Zeichenfolgen

Der SML-Typ `string` ist die Menge der endlichen Zeichenfolgen.
In SML werden Zeichenfolgen eingeklammert zwischen zwei " geschrieben. `""` bezeichnet in SML die leere Zeichenfolge.
Das Zeichen " wird in SML innerhalb einer Zeichenfolge `\"` geschrieben: z.B `"ab\"cd"` bezeichnet in SML die Zeichenfolge `ab"cd`. Weitere „escape sequences“, die mit dem Zeichen \ anfangen, dienen zur Darstellung von Sonderzeichen in SML:

\n : newline
\t : tab
\\ : \

Die Folge:

\ gefolgt von *white-space*–Zeichen gefolgt von \

ermöglicht es, sogenannte *white-space*–Zeichen wie newline, tab oder Leerzeichen, die zur lesbareren Darstellung eines Programms nützlich sind, innerhalb einer SML-Zeichenfolge zu ignorieren: z.B.

```
- "aaaa\   \b";
val it = "aaaab" : string

- "ccc\
=
=
= \d";
val it = "cccd" : string
```

Über dem Typ string bietet SML die folgenden Operationen an:

`size`	: (präfix, unär)	Länge einer Zeichenfolge
`^`	: (infix, binär)	Konkatenation (Aneinanderfügen) zweier Zeichenfolgen

SML bietet über dem Typ `string` die folgenden Vergleichsoperatoren an:

`=`	(infix)	gleich
`<>`	(infix)	nicht gleich
`<`	(infix)	echt kleiner
`<=`	(infix)	kleiner-gleich
`>`	(infix)	echt größer
`>=`	(infix)	größer-gleich

Diese Operatoren beziehen sich auf die sogenannte lexikographische Ordnung, nach der zum Beispiel `"a" < "aa" < "b"` ist.

5.2.5 Zeichen

Der SML-Typ `char` besteht aus der Menge der Zeichen. Man beachte den Unterschied zwischen Zeichen und Zeichenfolgen der Länge 1: Wie die einelementige Menge {2} nicht dasselbe wie die ganze Zahl 2 ist, so ist das Zeichen `b` nicht dasselbe wie die Zeichenfolge `b`.

Im SML wird ein Zeichen `z` als `#"z"` geschrieben, das ist `#` gefolgt von der Zeichenfolge `z`, also von `"z"`.

Über dem Typ `char` bietet SML die folgenden Funktionen an:

`chr` : `int` → `char`	für $0 \leq$ `n` ≤ 255 liefert `chr(n)` das Zeichen mit Code `n`;
`ord` : `char` → `int`	liefert den Code eines Zeichens `z`

Zum Beispiel:

```
- chr(100);
val it = #"d" : char

- ord(#"d");
val it = 100 : int

-  chr(ord(#"d"));
val it = #"d" : char

- ord(chr(89));
val it = 89 : int
```

Als Basis der Kodierung von Zeichen, d.h. der Zuordnung numerischer Codes zu Zeichen des Datentyps `char`, dient der ASCII-Code (American Standard Code for Information Interchange; siehe Tabelle 5.1)
Zur Konvertierung zwischen Zeichen und Zeichenfolgen bietet SML bzw. die Standardbibliothek von SML die unäre Funktion `str` und die binäre Funktion `String.sub` an:

```
- str(#"a");
val it = "a" : string

-  String.sub("abcd", 0);
val it = #"a" : char

 - String.sub("abcd", 2);
 val it = #"c" : char
```

5.2.6 unit

Der SML-Typ `unit` besteht aus der Wertemenge {`()`}. `()` wird oft *unity* ausgesprochen. Dieser einzige Wert des Typs `unit` wird als Wert von Prozeduraufrufen verwendet.

5.3 Zusammengesetzte Typen in SML

5.3.1 Vektoren (Tupel)

Sind $n \geq 0$ und $t_1, t_2, \ldots, t_n$ SML-Typen und $A_1, A_2, \ldots A_n$ Ausdrücke der Typen $t_1, t_2, \ldots, t_n$, so ist $(A_1, A_2, \ldots, A_n)$ ein n-stelliger Vektor (oder n-Tupel) vom Typ $t_1 * t_2 * \ldots * t_n$ (dabei bezeichnet „*“ in SML das kartesische Produkt von Typen).
Zum Beispiel:

```
- ("abc", 44, 89e~2);
val it = ("abc",44,0.89) : string * int * real

- (("abc", 44), (44, 89e~2));
val it = (("abc",44),(44,0.89)) : (string * int) * (int * real)
```

Tabelle 5.1: ASCII-Code / ASCII-Zeichensatz

Zeichen	Code		Zeichen	Code		Zeichen	Code		Zeichen	Code
NUL	0		␣	32		@	64		‘	96
SOH	1		!	33		A	65		a	97
STX	2		"	34		B	66		b	98
ETX	3		#	35		C	67		c	99
EOT	4		$	36		D	68		d	100
ENQ	5		%	37		E	69		e	101
ACK	6		&	38		F	70		f	102
BEL	7		’	39		G	71		g	103
BS	8		(	40		H	72		h	104
HT	9		)	41		I	73		i	105
LF	10		*	42		J	74		j	106
VT	11		+	43		K	75		k	107
FF	12		,	44		L	76		l	108
CR	13		-	45		M	77		m	109
SO	14		.	46		N	78		n	110
SI	15		/	47		O	79		o	111
DLE	16		0	48		P	80		p	112
DC1	17		1	49		Q	81		q	113
DC2	18		2	50		R	82		r	114
DC3	19		3	51		S	83		s	115
DC4	20		4	52		T	84		t	116
NAK	21		5	53		U	85		u	117
SYN	22		6	54		V	86		v	118
ETB	23		7	55		W	87		w	119
CAN	24		8	56		X	88		x	120
EM	25		9	57		Y	89		y	121
SUB	26		:	58		Z	90		z	122
ESC	27		;	59		[	91		{	123
FS	28		<	60		\	92		|	124
GS	29		=	61		]	93		}	125
RS	30		>	62		^	94		~	126
US	31		?	63		_	95		DEL	127

Erläuterung der Steuerzeichen:

NUL = Null, SOH = Start of heading, STX = Start of text, ETX = end of text,
EOT = end of transmission, ENQ = enquiry, ACK = acknowledge, BEL = bell,
BS = backspace, HT = horizontal tab, LF = line feed, VT = vertical tab,
FF = form feed, CR = carriage return, SO = shift out, SI = shift in,
DLE = data link escape, DC1 = device control 1, DC2 = device control 2,
DC3 = device control 3, DC4 = device control 4, NAK = negative acknowledge,
SYN = synchronous idle, ETB = end of transmission block, CAN = cancel,
EM = end of medium, SUB = substitute, ESC = escape, FS = file separator,
GS = group separator, RS = record separator, US = unit separator, DEL = delete.

Man beachte, dass Komponenten von Vektoren selbst Vektoren sein dürfen.
Die Gleichheit über Vektoren (derselben Länge!) ist komponentenweise definiert:

```
- val eins = 1;
val eins = 1 : int

- val drei = 3;
val drei = 3 : int

- (eins, drei) = (1, 3);
val it = true : bool
```

Vektoren der Längen 1 und 0 stellen Ausnahmen dar:

- Ein einstelliger Vektor ist in SML identisch mit seiner (einzigen) Komponente:

  ```
  - (3) = 3;
  val it = true : bool
  ```

- Der 0-stellige Vektor `()` ist der (einzige) Wert des SML-Typs `unit`.

In SML hängen Vektoren und Argumente von „mehrstelligen“ Funktionen wie folgt zusammen: In einer Funktionsanwendung $\mathtt{f}(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ stellt $(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ einen Vektor dar, so dass die Funktion `f` einstellig ist (und auf dreistellige Vektoren angewandt wird); zum Beispiel:

```
- fun f(n:int, m:int) = n + m;
val f = fn : int * int -> int

- val paar = (1, 2);
val paar = (1,2) : int * int

- f paar;
val it = 3 : int
```

In SML sind also jede Funktion und der Wert eines jeden Ausdrucks einstellig.
Zur Selektion der Komponenten eines Vektors kann in SML Pattern Matching verwendet werden:

```
- val tripel = (1, #"z", "abc");
val tripel = (1,#"z","abc") : int * char * string

 - val (komponente1, komponente2, komponente3) = tripel;
 val komponente1 = 1 : int
 val komponente2 = #"z" : char
 val komponente3 = "abc" : string
```

Zur Selektion der Komponenten eines Vektors können in SML auch die Funktionen `#1`, `#2`, usw. verwendet werden:

```
- #1("a", "b", "c");
val it = "a" : string

- #3("a", "b", "c");
val it = "c" : string
```

5.3.2 Deklaration eines Vektortyps

Einem Vektortyp $\texttt{t}_1 * \texttt{t}_2 * \ldots * \texttt{t}_n$ kann wie folgt ein Name gegeben werden:

```
- type punkt = real * real;

- fun abstand(p1: punkt, p2: punkt) =
      let fun quadrat(z) = z * z
          val delta_x = #1(p2) - #1(p1)
          val delta_y = #2(p2) - #2(p1)
      in
          Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
      end;
val abstand = fn : punkt * punkt -> real

- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Man beachte, dass `punkt` lediglich ein Synonym für `real * real` ist. In der Tat ist `(real * real) * (real * real)` der Typ des aktuellen Parameters der vorangehenden Funktionsanwendung (wie gesagt ist die Funktion einstellig, und ihr Argument ist somit ein Paar von Paaren von Gleitkommazahlen).

Wegen der Typ-Constraints `p1: punkt` und `p2: punkt` verlangt die Definition der lokalen Funktion `quadrat` keine Typ-Constraints, um die Überladung der Multiplikation aufzulösen.

5.3.3 Verbunde (Records)

Ein n-stelliger Vektor ist eine geordnete Zusammensetzung von n Komponenten, so dass die Komponenten durch ihre Position bestimmt werden. Man kann also einen dreistelligen Vektor vom Typ `string * char * int` wie z.B. `("Bry", #"F", 2210)` als eine Zusammensetzung von einer ganzen Zahl, einer Zeichenfolge und einem Zeichen beschreiben, so dass gilt:

- die ganze Zahl hat die Position 3,
- die Zeichenfolge hat die Position 1,
- das Zeichen hat die Position 2.

Es bietet sich an, anstelle von nummerierten Positionen Bezeichner zu verwenden wie etwa:

- Nachname

- Vornamenbuchstabe
- Durchwahl

Diese Idee liegt den Verbunden (oder records) zu Grunde. Das vorangehende Beispiel kann wie folgt unter Verwendung eines SML-Verbunds dargestellt werden:

```
- val adressbucheintrag = {Nachname          = "Bry",
                           Vornamenbuchstabe = #"F",
                           Durchwahl         = "2210"};
val adressbucheintrag =
  {Durchwahl="2210",Nachname="Bry",Vornamenbuchstabe=#"F"}
  : {Durchwahl:string, Nachname:string, Vornamenbuchstabe:char}
```

Man beachte, dass die Reihenfolge der Komponenten eines Verbunds keine Rolle spielt. Dies folgt logisch daraus, dass die Komponenten eines Verbundes mit Bezeichnern identifiziert werden anstatt mit Positionen wie bei Vektoren.

Man beachte auch, dass die Bezeichner der Komponenten eines Verbundes im Typ des Verbundes vorkommen. Es ist eben logisch, dass die Verbunde {`a = 1, b = 2`} und {`aaa = 1, bbb = 2`} nicht denselben Typ haben:

```
- {a = 1, b = 2};
val it = {a=1,b=2} : {a:int, b:int}

- {aaa = 1, bbb = 2};
val it = {aaa=1,bbb=2} : {aaa:int, bbb:int}
```

Verbunde werden komponentenweise verglichen:

```
-  {a=1, b=2} = {b=2, a=1};
val it = true : bool
```

Zur Selektion der Komponentenwerte mit Hilfe der Komponentenbezeichner eines Verbundes bietet SML die Funktion #*Bezeichner* an:

```
- #bbb({aaa=1,bbb=2});
val it = 2 : int

- #a({a = 1, b = 2});
val it = 1 : int
```

SML bietet auch die folgende Kurzschreibweise für Deklarationen an:

```
- val info1dozent = {Nachname = "Bry", Vorname = "Francois"};
val info1dozent = {Nachname="Bry",Vorname="Francois"}
   : {Nachname:string, Vorname:string}

- val {Nachname, Vorname} = info1dozent;
val Nachname = "Bry" : string
val Vorname = "Francois" : string
```

So werden Variablen deklariert, die dieselben Namen wie die Komponentenbezeichner haben. Das Folgende ist aber nicht möglich:

```
- val {nn, vn} = info1dozent;
 stdIn:1.1-39.11 Error: pattern and expression in val dec don't
 agree [tycon mismatch]
   pattern:    {nn:'Z, vn:'Y}
   expression:    {Nachname:string, Vorname:string}
   in declaration:
     {nn=nn,vn=vn} =
       (case info1dozent
         of {nn=nn,vn=vn} => (nn,vn))
```

Verbunde sind den „structures“ der Programmiersprache C und den „records“ der Programmiersprachen Pascal und Modula ähnlich.

5.3.4 Deklaration eines Vektor- oder Verbundstyps

Wie für Vektoren bietet SML die Möglichkeit an, einem Verbundtyp einen Namen zu geben, wie z.B.:

```
- type complex = real * real;

- type dateieintrag = {Vorname:string, Nachname:string};
```

Der so vergebene Name ist lediglich ein Synonym für den Verbundtyp.

5.3.5 Vektoren als Verbunde

In SML sind Vektoren Verbunde mit besonderen Komponentenbezeichnern, wie die folgende Sitzung zeigt:

```
- {1="abc", 2="def"};
val it = ("abc","def") : string * string

- fun vertauschen {1 = x:string, 2 = y:string} =  {1=y, 2=x};
val vertauschen = fn : string * string -> string * string

- val paar = ("abc", "def");
val paar = ("abc","def") : string * string

- vertauschen paar;
val it = ("def","abc") : string * string
```

5.4 Listen

Der Begriff „Liste“ kommt in den meisten Programmiersprachen und in vielen Algorithmen — mit einigen unwesentlichen Unterschieden vor allem in der Syntax — vor. Wir wollen zunächst den Begriff „Liste“ unabhängig von jeglicher Programmiersprache erläutern. Danach werden wir den SML-Typ „Liste“ einführen.

5.4.1 Der Begriff „Liste“ in Algorithmenspezifikations- und Programmiersprachen

Eine Liste ist eine endliche geordnete Folge von Elementen. Listen werden oft wie folgt dargestellt: `[1, 2, 3]` oder `["a", "bcd", "e", "fg"]`. Die leere Liste ist möglich, sie wird als `[]` dargestellt.

Der Typ Liste verfügt über eine Funktion, `cons` für „list constructor“ genannt, um Listen wie folgt aufzubauen:

> Angewandt auf einen Wert `W` und eine Liste `L` bildet `cons` die Liste, deren erstes (d.h. am weitesten links stehendes) Element `W` ist und deren weitere Elemente die Elemente der Liste `L` (in derselben Reihenfolge) sind.

Angewandt auf `5` und die Liste `[9, 8]` bildet also `cons` die Liste `[5, 9, 8]`. In anderen Worten ist `[5, 9, 8]` der Wert von `cons(5, [9, 8])`. Die Funktion `cons` wird oft infix geschrieben. Man schreibt also `5 cons [9, 8]` statt `cons(5, [9, 8])`.

Aus der Definition von `cons` folgt, dass eine Liste wie `[5, 9, 8]` auch

```
5 cons (9 cons (8 cons []))
```

notiert werden kann. Ist zudem die Infixfunktion `cons` rechtsassoziativ, was in vielen Programmiersprachen der Fall ist, so kann eine Liste wie `[5, 9, 8]` auch wie folgt notiert werden:

```
5 cons 9 cons 8 cons []
```

Die meisten Programmiersprachen bieten eine Notation wie `[5, 9, 8]` als Ersatz (syntaktischer Zucker) für den weniger lesbaren Ausdruck `5 cons 9 cons 8 cons []`.

Der Typ Liste verfügt zudem über zwei Funktionen, mit dem auf Werte aus einer Liste zugegriffen werden kann: `head` und `tail`:

- Angewandt auf eine nicht leere Liste `L` liefert `head` das erste (d.h. das am weitesten links stehende) Element von `L`.
- Angewandt auf eine nicht leere Liste `L` liefert `tail` die Liste, die sich aus `L` ergibt, wenn das erste Element von `L` gestrichen wird.

Angewandt auf `[5, 9, 8]` liefern also `head` den Wert `5` und `tail` den Wert `[9, 8]`.

Die Funktionen `head` und `tail` sind auf Listen nicht total, weil sie für die leere Liste nicht definiert sind.

Weil sie ermöglichen, Listen zu zerlegen (to decompose), werden die Funktionen `head` und `tail` oft „decomposers“ genannt. In Anlehnung an die Bezeichnung „constructor“ werden `head` und `tail` auch gelegentlich „destructors“ genannt.

Wie kann man unter Verwendung von `head` und `tail` eine Funktion spezifizieren, die das letzte Element einer nichtleeren Liste liefert? Selbstverständlich unter Anwendung der Rekursion:

> Das letzte Element `E` einer nichtleeren Liste `L` ist definiert als: Falls `L` die einelementige Liste `[A]` ist, so ist `A` das letzte Element. Andernfalls ist das letzte Element von `L` das letzte Element der Liste `tail(L)`.

Der Test, ob eine Liste nur ein Element enthält, lässt sich ebenfalls einfach wie folgt spezifizieren:

Eine Liste `L` enthält (genau) ein Element genau dann, wenn `tail(L) = []` ist.

Listenelemente können allgemein die Werte von atomaren oder zusammengesetzten Ausdrücken sein. So ist z.B. der Wert des Ausdrucks `[eins, zwei]` die Liste `[1, 2]`, wenn `1` der Wert von `eins` und `2` der Wert von `zwei` ist.

Selbstverständlich sind Listen von Listen wie etwa `[[1,2],[1,5]]` möglich.

Die Gleichheit für Listen ist elementweise definiert: `[a,b,c] = [d,e,f]` genau dann, wenn `a=d`, `b=e` und `c=f` gilt.

5.4.2 Die Listen in SML

Eine SML-Liste ist eine endliche Folge von Werten desselben Typs. In SML ist es also nicht möglich, Listen von Werten aus verschiedenen Typen zu bilden.

Für jeden gegebenen Typ `'a` (`'a` wird oft *alpha* ausgesprochen) bezeichnet in SML `'a list` den Typ der Listen von Werten vom Typ `'a`. Ist z.B. `'a` der Typ `int`, so ist `int list` der Typ, der aus den Listen von ganzen Zahlen besteht.

SML bietet zwei Notationen für Listen:

1. mit dem Listkonstruktor `cons`, der in SML `::` geschrieben wird und rechtsassoziativ ist, und der leeren Liste, die in SML `nil` geschrieben wird.

   In dieser Notation kann die Liste der ersten vier natürlichen Zahlen als `0 :: (1 :: (2 :: (3 :: nil)))`, d.h. dank der Rechtsassoziativität von `::` auch als `0 :: 1 :: 2 :: 3 :: nil` geschrieben werden.

   ```
   - 0 :: 1 :: 2 :: 3 :: nil;
   val it = [0,1,2,3] : int list

   - 0 :: (1 :: (2 :: (3 :: nil)));
   val it = [0,1,2,3] : int list
   ```

2. unter Verwendung der Listenklammern „`[`“ und „`]`“ mit „`,`“ als Trennzeichen zwischen den Listenelementen. In dieser Notation werden z.B. die Liste der ersten vier natürlichen Zahlen als `[0,1,2,3]` und die leere Liste als `[]` dargestellt:

   ```
   - [0,1,2,3];
   val it = [0,1,2,3] : int list
   ```

Selbstverständlich dürfen beide Notationen zusammen verwendet werden:

```
- 0 :: 1 :: [2, 3];
val it = [0,1,2,3] : int list
```

Die Notation mit den Listenklammern „`[`“ und „`]`“ und dem „`,`“ als Trennzeichen ist lediglich „syntaktischer Zucker“, d.h. eine Kurzform für die Notation mit dem Listenkonstruktor `::` (cons).

Der SML-Typ der leeren Liste `nil` (oder `[]`) ist `'a list`, was „Liste von Elementen eines beliebigen Typs“ heißt:

```
- nil;
val it = [] : 'a list

- [];
val it = [] : 'a list
```

Dabei ist `'a` (*alpha*) eine Typvariable, d.h. eine Variable, die als Wert einen Typ erhalten kann. Man sagt, dass `nil` ein „polymorphes Objekt“ ist, d.h. ein „Objekt“, das mehreren Typen angehört. Dass `nil` ein polymorphes Objekt ist, hat den Vorteil, dass es nur eine leere Liste gibt. Wäre `nil` kein polymorphes Objekt, dann müsste für jeden möglichen Typ `'a` eine leere Liste für den Typ `'a list` geben, was ziemlich umständlich wäre.
SML bietet die Gleichheit für Listen:

```
- val eins = 1;
val eins = 1 : int

- val zwei = 2;
val zwei = 2 : int

- [eins, 2] = [1,zwei];
val it = true : bool
```

5.4.3 Mono- und Polytypen

Ein Typausdruck wie `'a` oder `'a list` wird „polymorpher Typ“ oder „Polytyp“ genannt, weil der Ausdruck für mehrere (griechisch „poly“) Typen steht: Mögliche Instanzen von `'a` sind z.B. `int`, `bool` oder `int list`; mögliche Instanzen von `'a list` sind z.B. `int list`, `bool list`, `(int list) list`, oder `(int * bool) list`.
Ein Typ, der kein Polytyp ist, wird „Monotyp“ genannt.

5.5 Beispiele: Grundlegende Listenfunktionen

5.5.1 Länge einer Liste

```
- fun laenge(nil)    = 0
    | laenge(_ :: L) = 1 + laenge(L);
val laenge = fn : 'a list -> int

 - laenge([0,1,2,3]);
val it = 4 : int
```

5.5.2 Letztes Element einer nichtleeren Liste

```
- fun letztes_element(x :: nil) = x
    | letztes_element(_ :: L)   = letztes_element(L);
Warning: match nonexhaustive
          x :: nil => ...
```

```
          _ :: L   => ...
val letztes_element = fn : 'a list -> 'a

- letztes_element([0,1,2,3]);
  val it = 3 : int
```

Das SML-System erkennt, dass die Deklaration der Funktion `letztes_element` keinen Fall für die leere Liste hat, d.h. dass diese Funktion über einem Typ `'a list` nicht total ist. Das SML-System gibt eine Warnung, weil nichttotale Funktionen manchmal fehlerhaft sind. Da aber die Deklaration einer nichttotalen Funktion in manchen Fällen — wie hier — notwendig ist, wird eine solche Deklaration nicht abgelehnt.

5.5.3 Kleinstes Element einer nichtleeren Liste von ganzen Zahlen

```
- fun kleinstes_element(x :: nil) = x : int
    | kleinstes_element(x :: L)   = let val y = kleinstes_element(L)
                                    in
                                        if x <= y then x else y
                                    end;
Warning: match nonexhaustive
          x :: nil => ...
          x :: L   => ...
val kleinstes_element = fn : int list -> int
```

Das Typ-Constraint `x : int` ist notwendig, weil der Boole'sche Operator `<=` überladen ist.

5.5.4 n-tes Element einer Liste

```
- fun ntes_element(1, x :: _) = x
    | ntes_element(n, _ :: L) = ntes_element(n-1, L);
```

Über welcher Menge ist diese Funktion total?

5.5.5 `head`

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a
```

SML bietet die vordefinierte Funktion `hd` an:

```
- hd([1,2,3]);
val it = 1 : int
```

5.5.6 tail

```
- fun tail(_ :: L) = L;
Warning: match nonexhaustive
          _ :: L => ...
val tail = fn : 'a list -> 'a list
```

SML bietet die vordefinierte Funktion `tl` an:

```
- tl([1,2,3]);
val it = [2,3] : int list
```

5.5.7 append

Die vordefinierte SML-Funktion `append`, infix „`@`“ notiert, dient dazu, zwei Listen aneinander zu fügen:

```
- [1,2,3] @ [4,5];
val it = [1,2,3,4,5] : int list
```

Die Funktion `append` kann wie folgt in SML implementiert werden:

```
- fun append(nil,    L) = L
    | append(h :: t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list

- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

Berechnungsschritte zur Auswertung von `append([1,2,3], [4,5])`:

```
append(1:: (2:: (3:: nil)),  4:: (5:: nil))
1 :: append(2:: (3:: nil),   4:: (5:: nil))
1 :: (2 :: append(3:: nil,   4:: (5:: nil)))
1 :: (2 :: (3 :: append(nil, 4:: (5:: nil))))
1 :: (2 :: (3 :: (4:: (5:: nil))))
```

Es gibt keinen weiteren Berechnungsschritt mehr; `1 ::(2 ::(3 ::(4 ::(5 ::nil))))` ist die Liste `[1, 2, 3, 4, 5]`.

Zeitbedarf von `append`:

Es ist naheliegend, als Zeiteinheit die Anzahl der Aufrufe der Funktion `cons` (`::`) oder aber die Anzahl der rekursiven Aufrufe von `append` zu wählen. Beide Zahlen stehen einfach miteinander in Verbindung: Wird zur Berechnung von `append(L, L')` die `append`–Funktion $n+1$ mal rekursiv aufgerufen, so wird die Funktion `cons` (`::`) n mal aufgerufen.

(*) Um eine Liste `L` der Länge n mit $n \geq 1$ vor einer Liste `L'` einzufügen, ruft die Funktion `append` die Funktion `cons` (`::`) genau n Male auf.

Bemerkenswert ist, dass die Länge des zweiten Parameters `L'` den Zeitbedarf von `append` nicht beeinflusst.

Ist n die Länge des ersten Parameters von `append`, so gilt: `append` $\in O(n)$.

5.5.8 naive-reverse

Mit der vordefinierten SML-Funktion „reverse“, `rev` notiert, kann aus einer Liste eine Liste in umgekehrter Reihenfolge erzeugt werden:

```
- rev([1,2,3]);
val it = [3,2,1] : int list
```

Eine Funktion „reverse“ kann in SML wie folgt implementiert werden:

```
- fun naive_reverse(nil)    = nil
    | naive_reverse(h :: t) = append(naive_reverse(t), h :: nil);

- naive_reverse([1,2,3]);
val it = [3,2,1] : int list
```

Berechnungsschritte zur Auswertung von `naive_reverse([1,2,3])`:

```
naive_reverse(1::(2::(3::nil)))
append(naive_reverse(2::(3::nil)), 1::nil)
append(append(naive_reverse(3::nil), 2::nil), 1::nil)
append(append(append(naive_reverse(nil),3::nil),2::nil),1::nil)
append(append(append(nil,               3::nil),2::nil),1::nil)
append(append(3::nil,                          2::nil),1::nil)
append(3::append(nil, 2::nil),                        1::nil)
append(3::(2::nil),                                   1::nil)
3::append(2::nil,                                     1::nil)
3::(2::append(nil,                                    1::nil))
3::(2::(1::nil))
```

Zeitbedarf von `naive_reverse`:

Zur Schätzung der Größe des Problems „Umkehrung einer Liste“ bietet es sich an, die Länge der Liste zu wählen.

Wie zur Schätzung des Zeitbedarfs der Funktion `append` bietet es sich an, als Zeiteinheit die Anzahl der rekursiven Aufrufe von `naive_reverse` oder die Anzahl der Aufrufe der Funktion `cons` (`::`) zu wählen.

Gegeben sei eine Liste `L` der Länge n mit $n \geq 1$. Während des Aufrufes von `naive_reverse(L)` wird die Funktion `naive_reverse` $n + 1$ mal rekursiv aufgerufen, zunächst mit einer Liste der Länge $n - 1$ als Parameter, dann mit einer Liste um ein Element kürzer als Parameter bei jedem weiteren Aufruf. Wegen (*) (siehe die Schätzung des Zeitbedarfs von `append`) wird zur Zerlegung der Eingabeliste die Funktion `cons` (`::`) also

$$(n-1) + (n-2) + \ldots + 1$$

Male aufgerufen. Zum Aufbau der auszugebenden Liste wird `cons` (`::`) zudem n mal aufgerufen. Die Gesamtanzahl der Aufrufe von `cons` (`::`) lautet also:

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n * (n+1)}{2}$$

Ist n die Länge des Parameters von `naive_reverse`, so gilt:

$$\texttt{naive_reverse} \in O(\frac{n * (n+1)}{2})$$

also

$$\texttt{naive_reverse} \in O(n^2)$$

5.5.9 reverse

Der quadratische Zeitbedarf von `naive_reverse` ist nicht zufriedenstellend, weil, wie das Folgende zeigt, eine Liste in einer Zeit in umgekehrte Reihenfolge gebracht werden kann, die linear von der Listenlänge abhängt:

Man fängt mit zwei Listen an: die linke Liste, die die Eingabeliste ist, und die rechte Liste, die anfangs leer ist. Nach und nach wird das erste Element der linken Liste von dieser Liste entfernt und am Anfang der rechten Liste eingefügt. Dabei werden nur Operationen verwendet, die der Typ Liste anbietet. Nach soviel Schritte, wie die Eingabeliste lang ist, ist die linke Liste leer und die rechte Liste die Liste in umgekehrten Reihenfolge, die aufzubauen war:

Schritt	Linke Liste	Rechte Liste
0	`[1, 2, 3]`	`[]`
1	`[2, 3]`	`[1]`
2	`[3]`	`[2, 1]`
3	`[]`	`[3, 2, 1]`

Dieses Verfahren lässt sich einfach in SML wie folgt implementieren:

```
- fun aux_reverse(nil,  R) = R
    | aux_reverse(h::t, R) = aux_reverse(t, h::R);
 val aux_reverse = fn : 'a list * 'a list -> 'a list

- aux_reverse([1,2,3], []);
val it = [3,2,1] : int list
- aux_reverse([1,2], [8,9]);
val it = [2,1,8,9] : int list
```

Die gewünschte einstellige `reverse`–Funktion folgt unmittelbar aus der Spezifikation von `aux_reverse`:

```
- fun reverse(L) = let fun aux_reverse(nil,  R) = R
                         | aux_reverse(h::t, R) = aux_reverse(t, h::R)
```

```
                    in
                        aux_reverse(L, nil)
                    end;
val reverse = fn : 'a list -> 'a list

- reverse([1,2,3]);
val it = [3,2,1] : int list
```

Berechnungsschritte zur Auswertung von reverse([1,2,3]):

```
reverse(1::(2::(3::nil)))
aux_reverse(1::(2::(3::nil)),  nil)
aux_reverse(2::(3::nil),    1::nil)
aux_reverse(3::nil,     2::(1::nil))
aux_reverse(nil,    3::(2::(1::nil)))
3::(2::(1::nil)))
```

Zeitbedarf von `reverse`:

Ist $n \geq 0$ die Länge einer Liste `L`, so bedarf die Auswertung von `aux_reverse(L, nil)` n rekursiver Aufrufe sowie n Aufrufe der Funktion `cons` (`::`). Es gilt also:

`aux_reverse` $\in O(n)$

Folglich gilt auch:

`reverse` $\in O(n)$

Der zweite Parameter der Funktion `aux_reverse`, der der rechten Liste aus unserem Beispiel entspricht, ist ein sogenannter Akkumulator. Die Nutzung eines Akkumulators wurde bereits im Abschnitt 4.3 erläutert, um die Berechnung der Fakultät einer natürlichen Zahl in einem iterativen Prozess zu berechnen.

Wie die Funktion `fak_iter` aus Abschnitt 4.3 ist die Funktion `reverse` endrekursiv, so dass sie einen iterativen Berechnungsprozess auslöst.

5.6 Hinweis auf die Standardbibliothek von SML

Die Standardbibliothek von SML, die unter der URI

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

zugreifbar ist, bietet für die Basistypen von SML Funktionen an, die herkömmliche Operationen über diesen Typen in SML implementieren.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 6

Typprüfung

Moderne Programmiersprachen ermöglichen die Festlegung von Typen für Variablen oder Programmausdrücke, die dann automatisch überprüft werden. Dieses Kapitel führt in diese Technik, „Typprüfung“ genannt, ein. Zunächst wird zwischen statischer und dynamischer Typprüfung unterschieden. Dann wird der Begriff Polymorphie erläutert. Die Sprachkonstrukte von SML zur Spezifikation von Typen und Typ-Constraints werden ferner eingeführt. Schlussregeln für die Typinferenz für vereinfachte SML-Programme werden angegeben, und der Unifikationsalgorithmus wird eingeführt. Letztlich wird das Prinzip eines Verfahrens zur automatischen Typinferenz erläutert.

6.1 Die Typprüfung: Eine nützliche Abstraktion für die Entwicklung von korrekten Programmen

Die Typprüfung (*type checking*) umfasst zwei komplementäre Aufgaben:

1. Die Ermittlung der Typen von Ausdrücken eines Programms aus den Typ-Constraints, die in diesem Programm vorkommen, z.B. die Ermittlung der Typen der Namen `x` (`int`) und `zweimal` (`int -> int`) im folgenden Programm:

   ```
   - fun zweimal(x) = 2 * x;
   ```

2. Die Überprüfung der Typen, die die Typ-Constraints eines Programms angeben, z.B. die Überprüfung der Korrektheit der angegebene Typen im folgenden Programm (die Typ-Constraints des Parameters `x` und der Typ der Konstante `2.0` (`real`) sind nicht kompatibel, so dass das Programm einen Typfehler aufweist):

   ```
   - fun zweimal'(x:int):real = 2.0 * x;
   ```

Typen stellen eine Abstraktion dar, die zur Entwicklung von Algorithmen und Programmen äußerst nützlich ist. Sie ermöglichen zu überprüfen, bevor ein Programm aufgeführt wird oder während einer Ausführung eines Programms, dass die Parameter, die an Prozeduren „weitergegeben“ werden, nicht verwechselt worden sind.

6.2 Statische versus dynamische Typprüfung

Eine Typprüfung kann zur Laufzeit durchgeführt werden, d.h. wenn das Programm ausgeführt wird und Ausdrücke ausgewertet werden. Man spricht dann von einer „dynamischen Typprüfung“. Die Programmiersprachen Lisp, Pascal und Smalltalk führen z.B. eine dynamische Typprüfung durch. Die meisten stark (oder streng) typisierten Programmiersprachen führen eine dynamische Typprüfung durch.

Eine Typprüfung kann aber auch zur Übersetzungszeit durchgeführt werden, d.h. wenn das Programm auf syntaktische Korrektheit überprüft wird und wenn es in ein Programm in Maschinensprache umgewandelt wird, also bevor das Programm ausgeführt wird. Man spricht dann von einer „statischen Typprüfung“. Die Programmiersprachen SML, Miranda, C++ und auch Java führen eine statische Typprüfung durch. Eine Besonderheit von SML liegt darin, dass diese Programmiersprache ausschließlich eine statische Typprüfung durchführt. Die statische Typprüfung wurde für SML konzipiert und mit SML eingeführt.[1]

Eine Programmiersprache, die eine dynamische Typprüfung durchführt, heißt „dynamisch typisiert“. Eine Programmiersprache, die eine statische Typprüfung und keine dynamische Typprüfung durchführt, heißt „statisch typisiert“. Eine Programmiersprache, die überhaupt keine Typprüfung durchführt, heißt „nicht typisiert“.

Es ist zu betonen, dass manche moderne Programmiersprachen wie C nicht typisiert sind bzw. nur eine partielle Typprüfung erlauben. Solche Programmiersprachen führen häufig zu Programmierfehlern — in C oft, wenn Zeiger verwendet werden.

Im Gegensatz zu anderen Programmiersprachen wie C++ und Java führt SML überhaupt keine dynamische Typprüfung durch. Keine dynamische Typprüfung, sondern eine ausschließlich statische Typprüfung durchzuführen hat die folgenden Vorteile:

1. Zum einen trägt die statische Typprüfung zur frühen Erkennung von Programmier- oder Konzeptionsfehler schon während der Programmentwicklung bei.

2. Zum zweiten entlastet eine ausschließlich statische Typprüfung die Laufzeit von einer zeitaufwendigen Aufgabe und trägt zur Einfachheit des Übersetzers (bzw. des Auswerters) bei.

3. Zum dritten können bei statisch typisierten Programmiersprachen keine typbedingten „Laufzeitfehler“ vorkommen, d.h. Fehler im Programm in der Maschinensprache, in das das ursprüngliche Programm übersetzt wird. (Selbstverständlich gilt dies nur insofern, dass der Übersetzer selbst fehlerfrei ist.)

Die Programmierpraxis hat gezeigt, dass die statische Typprüfung eine sehr große Hilfe zur Entwicklung von fehlerfreien Programmen ist. Es ist zu erwarten, dass die statische Typprüfung sich unter den zukünftigen industriellen Programmiersprachen verbreiten wird.

Welche Programmiersprache und welches Berechnungsmodell — wie etwa das funktionale, das imperative oder das logische Modell — gerade betrachtet werden, die Typprüfung beruht immer auf denselben Techniken. Es ist günstig, diese Techniken in Zusammenhang mit SML zu lernen, weil die Typprüfung dieser Programmiersprache besonders ausgereift ist.

[1] R. Milner. A theory of type polymorphism in programming languages, Journal of Computer and System Science, vol. 17, pp. 348-375, 1978.

SML ist eine sogenannte „polymorphe“ Programmiersprache. Im nächsten Abschnitt wird dieser Begriff eingeführt. Die Polymorphie hat Auswirkungen auf die Typprüfung: Sie macht fortgeschrittene Techniken zur Typprüfung, wie die Unifikation, notwendig.

6.3 Die Polymorphie: Eine wünschenswerte Abstraktion

6.3.1 Polymorphe Funktionen, Konstanten und Typen

Der Algorithmus zum Aneinanderhängen zweier Listen ist derselbe, ganz egal ob es sich um Listen von ganzen Zahlen, um Listen von Zeichen, um Listen von Boole'schen Werten oder um Listen von Objekten eines weiteren Typs handelt.

Dieser Algorithmus kann in SML wie folgt durch die Funktion `append` implementiert werden (siehe Abschnitt 5.5.7):

```
- fun append(nil, l) = l
    | append(h :: t, l) = h :: append(t, l);
val append = fn : 'a list * 'a list -> 'a list
```

Die SML-Funktion `append` ist tatsächlich auf Listen von Objekten beliebigen Typs anwendbar:

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list

- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it = [#"a",#"b",#"c",#"d",#"e"] : char list

- append([10e~1,20e~1],[30e~1]);
val it = [1.0,2.0,3.0] : real list

- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]] : int list list
```

Eine Funktion oder Prozedur, die wie die oben definierte SML-Funktion `append` auf aktuellen Parametern unterschiedlicher Typen anwendbar ist, heißt „polymorph“. Diese Bezeichnung bezieht sich auf die Parameter, die unterschiedlicher (griechisch „poly“) Gestalt (griechisch „morph“) sein können. Den Typ einer polymorphen Funktion nennt man einen „polymorphen Typ“ oder kurz ein „Polytyp“. Beispiele von polymorphen Typen sind in der SML-Syntax:

`'a`	irgendein Typ
`'a list`	Liste von Objekten eines beliebigen Typs (aber allen von demselben Typ)

Polymorph können nicht nur Funktionen und Typen sein, sondern auch beliebige Ausdrücke. In SML ist z.B. die leere Liste `nil` (auch `[]` notiert) eine polymorphe Konstante.

Eine Programmiersprache, die wie SML polymorphe Funktionen und Prozeduren ermöglicht, wird „polymorph“ genannt. Man sagt auch, dass sie die „Polymorphie“ anbietet.

6.3.2 Typen von Vorkommen eines polymorphen Ausdrucks

Ist ein Ausdruck polymorph, so dürfen unterschiedlichen Vorkommen dieses Ausdrucks im selben Programm unterschiedliche Typen erhalten. Wenn z.B. die polymorphe Funktion `append` auf Listen von ganzen Zahlen angewendet wird, so erhält die polymorphe Konstante `nil` im Rumpf von `append` den Typ `int list`. Wird aber nun dieselbe polymorphe Funktion `append` auf Listen von Zeichen angewendet, so erhält die polymorphe Konstante `nil` im Rumpf von `append` den Typ `char list`.

Ein weiteres Beispiel einer polymorphen Funktion ist die Identitätsfunktion:

```
- val id = fn x => x;
val id = fn : 'a -> 'a
```

Im Ausdruck `id(id)(2)` erhält das innere Vorkommen von `id` den Typ `int -> int`, das äußere den Typ `('a -> 'a) -> ('a -> 'a)`:

```
- id(id)(2);
val it = 2 : int
```

6.3.3 Vorteile der Polymorphie

Nicht alle Programmiersprachen sind polymorph. Viele Programmiersprachen wie z.B. Pascal und Basic verlangen, dass für jede Art von Listen eine `append`-Funktion speziell für diese Art von Listen programmiert wird. Nichtpolymorphe Programmiersprachen haben die folgenden Nachteile:

1. Die Nichtpolymorphie vergrößert die Programme unnötig und trägt damit dazu bei, sie unübersichtlich zu machen.
2. Die Nichtpolymorphie erschwert die Wartung von Programmen, weil derselbe Algorithmus in verschiedenen Prozeduren verbessert werden muss.

Die Polymorphie stellt also eine Abstraktion dar, die in Programmiersprachen äußerst wünschenswert ist.

Die Polymorphie ist eine Verbesserung von Programmiersprachen, die erst in den 80er Jahren vorgeschlagen wurde. Dabei war SML sowohl Vorreiterin als auch Untersuchungsfeld.[2]

6.4 Polymorphie versus Überladung

Wie die Funktion `append` können einige vordefinierte Funktionen von SML wie z.B. die Addition auf aktuelle Parameter von unterschiedlichen Typen angewandt werden:

[2]Der folgende, immer noch aktuelle Artikel bespricht das Thema und beschreibt verschiedene Ansätze zur Polymorphie:

L. Cardelli and P. Wegner: On understanding types, data abstraction, and polymorphism, ACM Computing Surveys, 17, pp. 471–522, 1985

```
- 2 + 5;
val it = 7 : int

- 2.1 + 4.2;
val it = 6.3 : real
```

Diese Eigenschaft der Addition wurde im Abschnitt 2.2) als „Überladung“ genannt. Die Frage stellt sich also, ob Überladung und Polymorphie unterschieden werden sollten.

Addition

Bei der Addition handelt es sich um die Verwendung desselben Namens (Bezeichner) zum Aufruf von verschiedenen (System-)Prozeduren: der Prozedur zur Addition von ganzen Zahlen einerseits, der Prozedur zur Addition von Gleitkommazahlen andererseits.

Funktion `append`

Bei der Funktion `append` handelt es sich um die Verwendung derselben Prozedur, folglich auch desselben Namens, mit Aufrufparametern mit unterschiedlichen Typen.
Polymorphie und Überladung sind völlig unterschiedlich:

- Die Polymorphie bezeichnet die Verwendung derselben Prozedur mit Aufrufparametern mit unterschiedlichen Typen.
- Die Überladung bezeichnet die Verwendung desselben Namens zur Bezeichnung von verschiedenen Prozeduren.

Die Überladung war schon in FORTRAN, einer der ältesten Programmiersprachen, vorhanden. Die Polymorphie ist viel später erschienen, erst nachdem stark typisierte Programmiersprachen entwickelt wurden. Untypisierte Programmiersprachen benötigen keine Polymorphie. Die automatische Typanpassung von schwach typisierten Programmiersprachen ermöglicht in den vielen praktischen Fällen die Polymorphie.

ad hoc und parametrische Polymorphie

Die Überladung wird auch „*ad hoc*–Polymorphie“ genannt. Wird die Überladung so bezeichnet, so spricht man von „parametrischer Polymorphie“ für das, was hier „Polymorphie“ genannt wird.[3]

6.5 Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML

6.5.1 Typvariablen

Der Typ der polymorphen Funktion `append` aus Abschnitt 6.1 lautet:

[3] Die Bezeichnungen „*ad hoc*–Polymorphie“ und „parametrische Polymorphie“ gehen laut Peter Hancock auf C. Strachey zurück — siehe

Peter Hancock. Polymorphic type-checking, pp. 139-162 in: Simon L. Peyton Jones. The Implementation of functional programming languages, Prentice Hall, ISBN 0-13-453333-X ISBN 0-13-453325 Paperback, 1987

```
'a list * 'a list -> 'a list
```

wie das SML-System nach der Auswertung der Deklaration von `append` mitteilt:

```
- fun append(nil, l) = l
    | append(h :: t, l) = h :: append(t, l);
val append = fn : 'a list * 'a list -> 'a list
```

Dabei ist `'a` (oft *alpha* gesprochen) eine „Typvariable“. Die Polymorphie der Funktion `append` macht es nötig, dass eine Variable im Typ dieser Funktion vorkommt. Wird die Funktion `append` auf aktuelle Parameter eines Typs `t` angewandt, so wird die Typvariable `'a` an `t` gebunden. Daraus lässt sich der aktuelle Typ der polymorphen Funktion `append` in der gegebene Funktionsanwendung bestimmen. Eine solche Ermittlung eines Typs wird „Typinferenz“ genannt.

Typvariablen werden auch „schematische Typvariablen“ und „generische Typvariablen“ (letzterer Begriff wird z.B. für die Programmiersprache Miranda verwendet) genannt.

6.5.2 Typinferenz

Die Typinferenz ist die Schlussfolgerung, durch die der Typ eines Ausdrucks ermittelt wird oder die Erfüllung der Typ-Constraints eines Programms überprüft wird. Betrachten wir die folgende Funktionsdeklaration:

```
- fun zweimal(x) = 2 * x;
```

Der Typ der Funktion `zweimal` kann wie folgt ermittelt werden. Da `2` eine ganze Zahl ist, steht der überladene Name `*` für die Multiplikation zweier ganzen Zahlen. Folglich hat `x` den Typ `int`. Daraus folgt der Typ `int -> int` der Funktion `zweimal`.

Fall 1:

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list
```

Die aktuellen Parameter der Funktionsanwendung haben den Typ `int list`. Aus dem polymorphen Typ `'a list * 'a list -> 'a list` von `append` und dem Typ `int list` der aktuellen Parameter der Funktionsanwendung folgt der aktuelle Typ von `append` in der Funktionsanwendung:

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

Fall 2:

```
- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it = [#"a",#"b",#"c",#"d",#"e"] : char list
```

In diesem Fall wird `'a` an `char` gebunden, so dass aus dem polymorphen Typ `'a list * 'a list -> 'a list` der aktuelle Typ

```
char list * char list -> char list
```

von `append` in der Funktionsanwendung folgt.

Fall 3:

```
- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]] : int list list
```

Dieser Fall ist nur scheinbar komplizierter. Hier wird die Typvariable `'a` an den Typ `int list` gebunden, so dass der aktueller Typ von `append` in der Funktionsanwendung

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

ist. Wir erinnern daran, dass der Postfix-Operator `list` linksassoziativ ist und dass der Operator `*` (Kartesisches Produkt) stärker bindet als der Operator `->`, so dass der obige Typausdruck für den folgenden steht:

```
((int list) list * (int list) list) -> ((int list) list)
```

6.5.3 Typausdrücke

In den vorangegangenen Beispielen kommen „Typausdrücke“ vor. In diesem Abschnitt wird der Formalismus näher erläutert, mit dem in einem SML-Programm die Typen von Ausdrücken festgelegt werden können.

Typvariablen sind herkömmlichen Variablen ähnlich. Sie werden an Typausdrücke gebunden.

Ein Typausdruck ist entweder *atomar*, wie z.B. `int` und `char`, oder *zusammengesetzt*, wie z.B. `int list` oder `(int list) list`.

6.5.4 Typkonstanten

Atomare Typausdrücke, die keine Typvariablen sind, werden „Typkonstanten“ genannt. Beispiele von Typkonstanten sind: `int`, `real`, `bool`, `string`, `char` und `unit`. Typkonstanten bezeichnen Typen, die nicht zusammengesetzt sind. Im Kapitel 8 wird gezeigt, wie solche Typen definiert werden können.

6.5.5 Typ-Constraints

Ein Typ-Constraint (siehe Abschnitt 2.4) ist ein Ausdruck der Gestalt:

```
Ausdruck : Typausdruck
```

wie z.B.:

```
x : int
l : char list
(fn x => x * x) : int -> int
```

Typ-Constraints werden auch Typisierungsausdrücke genannt.

6.5.6 Zusammengesetzte Typausdrücke und Typkonstruktoren

Zusammengesetzte Typausdrücke werden ähnlich wie Funktionsanwendungen, oft mit Postfix- oder Infix-Operatoren, gebildet, z.B.:

```
'a * 'a
int list
int -> int
int list * int list -> int list
{ Vorname:string, Nachname:string }
```

Die Operatoren, die dabei verwendet werden, werden „Typkonstruktoren“ genannt.

Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionsnamen. Typkonstruktoren werden aber anders als Funktionsnamen verwendet:

- Wird eine Funktion wie `append` auf die Listen `[1,2]` und `[3]` angewandt, so geschieht dies, damit die Funktionsanwendung `append([1,2],[3])` ausgewertet wird, d.h. damit der Wert `[1,2,3]` berechnet wird.
- Wird der Typkonstruktor `*` auf die Typausdrücke `int` und `'a list` angewandt, so geschieht dies lediglich, damit der Typausdruck `(int * 'a list)` gebildet wird.

Bei der Anwendung eines Typkonstruktors auf „Parameter“ findet keine Auswertung statt. Eine solche Auswertung könnte in der Regel nicht berechnet werden. Die Anwendung des Typkonstruktors `*` (Kartesisches Produkt) auf die beiden Typkonstanten `int` und `bool` bezeichnet z.B. die Menge aller Paare (n, w), so dass n eine ganze Zahl ist und w ein Boole'scher Wert ist. Mathematisch gesehen bildet die Anwendung des Typkonstruktors `*` (Kartesisches Produkt) die Typen `int` und `bool`, d.h. die Mengen $\mathbb{Z}$ und $\{true, false\}$, auf die folgende Menge ab:

$$\{\ (n, w) \mid n \in \mathbb{Z} \text{ und } w \in \{true, false\}\ \}$$

Diese Menge kann nicht berechnet werden, weil sie unendlich ist. In einem Fall wie etwa `bool * bool` wäre die Berechnung des zusammengesetzten Typs möglich, weil er endlich ist. Eine solche Berechnung eines (endlichen) Typs ist aber sinnlos. Typen werden dazu verwendet, Programmierfehler zu vermeiden, aber nicht um alle möglichen Werte zu berechnen.

6.5.7 Die ''-Typvariablen zur Polymorphie für Typen mit Gleichheit

Die Gleichheitsfunktion `=` ist überladen, weil dasselbe Symbol `=` für viele verschiedene Typen wie etwa `bool`, `int`, die polymorphen Typen List, Vektor und Verbund verwendet werden kann.

Viele Algorithmen, für die eine Implementierung als polymorphe Funktion nahe liegt, beziehen sich auf die Gleichheit.

Damit die Spezifikation von solchen Funktionen in SML möglich ist, bietet SML die ''-Typvariablen. Eine ''-Typvariable wird `''Name` geschrieben. ''-Typvariablen stehen immer für Typen mit Gleichheit.

Ein Beispiel der Verwendung von ''-Typvariablen ist das polymorphe Prädikat `member` zum Testen, ob ein Element in einer Liste vorkommt (wir erinnern daran, dass ein Prädikat eine Funktion ist, deren Anwendung Boole'sche Werte liefert):

```
- fun member(x, nil)         = false
    | member(x, head::tail) = if x = head
                              then true
                              else member(x, tail);
val member = fn : ''a * ''a list -> bool

- member(3,[1,2,3,4]);
val it = true : bool

- member(#"c",[#"a",#"b",#"c",#"d"]);
val it = true : bool

- member([1,2],[[1,2,3],[1,2],[1,2,3,4]]);
val it = true : bool
```

''-Typvariablen können nur an Typausdrücke gebunden werden, die Typen mit Gleichheit bezeichnen:

```
- member((fn x => x), [(fn x => x)]);
Error: operator and operand don't agree [equality type required]
   operator domain: ''Z * ''Z list
   operand:         ('Y -> 'Y) * ('X -> 'X) list
   in expression:
     member ((fn x => x),(fn <pat> => <exp>) :: nil)
```

Die Gleichheit ist auf dem Typ `'x -> 'x` nicht definiert.

6.6 Typkonstruktor versus Wertkonstruktor

Ein Typkonstruktor darf nicht mit dem (Wert-)Konstruktor dieses Typs verwechselt werden. So sind z.B. der Typkonstruktor `list` und der Listenkonstruktor `cons` (`::`) zwei völlig verschiedene Konzepte:

- Mit `list` und einem Typausdruck wie z.B. `'a` oder `int * bool` werden der polymorphe Listentyp `'a list` und der Listentyp `(int * bool) list` gebildet. `list` und `*` sind in diesen Beispielen Typkonstruktoren.
- Mit `cons` (`::`) und einem (herkömmlichen) Ausdruck wie z.B. `1`, `"abc"` oder `(3, false)` werden wie folgt Listen gebildet:

  ```
  - 1 :: [];
  val it = [1] : int list

  - "abc" :: [];
  ```

```
val it = ["abc"] : string list

- (3, false) :: [];
val it = [(3,false)] : (int * bool) list
```

`cons` ist ein (Wert-)Konstruktor.

Die Unterscheidung gilt für alle zusammengesetzten Typen. In der folgenden Tabelle werden die Argumente der Typkonstruktoren und der Konstruktoren der jeweiligen Typen mit `.` dargestellt:

Typkonstruktor	**(Wert-)Konstruktor**
`. list`	`. :: .`
	`nil`
`. * .`	`( . , . )`
`{ . : . , . : . }`	`{ . = . , . = . }`
`. -> .`	`fn . => .`

Der allgemeine Fall ist, dass es zu *einem* Typkonstruktor *mehrere*, aber endlich viele (Wert-)Konstruktoren des Typs geben kann. Zum Beispiel sind `cons` (`::`) und `nil` die beiden (Wert-)Konstruktoren eines Listentyps. `nil` ist ein 0-stelliger (Wert-)Konstruktor, d.h. eine Konstante. Im Kapitel 8 werden Typen eingeführt, die mehrere (Wert-)Konstruktoren einer Stelligkeit größer-gleich 1 haben.

6.7 Schlussregeln für die Typinferenz

Im Abschnitt 6.5 wurde die Typinferenz informell eingeführt. Wir wollen sie nun formal und als Algorithmus für eine Vereinfachung von SML definieren. Ist die Typinferenz so definiert, so lässt sie sich formal untersuchen, z.B. auf Korrektheit prüfen, und auch als Programm implementieren.

6.7.1 Eine Vereinfachung von SML: SMalL

Die Programmiersprache SMalL ist eine Vereinfachung von SML. SMalL enthält die `val`–Deklarationen, die vordefinierten Typen `bool`, `int`, `real`, die Listen und Vektoren mit ihren vordefinierten Funktionen, das `fn`–Konstrukt zur Funktionsbildung, die Fallunterscheidung mit `if-then-else` und natürlich die Typ-Constraints.

SMalL lässt kein Pattern Matching, keine Verbunde, weder `let`– noch `local`–Ausdrücke zu. Zudem besitzt SMalL keines der Konstrukte von SML (zur Spezifikationen von u.a. Ausnahmen und Modulen), die bisher nicht eingeführt wurden.

Der Einfachheit halber und ohne Beschränkung der Allgemeinheit wird angenommen, dass alle Funktionen in SMalL präfix notiert werden.

Wir erinnern daran, dass eine n-stellige SML- oder SMalL-Funktion als eine einstellige Funktion angesehen werden kann, deren (einziger) Parameter ein n-Vektor ist (siehe Abschnitt 5.3.1). So ist es möglich, jede Funktionsanwendung `F(P1, ..., Pn)` als Objekt der Gestalt `F P` anzusehen.

6.7.2 Logischer Kalkül

Zur Spezifikation eines Algorithmus für die Typinferenz in SMalL-Programmen bedienen wir uns des Ansatzes eines „logischen Kalküls“. Ein logischer Kalkül besteht aus:

1. einer (formalen) Sprache, in der Aussagen über die zu beweisenden Eigenschaften ausgedrückt werden;
2. Schlussregeln, womit weitere Aussagen aus bereits festgestellten Aussagen geschlossen werden können;
3. Axiome und Annahmen, d.h. Aussagen, deren Gültigkeit nicht bewiesen werden muss, weil sie immer gelten (Axiome) oder weil sie angenommen werden (Annahme).

Im Fall des (logischen) Kalküls für die Typinferenz gilt:

- die (formale) Sprache ist der Formalismus, in dem die Typ-Constraints (oder Typisierungsaussagen) wie etwa `(fn x => x) : 'a -> 'a` ausgedrückt werden;
- die Axiome sind die Typ-Constraints für die vordefinierten Typen wie etwa

  `34 : int`, `~98.67e4 : real`, `false : bool`, `nil : 'a list`, und
  `@ : ('a list * 'a list) -> 'a list`
  (`@` ist die vordefinierte Infixfunktion `append` von SML)

- die Annahmen sind die Typ-Constraints aus dem Programm — wie etwa: `x : int` oder `(fn x => a*x + b*x**2) : real -> real` — sowie für jeden Ausdruck oder Teilausdruck `A` aus dem Programm, für den das Programm kein Typ-Constraint enthält, ein Typ-Constraint `A : V` mit `V` Typvariable, so dass für jedes Paar mit verschiedenen Ausdrücken `A` die Typvariablen `V` paarweise verschieden sind.

Die Axiome werden „Typisierungsaxiome“, die Annahmen „Typisierungsannahmen“ genannt.

6.7.3 Gestalt der Schlussregeln eines logischen Kalküls

Eine Schlussregel hat die folgende Gestalt

$$\frac{Pr_1 \ \ldots \ Pr_n}{Sch}$$

wobei $n \geq 1$, $Pr_1, \ldots, Pr_n$ und Sch Ausdrücke in der zugrunde liegenden formalen Sprache, also in der formalen Sprache des logischen Kalküls, sind. $Pr_1, \ldots, Pr_n$ sind die „Prämissen“, Sch der „Schluss“. Eine solche Regel bedeutet: Aus Pr_1 und ... und Pr_n folgt logisch Sch.
Verlangt eine Schlussregel eine Annahme A, so hat sie die Gestalt (Annahmen (A) können über jeder Prämisse stehen):

$$\frac{\begin{array}{c}(A)\\ Pr_1 \quad \ldots \quad Pr_n\end{array}}{Sch}$$

Mögliche Schlussregeln wären z.B.:

$$\texttt{R1}: \frac{\texttt{append:('a list * 'a list) -> 'a list} \quad \texttt{[1,2]:int list} \quad \texttt{[3,4]:int list}}{\texttt{append([1,2], [3,4]):int list}}$$

$$\texttt{R2}: \frac{\texttt{1 : int} \quad \texttt{2 : int}}{\texttt{[1,2] : int list}}$$

$$\texttt{R3}: \frac{\texttt{0 : int} \quad \texttt{[1,2] : int list}}{\texttt{[0,1,2] : int list}}$$

Obwohl sinnvoll und korrekt gebildet sind diese Schlussregeln keine geeigneten Schlussregeln des logischen Kalküls für die Typinferenz, weil sie zu speziell sind. Es werden Schlussregeln bevorzugt, die weder für eine besondere Prozedur noch für bestimmte konkrete aktuelle Parameter definiert sind.
Eine bekannte Schlussregel ist der „*modus ponens*“ für Formeln der Aussagenlogik:

$$\frac{a \qquad a \Rightarrow b}{b}$$

Eine andere bekannte Schlussregel ist die „*Kontraposition*“, ebenfalls für Formeln der Aussagenlogik:

$$\frac{a \Rightarrow b}{(not\ b) \Rightarrow (not\ a)}$$

In den Schlussregeln „modus ponens“ und „Kontraposition“ stehen a und b für aussagenlogische Formeln. a und b sind also Variablen der Metasprache, die zur Definition der Schlussregeln verwendet werden, aber keine aussagenlogischen Variablen.

6.7.4 Beweisbegriff in logischen Kalkülen

Ist ein logischer Kalkül durch eine formale Sprache, Schlussregeln und Axiome sowie möglichen Annahmen definiert, so werden Beweise als Bäume (im Sinne der Graphentheorie) definiert, deren Blätter Axiome und Annahmen sind und wie folgt von den Blättern her aufgebaut werden können:

1. Axiome und Annahmen sind Beweise.
2. Sind $B_1, \ldots, B_n$ Beweise mit Wurzel $P_1, \ldots$ bzw. P_n und ist

   $$\frac{P_1 \quad \ldots \quad P_n}{S}$$

   eine Schlussregel, so ist der Baum

   $$\frac{B_1 \quad \ldots \quad B_n}{S}$$

   ein Beweis (mit S als Wurzel, $B_1, \ldots, B_n$ als Unterbäume).

Oft werden zusätzliche Bedingungen gestellt, die so gebildete Bäume erfüllen müssen, um Beweise des logischen Kalküls zu sein.
Die Bäume, die aus den Axiomen und Annahmen unter Verwendung der Schlussregeln gebildet werden, haben ein paar bemerkenswerte Merkmale, die charakteristisch für die Beweise eines logischen Kalküls sind:

1. Ihre Kanten werden durch waagerechte Linien repräsentiert;
2. Ihre Wurzel befindet sich unten, ihre Blätter befinden sich oben.

Letzteres ist deshalb bemerkenswert, weil in der Informatik Bäume üblicherweise „verkehrt herum“, also mit der Wurzel nach oben und den Blättern nach unten, dargestellt werden).

Beispiel:
Mit den zuvor angegebenen Schlussregeln `R2` und `R3` und den Axiomen `0:int`, `1:int` und `2:int` lässt sich der folgende Beweis bilden:

$$\dfrac{\texttt{0:int} \qquad \dfrac{\texttt{1:int} \quad \texttt{2:int}}{\texttt{[1,2]:int list}}\,\texttt{R2}}{\texttt{[0,1,2]:int list}}\,\texttt{R3}$$

Verlangt eine Schlussregel eine Annahme A, so kann sie nur dann auf Beweise B_1, ..., B_n zur Bildung eines Beweises B angewandt werden, wenn einer der Beweise B_1, ..., B_n diese Annahme A als Blatt besitzt.

6.7.5 Die Schlussregeln für die Typinferenz oder „Typisierungsregeln“

Im Folgenden stehen `T`, $\texttt{T}_\texttt{i}$ für Typausdrücke, `V` für eine Typvariable, `F`, `B`, `P`, `A`, $\texttt{A}_\texttt{i}$ für Ausdrücke:

T1 (Funktionsanwendung): $\dfrac{\texttt{F} : \texttt{T}_1\texttt{->T}_2 \qquad \texttt{P} : \texttt{T}_1}{\texttt{F P} : \texttt{T}_2}$

T2 (if-then-else): $\dfrac{\texttt{B} : \texttt{bool} \quad \texttt{A}_1 : \texttt{T} \quad \texttt{A}_2 : \texttt{T}}{(\texttt{if B then A}_1\texttt{ else A}_2) : \texttt{T}}$

T3 (Funktionsbildung oder Lambda-Abstraktion): $\dfrac{\begin{array}{c}(\texttt{P} : \texttt{T}_1)\\ \texttt{A} : \texttt{T}_2\end{array}}{(\texttt{fn P => A}) : \texttt{T}_1\texttt{ -> T}_2}$

T4 (Instanziierung): $\dfrac{\texttt{A} : \texttt{V}}{\texttt{A} : \texttt{T}}$ mit `V` Typvariable und `T` Typausdruck, wobei `V` nicht in `T` vorkommt

Die Bedingung der Schlussregel T4 ist notwendig: Was wäre wohl der Typ `'a`, der gleich mit dem Typ `'a list` wäre?

Die Definition der Beweise verlangt, dass eine Bindung einer Typvariablen `V` an einem Typausdruck `T` unter Verwendung der Schlussregel T4 nicht nur für *ein*, sondern für *alle* Vorkommen von `V`, die gemäß der Blockstruktur dasselbe Objekt bezeichnen, im Beweis gilt.

$$\text{T5 (Einführung von Konstruktoren):} \quad \frac{\mathtt{A_1 : T} \qquad \mathtt{A_2 : T\ list}}{\mathtt{(A_1 :: A_2)\ :\ T\ list}}$$

$$\text{T5-}n\text{-Vektor:} \quad \frac{\mathtt{A_1 : T} \quad \ldots \quad \mathtt{A_n : T}}{\mathtt{(A_1, \ldots, A_n)} : \underbrace{\mathtt{T\ *\ \ldots\ *\ T}}_{(\mathtt{n}\text{ mal})}}$$

Für jedes $n \geq 1$ gibt es eine Regel T5-n-Vektor. Da ein Programm endlich ist, kann man sich für jedes gegebene Programm auf eine endliche Anzahl von Schlussregeln beschränken, nämlich alle Schlussregeln T1 bis T4 und nur die Schlussregeln T5-n-Vektor für n-Vektoren einer Stelligkeit n, die im Programm vorkommen.

6.7.6 Typisierungsbeweise

Typisierungsbeweise werden wie folgt definiert:

1. Typisierungsaxiome und -annahmen sind Typisierungsbeweise.
2. Ist

   $$\frac{P_1 \quad \ldots \quad P_n}{S}$$

   eine Typisierungsregel und sind $B_1, \ldots B_n$ Typisierungsbeweise von $P_1, \ldots$ bzw. P_n, so ist

   $$C = \frac{B_1 \quad \ldots \quad B_n}{S}$$

   ein Typisierungsbeweis, wenn in C für alle Ausdrücke A, die gemäß der Blockstruktur dasselbe Objekt bezeichnen, alle Anwendungen der Instanziierungsregel T4 mit Prämisse $A : V$ denselben Schluss $A : T$ (also mit demselben Typausdruck T) haben.

6.7.7 Beispiele für Typisierungsbeweise

Beispiel der Festlegung des Typs einer Funktion:

Seien die SMalL-Funktion

```
val f = fn x => +(*(a, x), 2.0)      (*   a*x + 2.0   *)
```

und die folgende Typisierungsannahme gegeben:

```
a : real
```

Der Typ `real -> real` von `f` lässt sich durch den folgenden Typisierungsbeweis feststellen:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{+:(real*real)->real}}\text{T1:}\quad \dfrac{\dfrac{\dfrac{}{\texttt{*:(real*real)->real}}\quad \dfrac{\dfrac{\dfrac{}{\texttt{x:'x}}\text{T4:}}{\texttt{x:real}}\ \textit{(Ann.)}\quad \dfrac{}{\texttt{a:real}}}{\texttt{(a,x):real*real}}\text{T5-2:}}{\texttt{*(a,x):real}}\text{T1:}\quad \dfrac{}{\texttt{2.0:real}}\ \textit{(Axiom)}}{\texttt{(*(a, x),2.0): real * real}}\text{T5-2:}}{\texttt{+(*(a, x),2.0) : real}}\text{T1:}}{\texttt{(fn x => +(*(a, x),2.0)) : real -> real}}\text{T3:}$$

Beispiel einer Überprüfung eines angegebenen Typs:

Sei nun die Funktion `f` wie folgt definiert:

```
val f = fn (x:int) => +(*(a, x), 2.0)      (*  a*x + 2.0  *)
```

Der zuvor geführte Beweis lässt sich nicht mehr aufbauen, weil sich mit der Typisierungsannahme `x : int` kein Beweis von `*(a, x) : real` bilden lässt.

Es ist bemerkenswert, dass der Beweis des ersten Beispiels genau die Struktur des Ausdrucks `(fn x => +(*(a, x), 2.0))` widerspiegelt: Wird der Beweis von seiner Wurzel zu seinen Blätter, d.h. Annahmen oder Axiome durchlaufen, so zerlegt der Beweis den Ausdruck in seinen Teilausdrücke. So könnte der Beweis wie folgt aufgebaut werden:

(*) `(fn x => +(*(a, x), 2.0)) : 'p1 -> 'p2` da es ein `fn`-Ausdruck ist
`+(*(a, x), 2.0) : 'p2`

`+ : ('p2 * 'p2) -> 'p2` Axiom
`2.0 : real` Axiom

Folglich: `'p2` = `real` und `*(a, x): real`

`* : (real * real) -> real` Axiom (bereits instanziiert)

Folglich: `x : real` und `'p1 : real`

Also:

`(fn x => +(*(a, x), 2.0)) : real -> real`

Das Prinzip, das sich hier anhand eines Beispiels erkennen lässt, gilt allgemein. Darauf beruht die Automatisierung der Typprüfung.

Der Beweis des ersten Beispiels beinhaltet einen „Eureka-Schritt“, der schwer zu automatisieren ist: die Anwendung der Instanziierungsschlussregel T4 auf `x : 'x`, um `x : real` zu erhalten. Da es unendlich viele Typen gibt, die Kandidaten zur Bindung von `'x` sind,

lässt sich daraus nur schwer ein brauchbarer Algorithmus entwickeln. Die Lösung dieses Problems liegt darin, Typvariablen nur soweit wie nötig an Typausdrücke zu binden — wie bereits im Beweis (*) angewendet.

Für die Bindung von Typvariablen „nur soweit wie nötig“ wird der sogenannte Unifikationsalgorithmus verwendet.

6.8 Der Unifikationsalgorithmus

Im Folgenden bezeichnen V, V_i Typvariablen, T, T_i Typausdrücke, K, K_i Typkonstanten.
Zur Unifikation zweier Typausdrücke TA_1 und TA_2 gehe wie folgt vor:

Initialisierung: $M := \{(TA_1, TA_2)\}$ und $U := \{\}$
(U ist eine Menge von Paaren, die Gleichungen zur Unifikation von TA_1 und TA_2 darstellen).
Im Unifikationsalgorithmus stellt M eine Menge von Paaren dar, die Gleichungen repräsentieren. Anfangs enthält M nur ein Paar, das die beiden Typausdrücke enthält, deren Unifizierbarkeit überprüft werden soll.
U stellt ebenfalls eine Menge von Paaren dar, die Gleichungen repräsentieren. Anfangs ist U leer. Terminiert der Unifikationsalgorithmus mit dem Ergebnis, dass die beiden Typausdrücke des Aufrufes unifizierbar sind, so enthält U die Gleichungen (repräsentiert als Paare), die diese Ausdrücke gleich machen, d.h. unifizieren.
Die Mengen M und U werden so lange wie möglich und so lange keine erfolglose Terminierung gemeldet wird wie folgt verändert. Wähle (willkürlich) ein Paar (T_1, T_2) aus M und verändere M und U, je nach dem, welche Gestalt T_1 und T_2 haben:

1. Falls $(T_1, T_2) \in M$ und T_1 eine Typvariable ist, dann streiche (T_1, T_2) aus M, ersetze in M jedes Vorkommen von T_1 durch T_2 und füge (T_1, T_2) in U ein.
2. Falls $(T_1, T_2) \in M$ und T_1 eine Typkonstante ist
   (a) Wenn T_2 dieselbe Typkonstante ist, dann streiche (T_1, T_2) aus M.
   (b) Andernfalls wenn T_2 keine Typvariable ist, dann terminiere erfolglos.
3. Falls $(T_1, T_2) \in M$ und T_1 ein zusammengesetzter Typausdruck ist, der aus einem (Präfix-, Infix- oder Postfix-)Typoperator `Op` und den Typausdrücken $T_{11}, \ldots, T_{1n}$ (in dieser Reihenfolge) besteht
   (a) Wenn T_2 ebenfalls aus demselben (Präfix-, Infix- oder Postfix-)Typoperator `Op` und den Typausdrücken $T_{21}, \ldots, T_{2n}$ (in dieser Reihenfolge) besteht, dann ersetze (T_1, T_2) in M durch die n Paare $(T_{11}, T_{21}), \ldots, (T_{1n}, T_{2n})$.
   (b) Andernfalls wenn T_2 keine Typvariable ist, dann terminiere erfolglos.
4. Falls $(T_1, T_2) \in M$, T_1 keine Typvariable ist und T_2 eine Typvariable ist, dann ersetze (T_1, T_2) in M durch (T_2, T_1).

Ist kein Fall mehr anwendbar, dann liefere U als Unifikator von T_1 und T_2 und terminiere.

Beispiel: Anwendung des Unifikationsalgorithmus

[Im Folgenden wird immer das erste Paar von M ausgewählt.]

$M = \{$(`'a -> 'a list`, `int list -> 'b`)$\}$,	$U = \{\}$	Fall 3 trifft zu
$M = \{$(`'a`, `int list`), (`'a list`, `'b`)$\}$,	$U = \{\}$	Fall 1 trifft zu
$M = \{$(`int list list`, `'b`)$\}$,	$U = \{$(`'a`, `int list`)$\}$	Fall 4 trifft zu
$M = \{$(`'b`, `int list list`)$\}$,	$U = \{$(`'a`, `int list`)$\}$	Fall 1 trifft zu
$M = \{\}$,	$U = \{$(`'a`, `int list`), (`'b`, `int list list`)$\}$	

U wird als Unifikator von `'a -> 'a list` und `int list -> 'b` geliefert.

6.9 Ein Verfahren zur automatischen Typinferenz

6.9.1 Prinzip des Verfahrens

Das Verfahren besteht aus zwei Phasen, deren Ausführungen beliebig ineinander verzahnt werden dürfen.
Das Verfahren erhält als Parameter

- einen Ausdruck `A` (mit oder ohne Typ-Constraint), dessen Typisierung überprüft oder festgestellt werden soll, und
- eine Menge M von Typ-Constraints. Beim ersten Aufruf des Verfahrens ist M leer.

Tabelle 6.1 (Seite 136) enthält die Beschreibung des Verfahrens.

6.9.2 Behandlung der Überschattung

Obwohl Deklarationen von lokalen Variablen mit `let`– oder `local`–Ausdrücken in SMalL nicht möglich sind, können die formalen Parameter einer Funktionsdefinition, d.h. in einem `fn`–Ausdruck `(fn . => .)` Namen überschatten.
Zur Behandlung der Überschattung wird eine Namensliste oder „Typisierungsumgebung“ verwendet, die wie eine Umgebung verwaltet wird:

- Stößt man während der Phase 1 auf einen Ausdruck, der eine `val`–Deklaration oder ein `fn`–Ausdruck ist, dann wird der im `val`–Ausdruck deklarierte Name oder die formalen Parameter des `fn`–Ausdrucks am Anfang der „Typisierungsumgebung“ eingefügt.
- Wird während der Phase 1 eine `val`–Deklaration oder ein `fn`–Ausdruck verlassen, so werden alle Namen von der Typisierungsumgebung entfernt, die beim Eintritt in diese `val`–Deklaration oder diesen `fn`–Ausdruck in die Typisierungsumgebung eingefügt wurden.
- Die Typ-Constraints, die während der Phase 2 einem Ausdruck zugeordnet werden, werden dem zutreffenden Ausdruck der Typisierungsumgebung zugeordnet. Dazu wird die Typisierungsumgebung vom Anfang her durchsucht.

Tabelle 6.1: Unifikationsalgorithmus

Phase 1: Hat der Ausdruck `A` einen (vorgegebenen) Typ-Constraint `C`, so füge $\texttt{A} : \texttt{C}$ in M ein.

1. Hat `A` die Gestalt `val N = W` (oder `val rec N = W`), so
   (a) gebe `N` (bzw. `W`) alle in M vorkommenden Typ-Constraints von `W` (bzw. von `N`).
   (b) Wende das Verfahren mit Ausdruck `W` und Menge M an.
2. Andernfalls gehe nach der Gestalt von `A` (entsprechend den Typisierungsregeln T1 – T5) wie folgt vor, wobei $\texttt{V}_1$ und $\texttt{V}_2$ bzw. `W` Typvariablen sind, die weder in `A` noch in M vorkommen:
   (a) Falls `A` von der Form `B C` ist, so füge in M die folgenden Typ-Constraints ein:
   $$\texttt{A} : \texttt{V}_2, \quad \texttt{B} : \texttt{V}_1 \texttt{ -> } \texttt{V}_2, \quad \texttt{C} : \texttt{V}_1$$
   Wende das Verfahren auf Ausdruck `B` und Menge M an.
   Wende das Verfahren auf Ausdruck `C` und die Menge an, die sich aus der Anwendung des Verfahrens auf `B` und `M` ergeben hat.
   (b) Falls `A` von der Form $\texttt{(if B then C}_1\texttt{ else C}_2\texttt{)}$ ist, so füge in M die folgenden Typ-Constraints ein:
   $$\texttt{A} : \texttt{V}, \quad \texttt{B} : \texttt{bool}, \quad \texttt{C}_1 : \texttt{V}, \quad \texttt{C}_2 : \texttt{V}$$
   Wende das Verfahren auf Ausdruck `B` und Menge M an.
   Wende dann das Verfahren auf Ausdruck $\texttt{C}_1$ und die Menge M_1 an, die sich aus der Anwendung des Verfahrens auf `B` und M ergeben hat.
   Wende das Verfahren auf Ausdruck $\texttt{C}_2$ und die Menge an, die sich aus der vorangegangenen Anwendung des Verfahrens auf $\texttt{C}_1$ ergeben hat.
   (c) Falls `A` von der Form `(fn P => B)` ist, so füge in M die folgenden Typ-Constraints ein:
   $$\texttt{A} : \texttt{V}_1 \texttt{ -> } \texttt{V}_2, \quad \texttt{P} : \texttt{V}_1, \quad \texttt{B} : \texttt{V}_2$$
   Wende das Verfahren mit Ausdruck `B` und Menge M an.

Phase 2: Kommen in M zwei Typ-Constraints $\texttt{A} : \texttt{T}_1$ und $\texttt{A} : \texttt{T}_2$ für denselben Ausdruck `A` vor, so unifiziere $\texttt{T}_1$ und $\texttt{T}_2$. Wenn die Unifikation von $\texttt{T}_1$ und $\texttt{T}_2$ erfolglos terminiert, dann melde, dass der Ausdruck `A` einen Typfehler enthält, und terminiere. Andernfalls binde die Typvariablen, wie es sich aus der Unifikation von $\texttt{T}_1$ und $\texttt{T}_2$ ergibt.

Da zur Typinferenz keine Auswertung stattfindet, werden auch keine Werte an Namen gebunden. Folglich reicht es aus, Namen statt Gleichungen der Gestalt „Namen = Wert“ in die Typisierungsumgebung aufzunehmen. Ferner werden keine lokale Umgebungen benötigt.

6.9.3 Beispiele

Beispiel 1:
Sei der Ausdruck A:

```
val f = fn x => +(*(a, x), 2.0)     (* d.h. a*x + 2.0 *)
```

Zerlegung des Ausdrucks:	Typisierungsumgebung (Anfang: unten)
`val f = fn x => +(*(a, x), 2.0)`	`f : 'f` `fn x => +(*(a, x), 2.0) : 'f`

`fn x => +(*(a, x), 2.0)`	`f : 'f`
	`fn x => +(*(a, x), 2.0) : 'f  'x->'v1`
	`x : 'x`

`+(*(a, x), 2.0)`	`f : 'f`
	`fn x => +(*(a, x), 2.0) : 'f  'x->'v1`
	`x : 'x`
	`+(*(a, x), 2.0)) : 'v1`

Unifikation: `'f = 'x->'v1`	

`+`	`f : 'x->'v1`
	`fn x => +(*(a, x), 2.0) : 'x->'v1`
	`x : 'x`
	`+(*(a, x), 2.0) : 'v1`
	`+ : 'v1*'v1->'v1`

`*(a, x)`	`f : 'x->'v1`
	`fn x => +(*(a, x), 2.0) : 'x->'v1`
	`x : 'x`
	`+ : 'v1*'v1->'v1`
	`*(a, x) : 'v1`

`*`	`f : 'x->'v1`
	`fn x => +(*(a, x), 2.0) : 'x->'v1`
	`x : 'x`
	`+ : 'v1*'v1->'v1`
	`*(a, x) : 'v1`
	`* : 'v2*'v2->'v2`

Unifikation: `'v2 = 'v1`	

a	f	: 'f
	fn x => +(*(a, x), 2.0)	: 'x->'f
	x	: 'x
	+	: 'v1*'v1->'v1
	*(a, x)	: 'v1
	*	: 'v1*'v1->'v1
	a	: 'v1

x	f	: 'x->'v1
	fn x => +(*(a, x), 2.0)	: 'x->'v1
	x	: 'x 'v1
	+	: 'v1*'v1->'v1
	*(a, x)	: 'v1
	*	: 'v1*'v1->'v1
	a	: 'v1

Unifikation: `'x = 'v1`

2.0	f	: 'v1->'v1
	fn x => +(*(a, x), 2.0)	: 'v1->'v1
	x	: 'v1
	+	: 'v1*'v1->'v1
	*(a, x)	: 'v1
	*	: 'v1*'v1->'v1
	a	: 'v1
	2.0	: 'v1 real

Unifikation: `'v1 = real`

Als Ergebnis sieht die Typisierungsumgebung nun wie folgt aus:

f	: real->real
fn x => +(*(a, x), 2.0)	: real->real
x	: real
+	: real*real->real
*(a, x)	: real
*	: real*real->real
a	: real
2.0	: real

Beispiel 2:
Sei der Ausdruck A:

```
val not = fn x => if x = true than false else true;
```

Dies führt zu folgenden Ablauf:

`val not = fn x => if x = true than false else true`

```
                not                                          : 'a
                fn x => if x = true than false else true     : 'a
```

```
fn x => if x = true than false else true
                not                                          : 'a
                fn x => if x = true than false else true     : 'a   'b->'c
                x                                            : 'b
                if x = true than false else true             : 'c

                Unifiziere 'a und 'b->'c                     'a = 'b -> 'c
```

```
if x = true than false else true
                not                                          : 'b -> 'c
                fn x => if x = true than false else true     : 'b -> 'c
                x                                            : 'b
                if x = true than false else true             : 'c   'd
                x = true                                     : bool
                false                                        : 'd   bool
                true                                         : 'd   bool
```

Unifiziere: 'c und 'd sowie 'd und bool: 'c = 'd
'd = bool

```
x = true        not                                          : 'b -> bool
                fn x => if x = true than false else true     : 'b -> bool
                x                                            : 'b   'e
                if x = true than false else true             : bool
                x = true                                     : bool
                false                                        : bool
                true                                         : bool
                =                                            : 'e * 'e -> bool
                true                                         : 'e   bool
```

Unifiziere: 'b und 'e sowie 'e und bool: 'b = 'e
'e = bool

```
                not                                          : bool -> bool
                fn x => if x = true than false else true     : bool -> bool
                x                                            : bool
                if x = true than false else true             : bool
                x = true                                     : bool
                false                                        : bool
                true                                         : bool
                =                                            : bool * bool -> bool
                true                                         : bool
```

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 7

Abstraktionsbildung mit Prozeduren höherer Ordnung

In SML sowie in anderen funktionalen Programmiersprachen sind Funktionen Werte. Wie alle Werte können Funktionen als Parameter von Funktionsaufrufen dienen. Ebenso können Funktionen als Ergebnis von Funktionsanwendungen geliefert werden. In früheren Kapiteln sowie in den Übungen sind schon Beispiele von solchen Funktionen gegeben worden. Funktionen, die Funktionen als Parameter oder Wert haben, heißen „Funktionen höherer Ordnung“. In diesem Kapitel werden sie systematisch untersucht. Selbstverständlich sind Prozeduren höherer Ordnung, die keine Funktionen sind, d.h. die Nebeneffekte haben, auch möglich.

7.1 Prozeduren als Parameter und Wert von Prozeduren

In SML und anderen funktionalen Programmiersprachen sind Funktionen Werte (siehe Abschnitt 2.4). Folglich können Funktionen wie andere Werte auch als aktuelle Parameter von Prozeduren, u.a. von Funktionen, auftreten. Prozeduren bzw. Funktionen, die als Parameter oder als Wert Funktionen haben, werden Prozeduren bzw. Funktionen höherer Ordnung genannt.

Der Bezeichnung „Prozeduren höherer Ordnung“ liegt die folgende Hierarchie zu Grunde:

- Die Ordnung 0 umfasst die Konstanten.
- Die Ordnung 1 umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung 0 sind.
- Die Ordnung n+1 umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung n sind.

Funktionen höherer Ordnung, die Funktionen unter ihren Parametern haben, werden manchmal „functionals“ und „Operatoren“ genannt.

Funktionen höherer Ordnung stellen eine nützliche Abstraktion dar, wie das folgende Beispiel zeigt.

Ist f eine (ableitbare) mathematische Funktion $\mathbb{R} \to \mathbb{R}$, so kann die Ableitung f' von f wie folgt geschätzt werden, wobei Δ (delta) eine kleine reelle Zahl ist (der tatsächliche Wert von $f'(x)$ ist bekanntlich der Limes dieses Quotienten für Δ gegen 0):

$$f'(x) = \frac{f(x+\Delta) - f(x)}{\Delta}$$

Diese Schätzung der Ableitung einer Funktion $\mathbb{R} \to \mathbb{R}$ lässt sich wie folgt in SML implementieren:

```
- val delta = 1e~5;
val delta = 1E~5 : real

- fun abl(f) = let
                   fun f'(x) = (f(x+delta) - f(x)) / delta
               in
                   f'
               end;
val abl = fn : (real -> real) -> real -> real
```

Die Funktion `abl` hat als Parameter eine Funktion `f`. Innerhalb von `abl` wird eine neue Funktion `f'` mit Hilfe von `f` und von `delta` definiert. Das Ergebnis von `abl(f)` entsteht nun nicht etwa durch Anwendung von `f'` auf irgendeinen Wert, sondern das Ergebnis ist diese neue Funktion `f'` selbst. Das bedeutet, dass `abl(f)` wiederum auf eine reelle Zahl angewandt werden kann.

Da der Typkonstruktor „`->`“ rechtsassoziativ ist, bezeichnet der Typausdruck `(real -> real) -> real -> real` den Typ `(real -> real) -> (real -> real)`. Das zeigt, dass `abl` als Parameter eine Funktion vom Typ `real -> real` hat und als Ergebnis ebenfalls eine Funktion vom Typ `real -> real` liefert. Diese Funktion kann, wie gesagt, wiederum auf eine reelle Zahl angewandt werden:

```
- abl( fn x => x*x )(5.0);
val it = 10.0000099994 : real

- abl( fn x => x*x*x )(5.0);
val it = 75.0001499966 : real
```

Die Ableitung der Funktion $x \mapsto x^2$ ist die Funktion $x \mapsto 2x$, und diese hat an der Stelle 5 den Wert 10.

Die Ableitung der Funktion $x \mapsto x^3$ ist die Funktion $x \mapsto 3x^2$, und diese hat an der Stelle 5 den Wert 75 ($= 3 * 25$).

Die mit SML berechneten Zahlen sind Schätzungen, die den richtigen Werten sehr nahe kommen. Mit anderen Werten für `delta` kann man die Schätzung verbessern. Da die Schätzung somit auch von `delta` abhängt, bietet es sich an, `delta` zu einem zweiten Parameter von `abl` zu machen:

```
- fun abl(f, delta:real) = let
                               fun f'(x) = (f(x+delta) - f(x)) / delta
                           in
                               f'
                           end;
val abl = fn : (real -> real) * real -> real -> real

- abl( fn x => x*x, 1e~10 )(5.0);
val it = 10.0000008274 : real

- abl( fn x => x*x*x, 1e~10 )(5.0);
val it = 75.0000594962 : real
```

Der Name `f'` für die neu definierte Funktion gilt nur lokal innerhalb der Definition von `abl` und ist bei genauerer Betrachtung gar nicht nötig. Man kann die neue Funktion genauso gut anonym definieren, das heißt, ohne ihr einen Namen zu geben, der außerhalb von `abl` sowieso nicht definiert ist. Damit kommt man zu folgender Definition, die der obigen völlig entspricht:

```
- fun abl(f, delta:real) = fn x => (f(x+delta) - f(x)) / delta;
val abl = fn : (real -> real) * real -> real -> real
```

Ebenfalls gleichwertig ist die Definition, die auf die „syntaktische Verzuckerung“ durch die Schreibweise mit `fun` ganz verzichtet:

```
- val abl = fn (f, delta:real) => fn x => (f(x+delta) - f(x)) / delta;
val abl = fn : (real -> real) * real -> real -> real
```

Egal welche der letzten drei Definitionen von `abl` man betrachtet, ist das Ergebnis von `abl` eine Funktion, die selbst wieder als Parameter von `abl` geeignet ist. Man kann `abl` also auch wie folgt verwenden:

```
- abl( abl(fn x => x*x, 1e~5), 1e~5 )(5.0);
val it = 2.00003569262 : real

- abl( abl(fn x => x*x*x, 1e~5), 1e~5 )(5.0);
val it = 30.0002511722 : real
```

Die Ableitung der Ableitung von $x \mapsto x^2$ ist die Funktion $x \mapsto 2$, und diese hat an der Stelle 5 den Wert 2.

Die Ableitung der Ableitung von $x \mapsto x^3$ ist die Funktion $x \mapsto 6x$, und diese hat an der Stelle 5 den Wert 30.

Ein weiteres Beispiel einer Funktionen höherer Ordnung ist die Identitätsfunktion (siehe Abschnitt 6.3.2):

```
- val id = fn x => x;
val id = fn : 'a -> 'a

- id(2);
val it = 2 : int

- id(id)(2);
val it = 2 : int
```

Im Teilausdruck `id(id)` hat die Funktion namens `id` sogar sich selbst als Parameter, und sie liefert auch sich selbst als Ergebnis. Da die Funktion eine Funktion als Parameter und auch als Wert haben kann, ist sie eine Funktion höherer Ordnung. Außerdem ist sie polymorph, kann also auf Werte verschiedener Typen angewandt werden. Die Polymorphie geht so weit, dass die Parameter und Ergebnisse sogar Objekte verschiedener Ordnungen sein können: Konstanten, Funktionen erster Ordnung oder Funktionen höherer Ordnungen. In der anfangs genannten Hierarchie von Ordnungen ist `id` für jedes $n \in \mathbb{N}$ mit $n > 0$ von der Ordnung n.

7.2 Currying

7.2.1 Prinzip

Betrachten wir die folgende mathematische Funktion:

$$\begin{array}{rcl} f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} & \rightarrow & \mathbb{Z} \\ (n_1, n_2, n_3) & \mapsto & n_1 + n_2 + n_3 \end{array}$$

f kann wie folgt in SML implementiert werden:

```
- fun f(n1, n2, n3) : int = n1 + n2 + n3;
val f = fn : int * int * int -> int
```

Bezeichnen wir mit $F(A, B)$ die Menge der Funktionen von A in B. Aus der Funktion f lässt sich die folgende Funktion definieren:

$$\begin{array}{rcl} f1 : \mathbb{Z} & \rightarrow & F(\mathbb{Z} \times \mathbb{Z}, \mathbb{Z}) \\ n_1 & \mapsto & f1(n_1) \end{array}$$

mit

$$\begin{array}{rcl} f1(n1) : \mathbb{Z} \times \mathbb{Z} & \rightarrow & \mathbb{Z} \\ (n_2, n_3) & \mapsto & f(n_1, n_2, n_3) = n_1 + n_2 + n_3 \end{array}$$

Die Funktion $f1$ kann wie folgt in SML implementiert werden:

```
- fun f1(n1)(n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- f1(1)(1, 1);
val it = 3 : int

- f1(1);
val it = fn : int * int -> int
```

Wegen der Rechtsassoziativität des Typkonstruktors „`->`“ und der Präzedenzen zwischen „`*`“ und „`->`“ bezeichnet `int -> int * int -> int` den Typ `int -> ((int * int) -> int)`. Die Funktion `f1` bildet also eine ganze Zahl auf eine Funktion vom Typ `((int * int) -> int)` ab.

In ähnlicher Weise lässt sich für jede ganze Zahl n_1 aus der Funktion $f1(n_1)$ die folgende Funktion definieren:

$$\begin{array}{rcl} f11(n_1) : \mathbb{Z} & \rightarrow & F(\mathbb{Z}, \mathbb{Z}) \\ n_2 & \mapsto & f11(n_1)(n_2) \end{array}$$

mit

$$\begin{array}{rcl} f11(n_1)(n_2) : \mathbb{Z} & \rightarrow & \mathbb{Z} \\ n_3 & \mapsto & f1(n_1)(n_2, n_3) = f(n_1, n_2, n_3) = n_1 + n_2 + n_3 \end{array}$$

Die Funktion $f11$ lässt sich wie folgt in SML implementieren:

```
- fun f11(n1)(n2)(n3) = f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

Wegen der Rechtsassoziativität des Typkonstruktors „`->`“ bezeichnet `int -> int -> int -> int` den Typ `int -> (int -> (int -> int))`.

```
- f11(1)(1)(1);
val it = 3 : int

- f11(1)(1);
val it = fn : int -> int

- f11(1);
 val it = fn : int -> int -> int
```

In dieser Weise lässt sich jede n-stellige Funktion durch einstellige (unäre) Funktionen darstellen. Eine praktische Folge davon ist, dass in vielen Fällen wesentlich kompaktere und übersichtlichere Schreibweisen von Funktionsdefinitionen möglich werden. Beispiele dafür kommen in späteren Abschnitten dieses Kapitels vor.

7.2.2 Andere Syntax zur Deklaration von „curried“ Funktionen

Da in SML ein einelementiger Vektor mit seinem Element identisch ist, lassen sich die Funktionen `f1` und `f11` auch wie folgt deklarieren:

```
- fun f1 n1 (n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- f1(1)(1, 1);
val it = 3 : int

- f1 1 (1, 1);
val it = 3 : int
```

```
- f1(1);
val it = fn : int * int -> int

- f1 1;
val it = fn : int * int -> int

- fun f11 n1 n2 n3 = f1 n1 (n2, n3);
val f11 = fn : int -> int -> int -> int

- f11(1)(1)(1);
val it = 3 : int

-  f11 1 1 1;
val it = 3 : int

- f11(1)(1);
val it = fn : int -> int

- f11 1 1;
val it = fn : int -> int

- f11(1);
val it = fn : int -> int -> int

- f11 1;
val it = fn : int -> int -> int
```

Unter Verwendung des `fn`-Konstrukts werden `f1` und `f11` wie folgt deklariert:

```
- val f1 = fn n1 => fn (n2, n3) => f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- val f11 = fn n1 => fn n2 => fn n3 => f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

Das (einfache!) Prinzip, das der Bildung von `f1` aus `f` und der Bildung von `f11` aus `f1` zugrunde liegt, wird nach dem Logiker Haskell B. Curry „Currying“ genannt.[1]

Die n-stellige Funktion, die sich aus der Anwendung des Currying auf eine $(n+1)$-stelligen Funktion f ergibt, wird „curried“ Form von f genannt.

7.2.3 Einfache Deklaration von curried Funktionen

Wie die Beispiele

```
- fun f11 n1 n2 n3 = ... ;
```

[1] Dieses Prinzip wird auch dem Logiker M. Schönfinkel zugeschrieben und deswegen im deutschsprachigen Raum manchmal „Schönfinkeln“ genannt.

oder

```
- val f11 = fn n1 => fn n2 => fn n3 => ... ;
```

zeigen, lassen sich curried Funktionen einfach dadurch deklarieren, dass ihre Parameter nicht als Vektor angegeben werden:

```
- fun f' n1 n2 n3 : int = n1 + n2 + n3;
val f' = fn : int -> int -> int -> int
```

Nun ist `f' n1` die Funktion, die `f1(n1)` entspricht, `f' n1 n2` die Funktion, die `f11(n1)(n2)` entspricht.

Curried Funktionen dürfen auch rekursiv sein, wie das folgende Beispiel zeigt (die Funktion `ggT` wurde im Abschnitt 4.5 eingeführt):

```
- fun ggT a b = if a < b then ggT b a
                         else if b = 0 then a
                                       else ggT b (a mod b);
val ggT = fn : int -> int -> int

- ggT 150 60;
val it = 30 : int

- ggT 150;
val it = fn : int -> int
```

Bemerkung: Die Schreibweise `ggT b a mod b` statt `ggT b (a mod b)` wäre inkorrekt: Wegen der Präzedenzen bezeichnet sie `((ggT b) a) mod b`.

7.2.4 Die Funktion höherer Ordnung `curry` zur Berechnung der curried Form einer binären Funktion

```
- fun curry(f) = fn x => fn y => f(x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

oder

```
- val curry = fn f => fn x => fn y => f(x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Man beachte, dass `curry` eine polymorphe Funktion ist.

```
- curry(fn (a,b) => 2*a + b);
val it = fn : int -> int -> int

- curry(fn (a,b) => 2*a + b) 3;
val it = fn : int -> int

- curry(fn (a,b) => 2*a + b) 3 1;
val it = 7 : int
```

7.2.5 Umkehrung der Funktion `curry`

```
- fun uncurry(f) = fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

oder

```
- val uncurry = fn f => fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

`uncurry` ist eine polymorphe Funktion:

```
- fun f x y = 2 + x + y;
val f = fn : int -> int -> int

- fun g x y = 2.0 + x + y;
val g = fn : real -> real -> real

- uncurry f;
val it = fn : int * int -> int

- uncurry f (1,1);
val it = 4 : int
```

`uncurry` ist die Umkehrung von `curry`:

```
- curry(uncurry(f));
val it = fn : int -> int -> int

- curry(uncurry(f)) 1 1;
val it = 4 : int

- uncurry(curry(uncurry(f)))(1,1);
val it = 4 : int
```

7.2.6 Nicht-curried und curried Funktionen im Vergleich

Am Beispiel einer einfachen Funktion können die Unterschiede zwischen herkömmlichen mehrstelligen Funktionen und Funktionen in curried Form erkannt werden. In diesem Beispiel wird angenommen, dass die vordefinierte Funktion „`*`“ den Typ `int * int -> int` hat.

	nicht-curried	**curried**

mögliche Deklaration	`fun mal(x, y) = x * y`
	`val mal = (fn (x, y) => x * y)`
		`fun mal x y = x * y`
		`val mal = (fn x y => x * y)`
		
		`fun mal x = (fn y => x * y)`
		`val mal = (fn x => (fn y => x * y))`

Typ	`int * int -> int`	`int -> int -> int`
Aufruf	`mal(2,3)` (hat Wert 6)	`mal 2 3` (hat Wert 6)
zu wenige Aufrufparameter	—	`mal 2` (hat eine Funktion als Wert)
	`val doppelt = fn y => mal(2, y)`	`val doppelt = mal 2`

7.3 Funktionskomposition

7.3.1 Funktionskomposition

Die SML-Standardbibliothek enthält eine Funktion höherer Ordnung, die wie folgt definiert werden kann:

```
- infix o;
infix o

- fun (f o g)(x) = f(g(x));
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- ((fn x => 2*x) o (fn x => x+1))(3);
val it = 8 : int

- Math.sqrt o ~;
val it = fn : real -> real

- val f = Math.sqrt o ~;
val f = fn : real -> real

- f(~4.0);
val it = 2.0 : real
```

Die Infixfunktion `o` (bzw. $\circ$) leistet eine Funktionskomposition in sogenannter „Anwendungsreihenfolge“ (oder Funktionalreihenfolge), d.h.:

$$(f \circ g)(x) = f(g(x))$$

In der Mathematik ist auch ein Funktionskomposition in sogenannter „Diagrammreihenfolge“ üblich, die wie folgt definiert ist:

$$(f \circ_d g)(x) = g(f(x))$$

Um Verwechslung mit dem SML-Funktionskompositionsoperator `o` zu vermeiden, verwenden wir hier die Notation $\mathtt{o_d}$ für den Funktionskompositionsoperator in Diagrammreihenfolge, der in der Mathematik üblicherweise $\circ$ notiert wird.

Die Bezeichnung „Diagrammreihenfolge“ kommt aus der folgenden Darstellung der Komposition zweier Funktionen, wobei f eine Funktion von A in B und g eine Funktion von B in C sind:

$$A \xrightarrow{f} B \xrightarrow{g} C$$

$(f \texttt{o}_\texttt{d} g)$ ist also eine Funktion von A in C. Die Reihenfolge f vor g in der Notation $(f \texttt{o}_\texttt{d} g)$ folgt der Reihenfolge im Diagramm, aber nicht der Reihenfolge der Funktionsanwendungen:

$$(f \texttt{o}_\texttt{d} g)(x) = g(f(x))$$

Im Gegensatz dazu entspricht die Anwendungsreihenfolge der Reihenfolge der Funktionsanwendungen:

$$(f \texttt{o} g)(x) = f(g(x))$$

Der Funktionskompositionsoperator `o` ist in SML vordefiniert.

7.3.2 Die Kombinatoren `I`, `K` und `S`

Die polymorphe Identitätsfunktion „`id`“ (siehe Abschnitt 4.4) wird auch als „`I`“ notiert und „Identitätskombinator“ genannt:

```
- fun I x = x;
val I = fn : 'a -> 'a

- I 5;
val it = 5 : int

- I [1,2];
val it = [1,2] : int list

- I (fn x => 2 * x);
val it = fn : int -> int
```

Der „Kompositionskombinator“ `S` ist eine Verallgemeinerung der Funktionskomposition `o`:

```
- fun S x y z = x z (y z);
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Der „Konstantenkombinator“ `K` entspricht der Projektion auf die erste Komponente:

```
- fun K x y = x;
val K = fn : 'a -> 'b -> 'a
```

Ein bekanntes Ergebnis ist, dass alle Funktionen des sogenannten Lambda-Kalküls alleine mittels `S` und `K` ausdrückbar sind. In anderen Worten kann man jeden Algorithmus nur unter Verwendung von `S` und `K` ausdrücken. Zum Beispiel gilt: Die Identitätsfunktion (oder der Identitätskombinator) `I` ist als `S K K` definierbar:

```
- S K K 5;
val it = 5 : int

- S K K [1,2];
val it = [1,2] : int list
```

Diese Kombinatoren spielen eine Rolle in der theoretischen Informatik. Außerdem gibt es auf diesen Kombinatoren basierende Ansätze für die Übersetzung von funktionalen Sprachen.

7.4 Grundlegende Funktionen höherer Ordnung

7.4.1 map

Die Funktion höherer Ordnung `map` dient dazu, eine unäre Funktion auf alle Elementen einer Liste anzuwenden, und als Wert die Liste der (Werte dieser) Funktionsanwendungen zu liefern:

```
- fun quadrat(x : int) = x * x;
val quadrat = fn : int -> int

- map quadrat [2, 3, 5];
val it = [4, 9, 25]

- map Math.sqrt [4.0, 9.0, 25.0];
val it = [2.0, 3.0, 5.0] : real list
```

Die Funktion `map` kann wie folgt in SML definiert werden:

```
- fun map f nil      = nil
    | map f (h :: t) = f(h) :: map f t;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

-  map (fn x:int => x*x) [1,2,3,4,5];
val it = [1,4,9,16,25] : int list
```

Man beachte, dass das Pattern Matching auf Funktionen in curried Form genauso anwendbar ist wie auf Funktionen mit einem Vektor als Parameter.

Die curried Form von `map` hat den Vorteil, dass `map` einfach auf Listen von Listen anzuwenden ist. Im folgenden Ausdruck

```
map (map quadrat) [[1,2], [3,4], [5,6]]
```

wird die Funktion `(map quadrat)` auf jedes (Listen-)Element der Liste `[[1,2], [3,4], [5,6]]` angewandt

```
- map quadrat;
val it = fn : int list -> int list

- map (map quadrat) [[1,2], [3,4], [5,6]];
val it = [[1,4], [9,16], [25,36]] : int list list
```

7.4.2 Vorteil der curried Form am Beispiel der Funktion `map`

Wäre `map` nicht in curried Form definiert, müsste man statt `(map quadrat)` einen komplizierteren Ausdruck verwenden, in dem die anonyme Funktion `(fn L => map quadrat L)` vorkommt.
Sei `map'` eine Version von `map`, die nicht in curried Form ist:

```
- fun map'(f, nil)    = nil
    | map'(f, h :: t) = f(h) :: map'(f, t);
val map' = fn : ('a -> 'b) * 'a list -> 'b list

- map'(quadrat, [2, 3, 5]);
val it = [4,9,25] : int list
```

Unter Verwendung von `map'` anstelle von

```
map (map quadrat) [[1,2], [3,4], [5,6]]
```

muss der kompliziertere Ausdruck

```
map'(fn L => map'(quadrat, L), [[1,2], [3,4], [5,6]])
```

verwendet werden:

```
- map'(fn L => map'(quadrat, L), [[1,2], [3,4], [5,6]]);
val it = [[1,4],[9,16],[25,36]] : int list list
```

Die Verwendung der anonymen Funktion `(fn L => map'(quadrat, L))` ist notwendig, weil im Gegensatz zu `map` die Funktion `map'` *zwei* Parameter verlangt.

7.4.3 `filter`

Diese Funktion höherer Ordnung filtert aus einer Liste alle Elemente heraus, die ein Prädikat erfüllen:

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool

- filter ist_gerade [1,2,3,4,5];
val it = [2,4] : int list

- fun filter pred nil      = nil
    | filter pred (h :: t) = if pred(h)
                             then h :: filter pred t
                             else filter pred t;
val filter = fn : ('a -> bool) -> 'a list -> 'a list

- filter (fn x => (x mod 2) = 0) [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- filter (not o (fn x => (x mod 2) = 0)) [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

Man beachte die Verwendung des (vordefinierten) Funktionskompositionsoperators `o`.
Auch `filter` ist in curried Form definiert, so dass kompaktere und übersichtlichere Definitionen möglich sind:

```
- val gerade_elemente = filter ist_gerade;
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente = filter (not o ist_gerade);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

7.4.4 Vorteil der curried Form am Beispiel der Funktion `filter`

Sei `filter'` eine Version von `filter`, die nicht in curried Form ist:

```
- fun filter'(pred, nil)      = nil
    | filter'(pred, (h :: t)) = if pred(h)
                                then h :: filter'(pred, t)
                                else filter'(pred, t);
val filter' = fn : ('a -> bool) * 'a list -> 'a list
```

Unter Verwendung von `filter'` statt `filter` müssen die Funktionen `gerade_elemente` und `ungerade_elemente` wie folgt definiert werden:

```
- val gerade_elemente = fn L => filter'(ist_gerade, L);
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente = fn L => filter'(not o ist_gerade, L);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

7.4.5 `foldl` und `foldr`

Die Funktionen höherer Ordnung `foldl` — zusammenfalten von links her — und `foldr` — zusammenfalten von rechts her — wenden wie folgt eine Funktion auf die Elemente einer Liste an:

`foldl f z [x1, x2, ..., xn]` entspricht `f(xn,...,f(x2, f(x1, z))...)`

`foldr f z [x1, x2, ..., xn]` entspricht `f(x1, f(x2, ..., f(xn, z)...))`

Beispiele:

```
- foldl (op +) 0 [2,3,5];     (* entspricht 5 + (3 + (2 + 0)) *)
val it = 10 : int

- foldr (op +) 0 [2,3,5];     (* entspricht 2 + (3 + (5 + 0)) *)
val it = 10 : int

- foldl (op -) 0 [7,10];      (* entspricht 10 - (7 - 0) *)
val it = 3 : int

- foldr (op -) 0 [7,10];      (* entspricht 7 - (10 - 0) *)
val it = ~3 : int
```

Die Funktionen können wie folgt in SML implementiert werden:

```
- fun foldl f z nil     =  z
    | foldl f z (x::L)  =  foldl f (f(x,z)) L;
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- fun foldr f z nil     =  z
    | foldr f z (x::L)  =  f(x, foldr f z L);
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Im zweiten Fall von `foldl` ist die Klammerung um `f(x,z)` notwendig, weil `foldl` eine curried Funktion ist. Im Ausdruck `foldl f f (x,z) L` würde `foldl` auf `f` angewandt werden und eine Funktion liefern, die dann auf `f` angewandt werden würde und nicht wie beabsichtigt auf `f(x,z)`.

Viele binäre Funktionen haben neutrale Elemente, zum Beispiel ist `0` das neutrale Element für `+` und `1` das neutrale Element für `*` und `nil` das neutrale Element für `@` (Listenkonkatenation) und `""` das neutrale Element für `^` (Stringkonkatenation). Die Funktionen `foldl` und `foldr` werden typischerweise so verwendet, dass das zweite Argument `z` das neutrale Element des ersten Arguments `f` ist.

Die wichtigste Anwendung von `foldl` und `foldr` ist die Definition von neuen Funktionen auf Listen mit Hilfe von binären Funktionen auf den Elementen der Listen:

```
- val listsum = fn L => foldl (op +) 0 L;
val listsum = fn : int list -> int

- val listprod = fn L => foldl (op * ) 1 L;
val listprod = fn : int list -> int

- val listconc = fn L => foldr (op ^) "" L;
val listconc = fn : string list -> string

- val listapp = fn L => foldr (op @) nil L;
val listapp = fn : 'a list list -> 'a list
```

```
- listsum [1,2];
val it = 3 : int

- listsum [1,2,3,4];
val it = 10 : int

- listprod [1,2];
val it = 2 : int

- listprod [1,2,3,4];
val it = 24 : int

- listconc ["abc", "de", "fghi", "j"];
val it = "abcdefghij" : string

- listapp [[1,2], [10,20,30], [100]];
val it = [1,2,10,20,30,100] : int list
```

Syntaktischer Hinweis: da `*)` das Symbol für das Ende eines Kommentars ist, führt der Ausdruck `(op *)` zu der folgenden Fehlermeldung:

```
Error: unmatched close comment
```

Zwischen `*` und `)` im Ausdruck `(op * )` muss also mindestens ein Leerzeichen vorkommen.

Man kann neue Funktionen wie `listsum`, `listprod` usw. natürlich explizit rekursiv definieren. Die Definitionen wären alle sehr ähnlich zueinander und würden sich nur an wenigen Stellen unterscheiden. Die Funktionen `foldl` und `foldr` sind Abstraktionen, die die Gemeinsamkeiten all dieser Definitionen darstellen und direkt zur Verfügung stellen, so dass man sie nicht bei jeder neuen Funktion wiederholen muss. Die Funktionen `foldl` und `foldr` sind damit sehr mächtige und grundlegende Hilfsmittel.

Da `foldl` und `foldr` curried Funktionen sind, haben sie den Vorteil, dass man neue Funktionen mit Hilfe von `foldl` und `foldr` definieren kann, ohne Namen für jedes Argument der neuen Funktionen erfinden zu müssen. Die kompakteste (und übersichtlichste) Definition der obigen Funktionen lautet:

```
- val listsum = foldl (op +) 0;
val listsum = fn : int list -> int

- val listprod = foldl (op * ) 1;
val listprod = fn : int list -> int

- val listconc = foldr (op ^) "";
val listconc = fn : string list -> string
```

Welche bekannten Listenfunktionen werden wie folgt mittels `foldl` und `foldr` definiert?

```
- val a = fn L => foldl (op ::) nil L;
val a = fn : 'a list -> 'a list

- a [1,2,3];
val it = [3,2,1] : int list

- val b = fn L => foldr (op ::) nil L;
val a = fn : 'a list -> 'a list

- b [1,2,3];
val it = [1,2,3] : int list
```

7.4.6 `exists` und `all`

Die Funktion höherer Ordnung `exists` überprüft, ob ein Prädikat für manche Elemente einer Liste erfüllt ist:

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool

- exists ist_gerade [1,2,3,4,5];
val it = true : bool

- exists ist_gerade [1,3,5];
val it = false : bool
```

Die Funktion `exists` kann wie folgt in SML implementiert werden:

```
- fun exists pred nil     =  false
    | exists pred (h::t)  =  (pred h) orelse exists pred t;
val exists = fn : ('a -> bool) -> 'a list -> bool
```

Die Funktion höherer Ordnung `all` überprüft, ob ein Prädikat für alle Elemente einer Liste erfüllt ist:

```
- all ist_gerade [1,2,3,4,5];
val it = false : bool

- all ist_gerade [2,4];
val it = true : bool
```

Die Funktion `all` kann wie folgt in SML implementiert werden:

```
- fun all pred nil     =  true
    | all pred (h::t)  =  (pred h) andalso all pred t;
val all = fn : ('a -> bool) -> 'a list -> bool
```

7.4.7 Wiederholte Funktionsanwendung

Angewandt auf eine Funktion `f` und eine natürliche Zahl `n` ist die Funktion höherer Ordnung `repeat` eine Funktion, die `n` mal die Funktion `f` anwendet:

```
- fun repeat f 0 x = x
    | repeat f n x = repeat f (n-1) (f x);
val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a

- repeat (fn x => x+1) 3 4;
val it = 7 : int

- repeat (fn x:int => x*x) 3 2;
val it = 256 : int
```

7.5 Beispiel: Ein Rekursionsschema zur Akkumulation

7.5.1 summe

Die folgende Funktion höherer Ordnung `summe` entspricht dem Summenzeichen, für das in der Mathematik die Sigma-Notation (Σ) üblich ist:

```
- fun summe(von, bis, schritt, funktion, akk) =
    if von > bis
    then akk
    else summe(von+schritt, bis, schritt, funktion, funktion(von)+akk);
val summe = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x), 0);
val it = 10 : int
```

7.5.2 produkt

Die folgende Funktion höherer Ordnung `produkt` entspricht dem Produktzeichen, für das in der Mathematik die Pi-Notation (Π) üblich ist:

```
- fun produkt(von, bis, schritt, funktion, akk) =
    if von > bis
    then akk
    else produkt(von+schritt, bis, schritt, funktion, funktion(von)*akk);
val produkt = fn : int * int * int * (int -> int) * int -> int

- produkt(1,4,1, (fn x => x), 1);
val it = 24 : int
```

7.5.3 Das gemeinsame Rekursionsschema

Das gemeinsame Rekursionsschema der Definitionen von `summe` und `produkt` kann wie folgt unabhängig von der verwendeten arithmetischen Operation definiert werden:

```
- fun akkumulieren(operation)(von, bis, schritt, funktion, akk) =
     if von > bis
     then akk
     else akkumulieren(operation)(von+schritt, bis, schritt, funktion,
                                  operation(funktion(von), akk));
val akkumulieren = fn
    ('a * 'b -> 'b) -> int * int * int * (int -> 'a) * 'b -> 'b

- val summe = akkumulieren (op +);
val summe = fn : int * int * int * (int -> int) * int -> int

- val produkt = akkumulieren (op * );
val produkt = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x), 0);
val it = 10 : int

- produkt(1,4,1,(fn x => x), 1);
val it = 24 : int
```

Welche Funktion wird wie folgt mittels `akkumulieren` definiert?

```
fun a n = akkumulieren (op * ) (1, n, 1, (fn x => x), 1);
```

7.5.4 Anwendung zur Integralschätzung

Eine Funktion `real_akkumulieren` kann wie folgt eingesetzt werden, um eine Schätzung des Integrals einer Funktion $\mathbb{R} \to \mathbb{R}$ zu definieren. Wir erinnern daran, dass das Integral einer Funktion f zwischen a und b, unter gewissen mathematischen Voraussetzungen wie der Stetigkeit von f, durch folgende Summe abgeschätzt werden kann:

$$\Delta * f(a + \Delta) + \Delta * f(a + 2\Delta) + \Delta * f(a + 3\Delta) + \ldots$$

```
- fun real_akkumulieren(operation)(von:real,
                                   bis:real,
                                   schritt:real,
                                   funktion,
                                   akk:real)  =
  if von > bis
  then akk
  else real_akkumulieren(operation)(von+schritt, bis, schritt, funktion,
                                    operation(funktion(von),akk) );
val real_akkumulieren = fn
:('a * real -> real) -> real * real * real * (real -> 'a) * real -> real
```

```
- fun integral(f, von, bis, delta) =
   real_akkumulieren(op +)(von+delta, bis, delta,
                           (fn x => delta*f(x)), 0.0);
val integral = fn : (real -> real) * real * real * real -> real

- integral((fn x => 1.0), 0.0, 3.0, 0.5);
val it = 3.0 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.5);
val it = 10.5 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.1);
val it = 8.7 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.0001);
val it = 8.9997 : real
```

Das unbestimmte Integral der Funktion $x \mapsto 2x$ nach x ist die Funktion $x \mapsto x^2$, das bestimmte Integral von 0 bis 3 hat also den Wert $3^2 - 0^2 = 9$. Die Größe von `delta` beeinflusst, wie nahe der Schätzwert bei diesem Wert liegt.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 8

Abstraktionsbildung mit neuen Typen

Moderne Programmiersprachen bieten nicht nur vordefinierte Basistypen (wie z.B. „ganze Zahlen“, „Boole'sche Werte“ und „Zeichenfolgen“) sowie zusammengesetzte Typen (wie z.B. die Vektoren-, Verbund- und Listentypen) an, sondern haben ein sogenanntes „erweiterbares Typsystem“. Das bedeutet, dass sie die Definition von neuen Typen nach den Anforderungen einer Programmieraufgabe ermöglichen. In diesem Kapitel wird die Definition von neuen Typen am Beispiel der Programmiersprache SML eingeführt. Zunächst wird das Hilfsmittel der Deklarationen von Typabkürzungen eingeführt. Dann wird die eigentliche Definition nichtrekursiver und rekursiver Typen in SML erläutert. Schließlich werden Programmierbeispiele behandelt.

8.1 Typen im Überblick

Wir erinnern kurz an den Begriff „Typ“ in Programmiersprachen, der bereits mehrmals behandelt wurde (vgl. insb. Abschnitte 2.2 und 5.1). Zudem führen wir die Begriffe „(Wert-)Konstruktoren“ und „(Wert-)Selektoren“ für zusammengesetzte Typen ein.

8.1.1 Typ als Wertemenge

Ein Typ (oder Datentyp) bezeichnet eine Menge von Werten. Diese Werte können atomar (wie z.B. die Werte des Typs „ganze Zahlen”: `int`) oder zusammengesetzt (wie z.B. die Werte des Typs „Listen von ganzen Zahlen”: `int list`) sein. Die Wertemenge, die ein Typ repräsentiert, kann endlich (wie z.B. im Falle des Typs „Boole'sche Werte”: `bool`) oder unendlich (wie im Falle des Typs „ganze Zahlen“ `int`) sein. Mit einem Typ werden üblicherweise Prozeduren zur Bearbeitung der Daten des Typs angeboten.

8.1.2 Typen mit atomaren und zusammengesetzten Werten

Ein Typ kann atomare Werte haben wie z.B. die Typen `bool` und `int`. Ein Typ kann auch zusammengesetzte Werte haben wie z.B. die Typen `int * int` und der Typ `'a list`.

Typen mit zusammengesetzten Werten werden auch zusammengesetzte Typen genannt, weil sie mit zusammengesetzten Typausdrücken wie etwa `real * real` oder `int list` oder `'a list` bezeichnet werden.

8.1.3 Typen in Programmiersprachen mit erweiterbaren Typsystemen

Ein Typ kann vordefiniert sein, d.h. von der Programmiersprache angeboten werden. Vordefinierte Typen in SML sind z.B. der Typ `int` („ganze Zahlen“) und der Typ `bool` („Boole'sche Werte“). Moderne Programmiersprachen ermöglichen auch die Definition von neuen Typen nach den Anforderungen einer Programmieraufgabe. Man spricht dann davon, dass diese Sprachen ein „erweiterbares Typsystem“ haben. Programmiersprachen mit erweiterbaren Typsystemen ermöglichen z.B. die Definition eines Typs „Wochentag“, eines Typs „Uhrzeit“, eines Typs „komplexe Zahl“ oder auch eines Typs „Übungsgruppe“ jeweils mit einer geeigneten Wertemenge (für diese Beispiele vgl. auch Abschnitt 5.1).

8.1.4 Monomorphe und Polymorphe Typen

Ein (atomarer oder zusammengesetzter) Typ kann monomorph sein. In diesem Fall kommt keine Typvariable im Typausdruck vor, der diesen Typ bezeichnet. Zum Beispiel sind `bool` und `int * int` monomorphe Typen von SML.

Ein (atomarer oder zusammengesetzter) Typ kann auch polymorph sein. In diesem Fall kommen eine oder mehrere Typvariablen im zugehörigen Typausdruck vor. Zum Beispiel ist `'a list` ein polymorpher Typ von SML.

8.1.5 (Wert-)Konstruktoren und (Wert-)Selektoren eines Typs

Zusammen mit einem Typ werden (Wert-)Konstruktoren (manchmal auch Operatoren genannt) definiert. Der vordefinierte (polymorphe) Typ „Liste“ hat z.B. zwei (Wert-)Konstruktoren: die leere Liste `nil` und den Operator `cons` (`::`).

(Wert-)Konstruktoren können 0-stellig sein (dann werden sie auch Konstanten genannt) oder eine beliebige andere Stelligkeit haben. Der (Wert-)Konstruktor `nil` des polymorphen Typs „Liste“ ist 0-stellig. Der (Wert-)Konstruktor `cons` (`::`) desselben Typs ist zweistellig (binär).

Für zusammengesetzte Typen werden auch sogenannte „(Wert-)Selektoren“ definiert, womit die zusammengesetzten Werte des Typs zerlegt werden können. Die (Wert-)Selektoren des vordefinierten (polymorphen) Typs „Liste“ sind die (vordefinierten) Funktionen `hd` (*head*) und `tl` (*tail*). (Wert-)Selektoren werden manchmal auch „Destruktoren“ genannt.

8.1.6 Typkonstruktoren

Zur Definition von Typen werden Typkonstruktoren angeboten, mit denen Typausdrücke zusammengesetzt werden können (siehe Abschnitt 6.5.5).

Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionen. Typkonstruktoren werden aber anders als Funktionsnamen verwendet:

- Wird eine Funktion auf aktuelle Parameter angewandt, so geschieht dies, um einen Wert zu berechnen.
- Wird ein Typkonstruktor auf Typausdrücke angewandt, so geschieht dies lediglich, um einen neuen Typausdruck zu bilden und so einen neuen Typ zu definieren. Dabei

findet keine Auswertung (im Sinne des Auswertungsalgorithmus aus Abschnitt 3.1.3) statt.

Vorsicht: Typkonstruktoren dürfen nicht mit (Wert-)Konstruktoren verwechselt werden (siehe Abschnitt 6.6).

8.2 Deklarationen von Typabkürzungen in SML: type-Deklarationen

8.2.1 Typabkürzungen

Im Abschnitt 5.3.2 wurde das folgende Beispiel eines Vektortyps eingeführt:

```
- type punkt = real * real;
type punkt = real * real

- fun abstand(p1: punkt, p2: punkt) =
       let fun quadrat(z) = z * z
           val delta_x = #1(p2) - #1(p1)
           val delta_y = #2(p2) - #2(p1)
       in
           Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
       end;
val abstand = fn : punkt * punkt -> real

- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Mit der `type`-Deklaration

```
- type punkt = real * real;
```

wird die Typkonstante `punkt` als Abkürzung für den Vektortyp `real * real` vereinbart. Eine solche Abkürzung, „Typabkürzung“ (*type abbreviation*) genannt, spezifiziert keinen neuen Typ, sondern lediglich ein Synonym für einen Typ.

Bietet eine Programmiersprache Typabkürzungen, so sagt man manchmal, dass die Programmiersprache eine „transparente Typbindung“ (*tranparent type binding*) ermöglicht.

8.2.2 Grenzen der Nutzung von Typabkürzungen

Benötigt man z.B. neben Kartesischen Koordinaten auch Polarkoordinaten, dann kann man die folgenden Typabkürzungen vereinbaren:

```
- type kartes_punkt = real * real;
  type kartes_punkt = real * real

- type polar_punkt = real * real;
  type polar_punkt = real * real
```

Da `punkt` und `kartes_punkt` beide denselben Typ `real * real` bezeichnen, ist keine Anpassung der Definition der Funktion `abstand` an die neueingeführte Typabkürzung `kartes_punkt` nötig:

```
- val A = (0.0, 0.0) : kartes_punkt;
val A = (0.0,0.0) : kartes_punkt

- val B = (1.0, 1.0) : kartes_punkt;
val B = (1.0,1.0) : kartes_punkt

- abstand (A, B);
val it = 1.41421356237 : real
```

Dies mag bequem erscheinen, ist aber gefährlich, weil die Funktion `abstand` auch auf Punkte in Polarkoordinaten angewendet werden kann, was aus der Sicht der Anwendung keinen Sinn ergibt:

```
- val C = (1.0, Math.pi/2.0) : polar_punkt;
val C = (1.0,1.57079632679) : polar_punkt

- abstand(B, C);
val it = 0.570796326795 : real
```

Der Punkt `C` hat ja die Kartesischen Koordinaten (0.0, 1.0), der Abstand zwischen `B` und `C` ist also in Wirklichkeit 1.0.

8.2.3 Nützlichkeit von Typabkürzungen: Erstes Beispiel

Die Nützlichkeit von Typabkürzungen ist am Beispiel der Funktionsdeklaration abstand ersichtlich. Würde SML Typabkürzungen nicht ermöglichen, so müsste die Funktion `abstand` wie folgt definiert werden:

```
- fun abstand(p1: real * real, p2: real * real) =
       let fun quadrat(z) = z * z
           val delta_x = #1(p2) - #1(p1)
           val delta_y = #2(p2) - #2(p1)
       in
           Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
       end;
val abstand = fn : (real * real) * (real * real) -> real
```

Diese Definition ist etwas weniger lesbar. Vor allem der ermittelte Typ

```
(real * real) * (real * real) -> real
```

der Funktion `abstand` ist wesentlich schlechter lesbar als

```
punkt * punkt -> real
```

8.2.4 Nützlichkeit von Typabkürzungen: Zweites Beispiel

Ein zweites Beispiel für die Nützlichkeit von Typabkürzungen ist:

```
- type person = {Name      : string,
                 Vorname   : string,
                 Anschrift : string,
                 Email     : string,
                 Tel       : int};
type person =
   {Anschrift:string, Email:string, Name:string,
                           Tel:int, Vorname:string}

- type kurz_person = {Name    : char,
                      Vorname : char,
                      Tel     : int};
type kurz_person = {Name:char, Tel:int, Vorname:char}

- fun kuerzen(x : person) : kurz_person=
       {Name    = String.sub(#Name(x), 0),
        Vorname = String.sub(#Vorname(x), 0),
        Tel     = #Tel(x)};
val kuerzen = fn : person -> kurz_person
```

Obwohl die Typausdrücke

```
person -> kurz_person
```

und

```
{Anschrift:string, Email:string, Name:string, Tel:int, Vorname:string}
-> {Name:char, Tel:int, Vorname:char}
```

denselben Typ bezeichnen, ist der erste Ausdruck lesbarer. Hier noch ein Beispiel für die Verwendung der Funktion:

```
- kuerzen {Name="Bry",  Vorname="Francois",
           Anschrift="D1.02",  Tel=2210,
           Email="bry@informatik.uni-muenchen.de"};
val it = {Name=#"B",Tel=2210,Vorname=#"F"} : kurz_person
```

8.2.5 Polymorphe Typabkürzungen

Typvariablen dürfen in `type`-Deklarationen vorkommen. Die Typabkürzungen, die so vereinbart werden, heißen „polymorphe Typabkürzungen“. `type`-Deklarationen, in denen Typvariablen vorkommen, heißen „parametrische `type`-Deklarationen“.

```
- type 'a menge = 'a -> bool;
type 'a menge = 'a -> bool

- val ziffer_menge : int menge =
     fn 0 => true
      | 1 => true
      | 2 => true
      | 3 => true
      | 4 => true
      | 5 => true
      | 6 => true
      | 7 => true
      | 8 => true
      | 9 => true
      | _ => false;
val ziffer_menge = fn : int menge

- ziffer_menge(4);
val it = true : bool

-  ziffer_menge(~12);
val it = false : bool
```

Die Sicht einer Menge als Funktion mit der Menge der Boole'schen Werte als Wertebereich ist übrigens in der Mathematik und in der Informatik geläufig. Solche Funktionen werden auch „Charakteristische Funktionen“ von Mengen genannt.

8.3 Definition von Typen: datatype-Deklarationen

Eine Typabkürzung definiert keinen neuen Typ. Neue Typen können in SML mit `datatype`- und `abstype`-Deklarationen vereinbart werden. In diesem Kapitel werden nur die Definitionen von neuen Typen mit `datatype`-Deklarationen behandelt. Die Definition von sogenannten „abstrakten Typen“ in SML mit `abstype`-Deklarationen wird in Kapitel 11 eingeführt.

8.3.1 Definition von Typen mit endlich vielen atomaren Werten

Ein Typ „Farbe“ bestehend aus der Wertemenge {`Rot`, `Gelb`, `Blau`} kann in SML wie folgt definiert werden:

```
- datatype Farbe = Rot | Gelb | Blau;
datatype Farbe = Blau | Gelb | Rot

- Rot;
val it = Rot : Farbe
```

Diese Deklaration legt das Folgende fest:

- Der Name `Farbe` ist eine Typkonstante.
- Die Typkonstante `Farbe` wird an die Wertemenge {`Rot`, `Gelb`, `Blau`} gebunden.
- Die Namen `Rot`, `Gelb` und `Blau` sind 0-stellige (Wert-)Konstruktoren.

Typen, die mit `datatype`-Deklarationen definiert werden, sind neue Typen ohne jegliche Entsprechung in den vordefinierten Typen. Ihre (Wert-)Konstruktoren können genauso verwendet werden, u.a. für Pattern Matching, wie die (Wert-)Konstruktoren von vordefinierten Typen:

```
- fun farbname(Rot)  = "rot"
    | farbname(Gelb) = "gelb"
    | farbname(Blau) = "blau";
val farbname = fn : Farbe -> string

- farbname(Gelb);
val it = "gelb" : string

- [Rot, Blau];
val it = [Rot,Blau] : Farbe list
```

Man beachte, dass die Definition der Funktion `farbname` einen Fall für jeden (Wert-)Konstruktor des Typs `Farbe` besitzt. Um Fehler zu vermeiden und für die Lesbarkeit des Programms ist es empfehlenswert, diese Fälle in der Reihenfolge der Typdefinition aufzulisten.

Es ist angebracht, den vordefinierten Typ „Boole'sche Werte“ als einen Typ anzusehen, der wie folgt hätte definiert werden können:

```
- datatype bool = true | false;
```

Eine solche Typdeklaration hätte den unerwünschten Effekt, die vordefinierten Funktionen des vordefinierten Typs bool „auszuschalten“, weil die Bindung der Typkonstanten `bool` an den benutzerdefinierten Typ die alte Bindung derselben Typkonstante an den vordefinierten Typ „Boole'sche Werte“ überschattet.

In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, stets über die Gleichheit, die implizit bei der `datatype`-Deklaration mit definiert wird:

```
- Blau = Blau;
val it = true : bool

- Blau = Rot;
val it = false : bool
```

In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, über keine implizit definierte Ordnung:

```
- Rot < Gelb;
Error: overloaded variable not defined at type
   symbol: <
   type: Farbe
```

Die folgenden Typdeklarationen sind also in SML austauschbar:

```
- datatype Farbe = Rot | Gelb | Blau;

- datatype Farbe = Gelb | Rot | Blau;
```

Andere Programmiersprachen würden aus der Reihenfolge der (Wert-)Konstruktoren in der Typdefinition die Ordnung `Rot` $<$ `Gelb` $<$ `Blau` implizit definieren.

8.3.2 Definition von Typen mit zusammengesetzten Werten

Nicht nur Typen, die Mengen von atomaren Werte bezeichnen, können definiert werden, sondern auch Typen mit zusammengesetzten Werten:

```
- datatype Preis = DM of real | EURO of real;
datatype Preis = DM of real | EURO of real
```

Diese Deklaration legt das Folgende fest:

- Der Name `Preis` ist eine Typkonstante.
- Die Typkonstante `Preis` wird an die Wertemenge (in mathematischer Notation) $\{\texttt{DM(x)}|\texttt{x} \in \mathbb{R}\} \cup \{\texttt{EURO(x)}|\texttt{x} \in \mathbb{R}\}$ gebunden.
- Die Namen `DM` und `EURO` sind unäre (einstellige) (Wert-)Konstruktoren, beide vom Typ `real -> Preis`.

```
- DM(4.5);
val it = DM 4.5 : Preis

- EURO(2.0);
val it = EURO 2.0 : Preis

- DM(1);
Error: operator and operand don't agree [literal]
   operator domain: real
   operand:         int
   in expression:
     DM 1
```

Man beachte die Syntax „`k of t2`“ in der Definition eines (Wert-)Konstruktors für einen Typ `t1` mit Bereich `t2`:

```
datatype t1 = ... | k of t2 | ...
```

anstelle der Schreibweise `k(t2)` oder `k t2`. Diese Syntax unterstreicht, dass ein (Wert-)Konstruktor eines Typs keine gewöhnliche Funktion ist (vgl. auch Abschnitt 6.5.1 und 8.1.6).

Wie für benutzerdefinierte Typen mit atomaren Werten ist Pattern Matching die bevorzugte Weise, Funktionen auf benutzerdefinierten Typen mit zusammengesetzten Werten zu definieren:

```
- fun wechseln(DM(x))   = EURO(0.51129 * x)
    | wechseln(EURO(x)) = DM(1.95583 * x);
val wechseln = fn : Preis -> Preis
```

Wir erinnern daran, dass es empfehlenswert ist, die Fälle beim Pattern Matching in der Reihenfolge der Typdefinition aufzulisten.

Die vorangehende Funktion `wechseln` rundet nach der zweiten Stelle nach dem Komma nicht ab. Unter Verwendung der vordefinierten Funktionen `real` und `round` kann `wechseln` wie folgt verbessert werden:

```
- fun wechseln(DM(x))   = EURO(real(round(0.51129 * x * 1e5)) * 1e~5)
    | wechseln(EURO(x)) =   DM(real(round(1.95583 * x * 1e5)) * 1e~5);
```

8.3.3 Gleichheit für Typen mit zusammengesetzten Werten

In SML verfügen auch benutzerdefinierte Typen, die Mengen von zusammengesetzten Werten bezeichnen, über die Gleichheit. Sie ist komponentenweise definiert:

```
- datatype zeitpunkt = Sek of int | Min of int | Std of int;
datatype zeitpunkt = Min of int | Sek of int | Std of int

- Sek(30) = Sek(0030);
val it = true : bool

- Min(60) = Std(1);
val it = false : bool

- Sek 2 = Sek(2);
val it = true : bool
```

Die Gleichheit ist auf benutzerdefinierten Typen mit zusammengesetzten Werten komponentenweise definiert: Zwei Werte sind gleich, wenn

- ihre (Wert-)Konstruktoren gleich sind (was für `Min(60)` und `Std(1)` nicht der Fall ist) und
- diese (Wert-)Konstruktoren auf identische Vektoren angewandt werden. Dabei muss berücksichtigt werden, dass in SML ein einstelliger Vektor mit seiner einzigen Komponente gleich ist (z.B. sind in SML `2` und `(2)` gleich).

Ist die Gleichheit auf dem Typ einer Komponente nicht definiert, so ist sie es auch nicht auf dem benutzerdefinierten Typ:

```
- DM(2.0) = DM(2.0);
Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         Preis * Preis
  in expression:
    DM 2.0 = DM 2.0
```

Wir erinnern daran, dass in SML die Gleichheit über den Gleitkommazahlen nicht vordefiniert ist (siehe Abschnitt 5.2.2).

8.3.4 „Typenmix“

Typdeklarationen ermöglichen es, Funktionen mit Parametern unterschiedlicher Grundtypen zu definieren:

```
- datatype nb = Int of int | Real of real;
datatype nb = Int of int | Real of real

- fun round(Int(x))  = Int(x)
    | round(Real(x)) = Int(trunc(x));
val round = fn : nb -> nb

- round(Int(56));
val it = Int 56 : nb

- round(Real(56.8976));
val it = Int 56 : nb
```

Ein weiteres Beispiel der Verwendung des Typs `nb` ist wie folgt:

```
- datatype nb = Int of int | Real of real;

- local fun real_abl f x  =
             let
                 val delta = 1E~10
             in
                 (f(x + delta) - f(x)) / delta
             end;
        fun konvertieren(Int(x))  = real(x)
          | konvertieren(Real(x)) = x
  in
    fun abl f (Int(x))   = Real(real_abl f  (real(x)))
      | abl f (Real(x)) =  Real(real_abl f x)
  end;
val abl = fn : (real -> real) -> nb -> nb

- abl (fn x => 2.0 * x) (Int(5));
val it = Real 2.00000016548 : nb

- abl (fn x => 2.0 * x) (Real(5.0));
val it = Real 2.00000016548 : nb
```

Man beachte, dass Ausdrücke, die mit den (Wert-)Konstruktoren `Int` und `Real` des Typs `nb` aufgebaut sind, in Klammern vorkommen. Dies ist wegen der Präzedenzen in SML notwendig.

8.4 Definition von rekursiven Typen

Ein Typ kann auch Werte unbegrenzter (aber endlicher) Größe haben. Beispiele dafür sind die Listen. Ein anderes Beispiel sind die sogenannten „Binärbäume“.

8.4.1 Wurzel, Blätter, Äste, Bäume und Wälder

Zunächst definieren wir die benötigten Begriffe:

Definition ((gerichteter) Graph)

Ein (gerichteter) Graph G ist ein Paar (Kn, Ka) mit $Ka \subseteq Kn \times Kn$.
Die Elemente von Kn werden *Knoten* von G genannt. Die Elemente von Ka sind die *Kanten* von G.
Ist Kn leer, so sagt man, dass der Graph $G = (Kn, Ka) = (\{\}, \{\})$ leer ist.
Ist $(k_1, k_2) \in Ka$, so heißt k_2 ein *Nachfolger* von k_1 (in G).

Definition (zusammenhängender Graph)

Ein (gerichteter) Graph $G = (Kn, Ka)$ heißt zusammenhängend, wenn es für jedes Paar (k_a, k_e) von Knoten eine Folge (*Pfad* genannt) von Knoten $k_1, \ldots, k_m \in Kn$ $(m \geq 1)$ gibt, so dass:

- $k_1 = k_a$
- $k_m = k_e$
- für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m-1$ ist $(k_i, k_{i+1}) \in Ka$ oder $(k_{i+1}, k_i) \in Ka$.

In anderen Worten kann man in einem zusammenhängenden Graph G von jedem Knoten von G über die Kanten von G jeden anderen Knoten von G erreichen, wobei die Kanten in beliebiger Richtung durchlaufen werden dürfen.

Definition (Zyklus in einem Graph)

Ein Zyklus in einem Graph $G = (Kn, Ka)$ ist eine endliche Folge $k_1, \ldots, k_m$ $(m \geq 1)$ von Knoten von G, so dass

- für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m-1$ ist $(k_i, k_{i+1}) \in Ka$
- $(k_m, k_1) \in Ka$

In anderen Worten ist ein Zyklus ein Rundgang durch den Graph über die Kanten des Graphen, bei dem die Richtung der Kanten eingehalten wird.

Definition (Baum, Wurzel, Blatt)
Sei K eine Menge. (K ist eine Referenzmenge für die Knoten. Das heißt, dass alle Knoten der Bäume, die im Folgenden definiert werden, Elemente von K sind). *Bäume* (mit Knoten in K) werden wie folgt definiert:

1. Der leere Graph $(\{\}, \{\})$ ist ein Baum (mit Knoten in K). Dieser Baum hat keine *Wurzel*.
2. Für jedes $k \in K$ ist $(\{k\}, \{\})$ ein Baum. Die *Wurzel* dieses Baums ist der Knoten k.
3. Ist $m \in \mathbb{N} \setminus \{0\}$ und ist (Kn_i, Ka_i) für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m$ ein Baum mit Wurzel k_i, so dass die Knotenmengen Kn_i paarweise disjunkt sind, und ist $k \in K$ ein „neuer“ Knoten, der in keinem Kn_i vorkommt, so ist (Kn, Ka) ein Baum (mit Knoten in K) mit

$$\begin{aligned} Kn &= \{\, k \,\} \cup \bigcup_{i=1}^{n} Kn_i \\ Ka &= \{\, (k, k_i) \mid 1 \leq i \leq m \,\} \cup \bigcup_{i=1}^{n} Ka_i \end{aligned}$$

   Die *Wurzel* dieses Baums ist der Knoten k.

Die Knoten eines Baumes, die keine Nachfolger haben, heißen *Blätter*.

Definition (Binärbaum)
Ein Binärbaum B ist ein Baum mit der Eigenschaft: Jeder Knoten von B ist entweder ein Blatt oder hat genau zwei Nachfolger.

Man kann leicht (strukturell induktiv) beweisen oder sich leicht anhand von Beispielen überzeugen, dass

- jeder nichtleere Baum eine Wurzel hat, und dass aus der Wurzel jeder andere Knoten des Baumes über die Kanten des Baumes erreicht werden kann, wobei die Richtung der Kanten eingehalten wird,
- in einem Baum kein Zyklus vorkommt und
- jeder Baum einen Pfad von seiner Wurzel zu jedem seiner Blätter enthält. Ein Wurzel-Blatt-Pfad in einem Baum heißt *Ast*.

Das Beweisprinzip der strukturellen Induktion wird in Abschnitt 8.5 ausführlicher behandelt.

Definition (Wald)
Ein Wald ist eine Menge von Bäumen, deren Knotenmengen paarweise disjunkt sind.

Bäume und Wälder werden in der Informatik häufig verwendet. Fast alle Bäume werden graphisch so dargestellt, dass ihre Wurzel oben und ihre Blätter unten sind. Eine seltene Ausnahme stellen die Beweisbäume dar (siehe Abschnitt 6.7.4).

8.4.2 Induktive Definition

Eine Definition wie die vorangehende Definition der Bäume heißt „induktive Definition". Induktive Definitionen bestehen immer aus einem oder mehreren Basisfällen (wie der Fall 2 der vorangehenden Definition) und einem oder mehreren Induktionsfällen (wie der Fall 3 der vorangehenden Definition). Zudem können sie Sonderfälle (wie der Fall 1 der vorangehenden Definition) besitzen. Die Basisfälle bestimmen Anfangsstrukturen. Die Induktionsfälle sind Aufbauregeln.[1]

8.4.3 Induktive Definition und rekursive Algorithmen

Ist eine Datenstruktur (wie im Abschnitt 8.4.1 die Datenstruktur „Baum") induktiv definiert, so lassen sich Algorithmen über dieser Datenstruktur leicht rekursiv spezifizieren. Die Anzahl der Knoten eines Baumes kann z.B. leicht wie folgt rekursiv ermittelt werden:

1. Der leere Baum hat 0 Knoten.
2. Ein Baum der Gestalt $(\{k\}, \{\})$ hat 1 Knoten.
3. Ein Baum der Gestalt (Kn, Ka) mit

$$\begin{aligned} Kn &= \{\, k \,\} \cup \bigcup_{i=1}^{n} Kn_i \\ Ka &= \{\, (k, k_i) \mid 1 \leq i \leq m \,\} \cup \bigcup_{i=1}^{n} Ka_i \end{aligned}$$

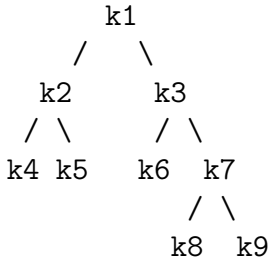
   hat einen Knoten mehr als die Summe der Knotenanzahlen der Bäume (Kn_i, Ka_i) mit $1 \leq i \leq m$.

Die Spezifikation von rekursiven Algorithmen über induktiv definierten Datenstrukturen ist eine grundlegende Technik der Informatik.

8.4.4 Darstellung von Bäumen: graphisch und durch Ausdrücke

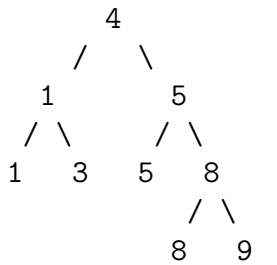
Es ist üblich, Bäume graphisch zu veranschaulichen, indem man die Knoten durch Symbole darstellt und die Kanten als Verbindungslinien dazwischen, wobei die Richtung der Kanten fast immer mit der Richtung von oben nach unten gleichgesetzt wird:

[1] Was sich aus mathematischer Sicht hinter einer induktiven Definition verbirgt, wird u.a. in der Hauptstudiumsvorlesung „Logik für Informatiker" erläutert



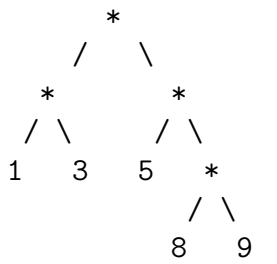
Dieser Baum ist ein Binärbaum mit Wurzel `k1` und Blättern `k4`, `k5`, `k6`, `k8`, `k9`. In vielen Fällen sind die Bezeichner der Knoten irrelevant, so dass man an Stelle von `k1`, `k2` usw. jeweils einfach einen kleinen Kreis zeichnet oder ein Symbol wie $*$.

Oft wird eine Variante von Bäumen benutzt, bei der die Knoten zusätzlich mit Werten markiert sind, zum Beispiel mit Zahlen. Dann zeichnet man einfach die Markierungen statt der Knoten oder $*$.



Die Werte, mit denen die Knoten in diesem Fall markiert sind, können mehrfach im Baum vorkommen, während die Knoten selbst nicht mehrfach vorkommen können.

In der Informatik werden auch Bäume benutzt, bei denen nur die Blätter mit Werten markiert sind, so dass sich eine Mischform zur Darstellung ergibt:



Eine andere Methode der Darstellung beruht auf Ausdrücken, wobei die Kanten des Baums durch die Verschachtelung von Teilausdrücken dargestellt werden. Der obige Baum ohne Markierungen von Knoten kann zum Beispiel so dargestellt werden:

```
Knt(Knt(Blt,
        Blt),
    Knt(Blt,
        Knt(Blt,
            Blt)))
```

Dieser Ausdruck ist mit zwei verschiedenen Symbolen gebildet, mit denen Blätter und andere Knoten unterschieden werden können. Es ist üblich, aber auch nur eine Konvention, dass die Reihenfolge der Argumente in den Ausdrücken der Reihenfolge der Nachfolger in

der graphischen Darstellung entspricht. Da man überdies die Struktur von verschachtelten Ausdrücken durch Einrückungen verdeutlicht, wirkt die Darstellung durch Ausdrücke so als sei sie durch eine Spiegelung aus der graphischen entstanden. Das wird deutlicher, wenn man die Darstellung durch Ausdrücke für die Varianten mit Markierungen betrachtet. Dazu sind lediglich zusätzliche Argumente für die Markierungen erforderlich:

```
Knt(4,
    Knt(1,
        Blt(1),
        Blt(3)),
    Knt(5,
        Blt(5),
        Knt(8,
            Blt(8),
            Blt(9))))
```

Natürlich kann man hier ebenso wie in der graphischen Darstellung auch Varianten betrachten, in denen nur die Blätter mit Werten markiert sind.

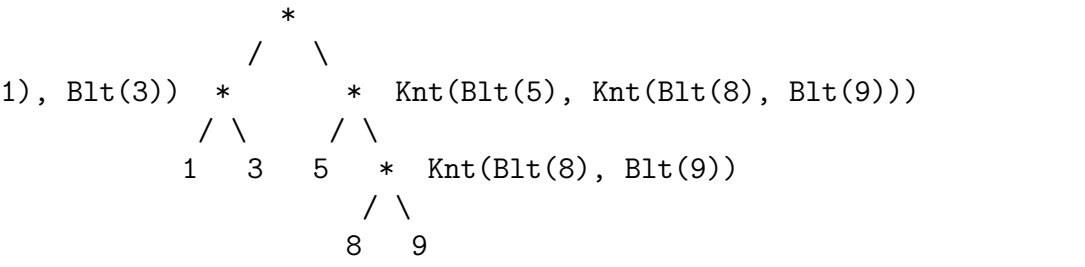
8.4.5 Rekursive Typen zum Ausdrücken von induktiven Definitionen — Der Binärbaum

In SML können nichtleere Binärbäume, deren Blätter mit ganzen Zahlen markiert sind, wie folgt spezifiziert werden:

```
- datatype binbaum1 = Blt of int                   (* Blt f"ur Blatt  *)
                    | Knt of binbaum1 * binbaum1; (* Knt f"ur Knoten *)
datatype binbaum1 = Blt of int
                  | Knt of binbaum1 * binbaum1

- val b = Knt(Knt(Blt(1), Blt(3)),
              Knt(Blt(5), Knt(Blt(8), Blt(9))));
val b = Knt (Knt (Blt #,Blt #),
             Knt (Blt #,Knt #)) : binbaum1
```

Man beachte, dass der Wert von `b` abgekürzt gedruckt wird. Die Abkürzung betrifft lediglich die interaktive Treiberschleife von SML. In der globalen Umgebung ist der richtige Wert abgespeichert.

Dieser Binärbaum kann wie folgt graphisch dargestellt werden:



Der (Wert-)Konstruktor Knt bildet aus zwei Binärbäumen einen neuen Binärbaum. Sein Typ ist also `binbaum1 * binbaum1 -> binbaum1`.

Ein solcher Binärbaum wird „beblättert“ genannt, weil seine Blätter und nur seine Blätter Markierungen (d.h. Werte) tragen. Im Abschnitt 8.6 wird eine andere Art von Binärbäumen eingeführt, der Binärbaum mit Knotenmarkierungen, deren Knoten alle Markierungen (d.h. Werte) tragen.

Die obige Typdefinition der Binärbäume hat genau dieselbe Struktur wie die Definition der Bäume in Abschnitt 8.4.1. Inhaltlich unterscheidet sie sich von der dort gegebenen Definition lediglich in der zusätzlichen Einschränkung, dass jeder Knoten, der kein Blatt ist, genau zwei Nachfolger haben muss. Syntaktisch unterscheidet sie sich von der Definition in Abschnitt 8.4.1 in der Verwendung von SML-Konstrukten. Eine Typdefinition wie die Definition des Typs `binbaum1` ist also eine induktive Definition.

Wegen ihrer syntaktischen Ähnlichkeit mit rekursiven Algorithmen werden aber solche Typdefinition in der Informatik manchmal „rekursive Typdefinitionen“ genannt.

Die Typen, die mittels induktiven (oder rekursiven) Typdefinitionen vereinbart werden, heißen „rekursive Typen“.

Die Funktion `blaetterzahl` zur Berechnung der Anzahl der Blätter eines Binärbaums vom Typ `binbaum1` kann in SML wie folgt implementiert werden:

```
- fun blaetterzahl  (Blt(_))      = 1
    | blaetterzahl (Knt(b1, b2)) =   blaetterzahl b1
                                   + blaetterzahl b2;
val blaetterzahl = fn : binbaum1 -> int

- val c = Knt(Blt(1), Blt(2));

- blaetterzahl(c);
val it = 2 : int

- blaetterzahl(Knt(c, Knt(c, c)));
val it = 6 : int
```

8.4.6 Polymorphe Typen

Es ist oft vorteilhaft, rekursive Typen polymorph zu definieren. Ein polymorpher Typ `binbaum1` kann wie folgt definiert werden:

```
- datatype 'a binbaum2 = Blt of 'a
                       | Knt of 'a binbaum2 * 'a binbaum2;
datatype 'a binbaum2 = Blt of 'a
                     | Knt of 'a binbaum2 * 'a binbaum2
```

Die Funktion `blaetterzahl` kann nun wie folgt als polymorphe Funktion redefiniert werden:

```
- fun blaetterzahl  (Blt(_))      = 1
    | blaetterzahl (Knt(b1, b2)) =   blaetterzahl b1
                                   + blaetterzahl b2;
```

```
val blaetterzahl = fn : 'a binbaum2 -> int

- val d = Knt(Blt("ab"), Blt("cd"));
val d = Knt (Blt "ab",Blt "cd") : string binbaum2

- blaetterzahl(d);
val it = 2 : int

- let val e = Knt(d, d);
      val f = Knt(e, e)
  in
      blaetterzahl(Knt(f, f))
  end;
val it = 16 : int

- val g = Knt(Blt([1,2,3]), Blt[4,5]);
val g = Knt (Blt [1,2,3],Blt [4,5]) : int list binbaum2

- let val h = Knt(g, g);
      val i = Knt(h, h)
  in
      blaetterzahl(Knt(i, i))
  end;
val it = 16 : int
```

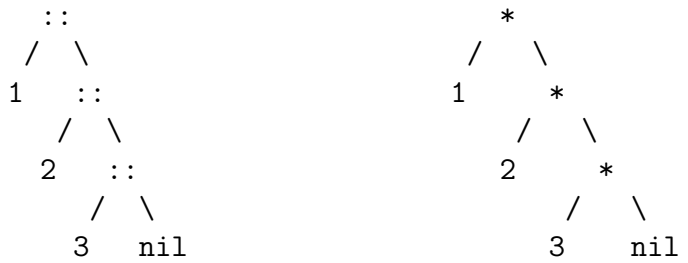
8.4.7 Suche in Binärbäumen

Mit dem folgenden Prädikat kann überprüft werden, ob ein Element in einem Binärbaum vorkommt:

```
- fun suche(x, Blt(M))      = (x = M)
    | suche(x, Knt(B1, B2)) = suche(x, B1)
                              orelse suche(x, B2);
val suche = fn : 'a * binbaum2 -> bool

- val b = Knt(Knt(Blt(1), Blt(3)), Knt(Blt(5), Knt(Blt(8), Blt(9))));
val b = Knt (Knt (Blt #,Blt #),Knt (Blt #,Knt #)) : int binbaum2

- suche(5, b);
val it = true : bool

- suche(12, b);
val it = false : bool
```

8.4.8 Die Liste als Binärbaum

Die Liste ist ein Sonderfall des Binärbaums, wie der Vergleich der folgenden Darstellungen erkennen lässt:



In der Tat besteht die Suche in einem Binärbaum einer solchen Gestalt aus denselben Schritten wie die Suche in einer Liste mit der Funktion `member`.

8.5 Beweisprinzip der strukturellen Induktion

Es stellt sich die Frage, wie Eigenschaften von Funktionen bewiesen werden können, die auf rekursiven Typen definiert sind. Es ist bemerkenswert, dass Induktionsbeweise (siehe Abschnitt 1.1.8) und induktive Definitionen eine ähnliche Gestalt haben:

- Induktionsbeweise und induktive Definitionen weisen Basisfälle und Induktionsfälle auf.
- Sowohl bei Induktionsbeweisen als auch bei induktiven Definitionen stellen die Induktionsfälle eine Art stufenweisen Aufbau dar.

In der Tat lassen sich Eigenschaften von Funktionen, die auf rekursiven Typen definiert sind, oft induktiv beweisen. Das Beweisprinzip der vollständigen Induktion bezieht sich auf natürliche Zahlen. Das Beweisprinzip, das hier anzuwenden ist, heißt „strukturelle Induktion":

Beweis:

Sei `t` ein rekursiver Typ mit den (Wert-)Konstruktoren $\mathtt{k}_i^{s_i}$ $(0 \leq i \leq I)$ wobei s_i die Stelligkeit des (Wert-)Konstruktors $\mathtt{k}_i^{s_i}$ ist.

Um zu zeigen, dass alle Werte des Typs `t` eine Eigenschaft **E** (wie etwa „die Auswertung einer Anwendung der Funktion f auf den Wert terminiert") besitzen, genügt es zu zeigen:

1. Basisfälle: Jeder 0-stellige (Wert-)Konstruktor $\mathtt{k}_i^0$ $(0 \leq i \leq I)$ besitzt die Eigenschaft **E**.
2. Induktionsfälle:

   Für jeden (Wert-)Konstruktor $\mathtt{k}_i^{s_i}$ (der Stelligkeit s_i) mit $i \geq 1$ gilt: wenn s_i Werte $W_1, \ldots, W_{s_i}$ des Typs `t` die Eigenschaft **E** besitzen (Induktionsannahme), dann besitzt auch der Wert $\mathtt{k}_i^{s_i}(W_1, \ldots, W_{s_i})$ des Typs `t` die Eigenschaft **E**.

qed.

Das Beweisprinzip der strukturellen Induktion lässt sich auf die vollständige Induktion zurückführen. Dies ist aber keineswegs unmittelbar.[2]

[2] siehe die Hauptstudiumsvorlesung „Logik für Informatiker"

Als Beispiel einer Anwendung des Beweisprinzips der strukturellen Induktion zeigen wir, dass jede Anwendung der polymorphen Funktion `blaetterzahl` auf einen Binärbaum vom Typ `binbaum1` terminiert:

Beweis:

Basisfall:

Ist `A` ein Ausdruck der Gestalt `Blt(x)`, so führt die Auswertung von `blaetterzahl(A)` nach Definition (der Funktion `blaetterzahl`) zur Auswertung von `1`, was offenbar terminiert.

Induktionsfall:

Seien `W1` und `W2` zwei Werte vom Typ `binbaum1`.

Induktionsannahme:
Sei angenommen, dass die Auswertung von `blaetterzahl(W1)` und von `blaetterzahl(W2)` jeweils terminiert.

Nach Definition der Funktion führt die Auswertung von `blaetterzahl(Knt(W1, W2))` zur Auswertung von `blaetterzahl(W1) + blaetterzahl(W2)`. Nach Induktionsannahme terminiert die Auswertung der beiden Komponenten dieser Addition. Folglich terminiert auch die Auswertung dieses Ausdrucks.

qed.

8.6 Beispiele: Implementierungen des Konzepts der „Menge“

In diesem Abschnitt wird untersucht, wie der mathematische Begriff „Menge“, der zur Lösung vieler praktischer Aufgaben nützlich ist, in einer Programmiersprache wiedergegeben werden kann.

Zunächst werden die Begriffe „Menge“ und „Datenstruktur“ erläutert.

8.6.1 Was ist eine „Menge“

Die „Menge“ ist ein grundlegender Begriff der Mathematik zur Zusammensetzung von Objekten. Die Zusammensetzung von Objekten als Menge bringt weder eine Reihenfolge noch irgendeine sonstige Strukturierung der Objekte mit sich. Die Objekte, die eine Menge zusammenfasst, werden „Elemente“ dieser Menge genannt.

Referenzmenge

Eine Menge wird immer bezüglich einer „Referenzmenge“ definiert, d.h. einer „Urmenge“, woraus die Elemente der zu definierenden Mengen stammen. Der Verzicht auf eine Referenzmenge würde Paradoxien ermöglichen, wie etwa das folgende Paradoxon:

Sei `M` die (Pseudo-)Menge, die alle Mengen umfasst, die nicht Element von sich selbst sind: Gilt $M \in M$?

Falls ja, dann gilt nach Definition von `M`: $M \notin M$, ein Widerspruch.

Falls nein, dann gilt nach Definition von `M`: $M \in M$, ein Widerspruch.

Die Bedingung, dass keine Menge `M` definiert werden kann, ohne eine Referenzmenge festzulegen, schließt einen solchen Fall aus.

In einer typisierten Programmiersprache stellen die Typen geeignete Kandidaten für etwaige Referenzmengen dar.

Extensional und intensional definierte Mengen

Eine Menge wird „extensional“ (oder „explizit“) definiert, wenn ihre Definition aus einer Auflistung ihrer Elemente besteht. So ist z.B. $\{1.0, 23.5, 12.45\}$ eine extensional definierte Menge, deren Referenzmenge $\mathbb{R}$ ist.

Eine Menge wird „intensional“ (oder „implizit“) definiert, wenn ihre Elemente durch eine Eigenschaft charakterisiert werden. $\{ x * x \,|\, x \in \mathbb{N} \}$ ist z.B. eine intensional definierte Menge, deren Referenzmenge $\mathbb{N}$ ist.

Funktionen sind in einer Programmiersprache das Gegenstück zu intensional definierten Mengen. Die Funktion

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

drückt in SML die Menge $\{ x * x \,|\, x \in \mathbb{Z} \}$ als Menge möglicher Ergebniswerte aus.

Die Datenstruktur „Menge“, die es zu implementieren gilt, kann sinnvollerweise also nur extensional definierte, zudem endliche Mengen wiedergeben.

Mengenoperationen

Mit dem Begriff „Menge“ werden die folgenden grundlegenden Operationen definiert:

Elementbeziehung:	$\in$
Vereinigung:	$M1 \cup M2 = \{ x \,|\, x \in M1 \vee x \in M2 \}$
Durchschnitt:	$M1 \cap M2 = \{ x \,|\, x \in M1 \wedge x \in M2 \}$
Gleichheit:	Zwei Mengen sind gleich, wenn sie genau dieselben Elemente haben.
Teilmengenbeziehung:	$M1$ ist eine Teilmenge von $M2$ ($M1 \subseteq M2$), wenn jedes Element von $M1$ auch Element von $M2$ ist.

Zudem ist die „leere Menge“ eine ausgezeichnete Menge, die keine Elemente hat.

8.6.2 Was ist eine „Datenstruktur”?

Unter „Datenstruktur“ versteht man in der Informatik

- eine Darstellung einer mathematischen Struktur (wie z.B. Mengen, Vektoren oder Listen) in einer Programmiersprache zusammen mit
- der Implementierung der grundlegenden Operationen dieser Struktur in derselben Programmiersprache basierend auf dieser Darstellung.

In einer typisierten Programmiersprache wie SML ist die Definition eines Typs ein gewöhnlicher Teil der Implementierung einer Datenstruktur. Die Definition eines Typs allein reicht in der Regel nicht aus, weil damit die grundlegenden Operationen der betrachteten Struktur noch lange nicht gegeben sind.

In der Praxis besteht die Implementierung einer Datenstruktur typischerweise aus einem Typ und aus Prozeduren (die nicht immer Funktionen sind), die sich auf diesen Typ beziehen (siehe auch Kapitel 11 zum Einsatz abstrakter Datentypen).

8.6.3 Die Menge als charakteristische Funktion

In Abschnitt 8.2.5 wurde die Menge der Ziffern wie folgt implementiert:

```
- type 'a menge = 'a -> bool;
type 'a menge = 'a -> bool

- val ziffer_menge : int menge =
     fn 0 => true
      | 1 => true
      | 2 => true
      | 3 => true
      | 4 => true
      | 5 => true
      | 6 => true
      | 7 => true
      | 8 => true
      | 9 => true
      | _ => false;
val ziffer_menge = fn : int menge
```

Eine solche Funktion nennt man charakteristische Funktion der Menge (aller Ziffern).

Diese Implementierung gibt die Elementbeziehung unmittelbar wieder. Die Vereinigung und der Durchschnitt lassen sich sehr einfach wie folgt realisieren:

```
- fun vereinigung(m1:'a menge, m2:'a menge) =
                              fn x => m1(x) orelse m2(x);
val vereinigung = fn : 'a menge * 'a menge -> 'a -> bool

- fun durchschnitt(m1:'a menge, m2:'a menge) =
                              fn x => m1(x) andalso m2(x);
val durchschnitt = fn : 'a menge * 'a menge -> 'a -> bool

- val M1 : string menge =
     fn "ab" => true
      | "bc" => true
      | "be" => true
      | _    => false;
val M1 = fn : string menge
```

```
- val M2 : string menge =
     fn "de" => true
      | "fg" => true
      | "be" => true
      | _    => false;
val M2 = fn : string menge

- vereinigung(M1, M2);
val it = fn : string -> bool

- vereinigung(M1, M2)("be");
val it = true : bool

- durchschnitt(M1, M2)("ab");
val it = false : bool
```

Diese Implementierung ist aber zur Implementierung der Gleichheit (von Mengen) und der Teilmengenbeziehung wenig geeignet. Viel geeigneter wäre eine Darstellung, die es ermöglicht, die Auflistung der Elemente beider Mengen, die auf Gleichheit oder Teilmengenbeziehung untersucht werden sollen, direkt zu vergleichen. Da die Auflistung im Programm selbst statt in einem Aufrufparameter vorkommt, ist ein solcher Vergleich in der obigen Implementierung nicht einfach.

8.6.4 Die Menge als Liste

Es bietet sich also an, eine extensional definierte, endliche Menge als Liste darzustellen.

Die Elementbeziehung wird durch das im Abschnitt 6.5.7 eingeführte polymorphe Prädikat `member` implementiert:

```
- fun member(x, nil)         = false
    | member(x, head::tail)  = if x = head
                               then true
                               else member(x, tail);
val member = fn : ''a * ''a list -> bool

- member(3,[1,2,3,4]);
val it = true : bool
```

Der Zeitbedarf einer Überprüfung einer Elementbeziehung kann wie folgt geschätzt werden. Als Schätzung der Problemgröße bietet sich die Größe (Kardinalität) der Menge an. Als Zeiteinheit bietet sich die Anzahl der rekursiven Aufrufe des Prädikats `member` an. Zur Überprüfung einer Elementbeziehung bezüglich einer Menge der Kardinalität n wird man bestenfalls `member` einmal, schlechtestenfalls $(n+1)$–mal aufrufen. Der Zeitbedarf einer Überprüfung einer Elementbeziehung ist also schlechtestenfalls $O(n)$.

Die Vereinigung wird durch die im Abschnitt 5.5.7 eingeführte polymorphe Funktion `append` (in SML auch als vordefinierte Infixfunktion „`@`“ vorhanden) implementiert:

```
- fun append(nil,    L) = L
    | append(h :: t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list

- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

Diese Implementierung der Vereinigung mag für manche Anwendungen unbefriedigend sein, weil sie die Wiederholung von Elementen nicht ausschließt.

Der Durchschnitt kann wie folgt implementiert werden:

```
- fun durchschnitt(nil,    _) = nil
    | durchschnitt(h :: t, L) = if member(h, L)
                                then h :: durchschnitt(t, L)
                                else durchschnitt(t, L);
val durchschnitt = fn : ''a list * ''a list -> ''a list

- durchschnitt([1,2,3,4], [3,4,5,6]);
val it = [3,4] : int list
```

Die Teilmengenbeziehung kann wie folgt implementiert werden:

```
- fun teilmenge(nil,    _) = true
    | teilmenge(h :: t, L) = if member(h, L)
                             then teilmenge(t, L)
                             else false;
val teilmenge = fn : ''a list * ''a list -> bool

- teilmenge([6,4,2],[1,2,8,4,9,6,73,5]);
val it = true : bool

- teilmenge([4,2,3,1], [3,6,5,4]);
val it = false : bool
```

Um die Menge zu verändern, stehen die Funktionen `cons` (in SML als Infix-Operator `::` vordefiniert), `head` (in SML als `hd` vordefiniert) und `tail` (in SML als `tl` vordefiniert) zur Verfügung (siehe Abschnitte 5.5.5 und 5.5.6).

8.6.5 Die Menge als sortierte Liste

Die im vorangehenden Abschnitt eingeführte Implementierung der Menge setzt keineswegs voraus, dass die Listenelemente in auf- oder absteigender Reihenfolge sortiert sind. Eine Sortierung der Elemente setzt selbstverständlich voraus, dass eine Ordnung (d.h. eine reflexive, transitive und antisymmetrische Relation) über der Referenzmenge vorhanden ist, die obendrein total ist, das heißt, dass je zwei beliebige Elemente in der einen oder anderen Richtung zueinander in der Relation stehen. Dies wird im Folgenden angenommen.

Die Frage stellt sich, ob eine Sortierung der Elemente (nach der Ordnung der Referenzmenge) von Vorteil wäre.

Das ist der Fall. Sind die Elemente nach aufsteigenden (bzw. absteigenden) Werten sortiert, so kann die sequenzielle Suche durch die Liste, die das Prädikat `member` durchführt, abgebrochen werden, sobald ein Listenelement gefunden wird, das größer (bzw. kleiner) als das gesuchte Element ist.

Unter der Annahme, dass die Listenelemente nach aufsteigenden Werten sortiert sind, kann die verbesserte Suche wie folgt implementiert werden:

```
- fun member_sort(x, nil)    = false
    | member_sort(x, h::t)  =
       if x < h
       then false
       else if x = h
            then true
            else member_sort(x, t);
val member_sort = fn : int * int list -> bool
```

Schlechtestenfalls wird mit `member_sort` und einer sortierten Liste sowie `member` und einer beliebigen Liste die Liste ganz durchlaufen. Schlechtestenfalls benötigt die Suche mit `member_sort` und einer sortierten Liste eine Zeit, die $O(n)$ ist, wenn n die Länge der Liste ist.

Eine weitere Frage stellt sich: Ermöglicht eine Sortierung der Elemente eine effizientere Suche als die sequenzielle Suche mit `member_sort`?

Die Antwort dazu liefert ein Beispiel aus dem Alltag. Wenn man im Telefonbuch nach der Telefonnummer eines Herrn Zimmermann sucht, schlägt man es am Ende auf. Wenn man aber die Telefonnummer eines Herrn Klein erfahren möchte, dann schlägt man das Telefonbuch in der Mitte auf. Und sicherlich wird man das Telefonbuch in seinen Anfangsseiten aufschlagen, wenn man die Telefonnummer einer Frau Ackermann erfahren möchte.

Da der Bereich der möglichen Namen bekannt ist — alle Namen fangen mit einem Buchstaben an, der zwischen A und Z liegt — und die Verteilung der Namen uns einigermaßen vertraut ist — z.B. fangen deutsche Familiennamen viel häufiger mit K oder S als mit I, N oder O an — kann man ziemlich schnell in die Nähe des gesuchten Namen kommen.

Es liegt also nahe anzunehmen, dass die Überprüfung der Elementbeziehung in einer sortierten Liste schneller erfolgen kann als in einer unsortierten Liste.

Diese Überlegung ist aber inkorrekt, weil — sortiert sowie unsortiert — eine Liste immer nur linear von ihrem ersten (am weitesten links stehenden) Element an durchlaufen werden kann. Die Darstellung der endlichen, extensional definierten Menge als sortierte Liste würde also schlechtestenfalls (wie etwa im Falle der Suche nach dem Namen Zimmermann im Telefonbuch) keinen Vorteil gegenüber der Darstellung als unsortierte Liste bringen. Die Verwendung von sortierten Listen würde sogar einen großen Nachteil mit sich bringen: Den Aufwand für die Sortierung der Elemente.

8.6.6 Die Menge als binärer Suchbaum

Prinzip

Betrachten wir wieder das Beispiel der Suche nach einem Namen im Telefonbuch. Verallgemeinern wir zunächst das Beispiel und nehmen wir an, dass der vom Telefonbuch

abgedeckte Namensbereich unbekannt ist, d.h., dass es z.B. nicht sicher ist, dass es für jeden möglichen Anfangsbuchstaben überhaupt Namen gibt. Nehmen wir zudem an, dass auch die Namensverteilung völlig beliebig sein kann. Der Verständlichkeit halber kann man auch annehmen, dass das Buch statt Namen (und Telefonnummern) lediglich ganze (d.h. negative sowie positive) Zahlen enthält, die beliebig klein oder gross sein können.

Ohne jegliche Auskunft über den Zahlenbereich, den das Buch abdeckt, und über die Zahlenverteilung im Buch, sucht man nach einer gegebenen Zahl am schnellsten dadurch, dass man das Buch (ungefähr) in seiner Mitte aufschlägt und das Verfahren im linken oder rechten Buchteil (rekursiv) wiederholt, je nach dem, ob die Zahl in der Mitte des Buches größer oder kleiner als die gesuchte Zahl ist.

Nichtleerer binärer Suchbaum

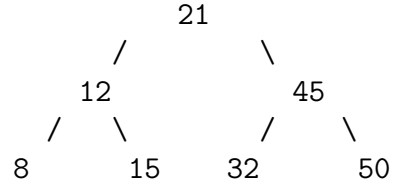
Eine solche Suche lässt sich anhand von nichtleeren Binärbäumen implementieren, wenn nicht nur die Blätter Zahlen (oder sonstige Werte) beinhalten (oder damit „markiert“ sind), sondern auch die sonstigen Knoten:

```
- datatype binbaum3 = Blt of int
                    | Knt of binbaum3 * int * binbaum3;
datatype binbaum3 = Blt of int
             | Knt of binbaum3 * int * binbaum3
```

Man beachte, dass dieser Datentyp die leere Menge ausschließt. Dies mag für manche Anwendungen ungeeignet sein.

```
- val b1 = Knt(Blt(8),12,Blt(15));
val b1 = Knt (Blt 8,12,Blt 15) : binbaum3

- val b2 = Knt(Blt(32),45,Blt(50));
val b2 = Knt (Blt 32,45,Blt 50) : binbaum3

- val b3 = Knt(b1,21,b2);
val b3 = Knt (Knt (Blt #,12,Blt #),21,Knt (Blt #,45,Blt #)) : binbaum3
```

Das Zeichen `#` wird in der gedruckten Mitteilung verwendet, um diese zu kürzen. Diese Kürzung betrifft nur die gedruckte Mitteilung und selbstverständlich nicht den in der Umgebung gespeicherten Wert des Namens `b3`. Dieser Wert von `b3` kann auch wie folgt dargestellt werden:



In diesem Baum gilt für jeden Knoten, dass alle Markierungen im linken Teilbaum kleiner sind als die Markierung des Knotens und alle Markierungen im rechten Teilbaum größer.

Verbesserung des „nichtleeren binären Suchbaums“

Binärbäume vom Typ `binbaum3` haben den Nachteil, nicht alle endlichen Mengen sortiert darstellen zu können. Zum Beispiel können die Mengen $\{1, 2\}$ und $\{1, 2, 3, 4\}$ nicht als Binärbaum vom Typ `binbaum3` dargestellt werden.

In der Tat kann man unter Anwendung der strukturellen Induktion beweisen, dass jeder Binärbaum vom Typ `binbaum3` eine ungerade Anzahl von Knotenmarkierungen hat:

Beweis:

Basisfall:

Ein Binärbaum der Gestalt `Blt(W)` hat genau eine Knotenmarkierung, also eine ungerade Anzahl von Knotenmarkierungen.

Induktionsfall:

Seien `B1` und `B2` zwei Binärbäume vom Typ `binbaum3` und `W` eine ganze Zahl.

Induktionsannahme: `B1` und `B2` haben jeweils eine ungerade Anzahl von Knotenmarkierungen.

Der Binärbaum `Knt(B1, W, B2)` hat $k = |\texttt{B1}| + 1 + |\texttt{B2}|$ Knotenmarkierungen. Da $|\texttt{B1}|$ ungerade ist, gibt es $n_1 \in \mathbb{N}$ mit $|\texttt{B1}| = 2 * n_1 + 1$. Da $|\texttt{B2}|$ ungerade ist, gibt es $n_2 \in \mathbb{N}$ mit $|\texttt{B2}| = 2*n_2+1$. Also $k = (2*n_1+1)+1+(2*n_2+1) = 2*(n_1+n_2+1)+1$, d.h. k ist ungerade.

qed.

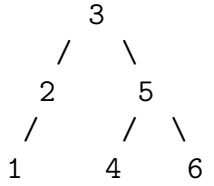
Der folgende Typ `binbaum4` beseitigt den Mangel der Binärbäume vom Typ `binbaum3`.

```
- datatype binbaum4 = Knt1 of int
                    | Knt2 of int * int
                    | Knt3 of binbaum4 * int * binbaum4;
datatype binbaum4
     = Knt1 of int | Knt2 of int * int
               | Knt3 of binbaum4 * int * binbaum4

- val c0 = Knt2(1,2);
val b2 = Knt2 (1,2) : binbaum4

- val c1 = Knt3(Knt1(1), 2, Knt2(3,4));
val c1 = Knt3 (Knt1 1,2,Knt2 (3,4)) : binbaum4

- val c2 = Knt3(Knt2(1,2), 3, Knt1(4));
val c2 = Knt3 (Knt2 (1,2),3,Knt1 4) : binbaum4

- val d = Knt3(Knt2(1,2), 3, Knt3(Knt1(4), 5, Knt1(6)));
val d = Knt3 (Knt2 (1,2),3,Knt3 (Knt1 #,5,Knt1 #)) : binbaum4
```

Die Binärbäume `c1` und `c2` können wie folgt dargestellt werden:

`c1` und `c2` sind die zwei Möglichkeiten, die Menge $\{1, 2, 3, 4\}$ als binärer Suchbaum vom Typ `binbaum4` darzustellen.

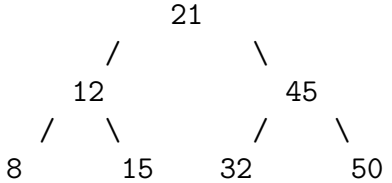
Der Binärbaum `d` kann wie folgt dargestellt werden:



Suche in einem binären Suchbaum vom Typ `binbaum4`

Betrachten wir die folgenden binären Suchbäume vom Typ `binbaum4`:

```
- val b1 = Knt3(Knt1(8), 12, Knt1(15));
val b1 = Knt3 (Knt1 8,12,Knt1 15) : binbaum4

- val b2 = Knt3(Knt1(32), 45, Knt1(50));
val b2 = Knt3 (Knt1 32,45,Knt1 50) : binbaum4

- val b3 = Knt3(b1, 21, b2);
val b3 = Knt3 (Knt3 (Knt1 #,12,Knt1 #),21,
             Knt3 (Knt1 #,45,Knt1 #)) : binbaum4
```



Die Suche nach `25` (bzw. nach `32`) in `b3` kann wie folgt durchgeführt werden:

1. Da $25 > 21$ (bzw. $32 > 21$) wird die Suche im rechten Teilbaum `b2` von `b3` fortgesetzt.
2. Da $25 < 45$ (bzw. $32 < 45$) ist, wird die Suche im linken Teilbaum `Knt1(32)` von `b2` fortgesetzt.
3. Da $25 \neq 32$ ist, terminiert die Suche erfolglos (bzw. da $32 = 32$ ist, terminiert die Suche erfolgreich).

Das Verfahren macht nur dann Sinn, wenn die Knotenmarkierungen so sortiert sind, dass für jeden Haupt- oder Teilbaum der Gestalt `Knt2(Markierung1, Markierung2)` gilt:

```
Markierung1 < Markierung2
```

und für jeden Haupt- oder Teilbaum der Gestalt `Knt3(LBaum, Markierung, RBaum)` für jeden Knoten `Kl` im linken Teilbaum `LBaum` und für jeden Knoten `Kr` im rechten Teilbaum `RBaum` gilt:

`Kl` $<$ `Markierung` $<$ `Kr`

Diese Suche in binären Suchbäumen vom Typ `binbaum4` kann wie folgt implementiert werden:

```
- fun suche(x, Knt1(M))               = (x = M)
    | suche(x, Knt2(M1, M2))          = (x = M1) orelse (x = M2)
    | suche(x, Knt3(LBaum, M, RBaum)) =
               if x = M
               then true
               else if x < M
                    then suche(x, LBaum)
                    else suche(x, RBaum);
val suche = fn : int * binbaum4 -> bool

- suche(25, b3);
val it = false : bool

- suche(32, b3);
val it = true : bool

- suche(4, d);
val it = true : bool
```

Zeitbedarf der Suche in einem binären Suchbaum

Der Zeitbedarf der Suche nach einer Zahl in einem binären Suchbaum vom Typ `binbaum4` kann wie folgt geschätzt werden.

Als Problemgröße bietet sich die Anzahl der Knotenmarkierungen an, d.h. die Kardinalität der Menge, die der binäre Suchbaum darstellt.

Als Zeiteinheit bietet sich die Anzahl der Knoten an, die besucht werden, bis eine Antwort geliefert wird.

Die Suche nach einer Zahl in einem binären Suchbaum wird also die folgenden Zeiten in Anspruch nehmen:

- 1 Zeiteinheit, wenn der Binärbaum die Gestalt `Knt1(M)` hat (d.h. aus einem Blatt besteht),
- 2 Zeiteinheiten, wenn der Binärbaum die Gestalt `Knt2(M1, M2)` hat (d.h. nur eine einzige Kante enthält);
- höchstens die Anzahl der Knoten entlang eines längsten Astes des Baumes.

Die Ausgeglichenheit ist wünschenswert

Offenbar wird eine Suche in einem binären Suchbaum die besten Zeiten haben, wenn die Äste des Baumes alle gleich lang sind. Längenunterschiede von einem Knoten zwischen zwei Ästen eines Binärbaumes von einem Knoten können offenbar nicht vermieden werden, sonst hätten alle nichtleeren Binärbäume eine ungerade Anzahl an Markierungen (d.h. an Knoten).

Definition (Ausgeglichener Baum)
Ein Binärbaum vom Typ `binbaum4` heißt „ausgeglichen“, wenn sich die Längen zweier beliebiger Äste dieses Baumes um höchstens 1 unterscheiden.

Definition (Tiefe)
Ein Knoten `k` eines Baumes hat die Tiefe t, wenn der Pfad von der Wurzel nach `k` in dem Baum t Knoten enthält.
Die Tiefe eines Baumes ist die maximale Tiefe eines seiner Knoten.

Satz A
Die Länge eines Astes eines ausgeglichenen Binärbaums vom Typ `binbaum4`, der n Knoten hat, ist $O(\ln n)$.

Aus dem Satz A folgt, dass die Suche in einem ausgeglichenen Binärbaum die Zeitkomplexität $O(\ln n)$ hat, wenn n die Kardinalität der Menge ist.
Der Satz A folgt aus der folgenden Beobachtung:

Satz B
Ein ausgeglichener Binärbaum der Tiefe t kann bis zu 2^t Knoten mit Tiefe t haben.

Beweis:

Der Satz B wird wie folgt induktiv bewiesen:

Basisfall: Die Aussage gilt für Binärbäume der Tiefe 0 und 1.

Induktionsfall: Sei $t \in \mathbb{N}$.

Induktionsannahme: Ein ausgeglichener Baum der Tiefe t habe bis zu 2^t Knoten mit Tiefe t.

Ein Binärbaum der Tiefe $t+1$ besteht aus einer Wurzel mit zwei Nachfolgern, die die Wurzeln von Teilbäumen der Tiefe t sind. Jeder dieser Teilbäume kann nach Induktionsannahme bis zu 2^t Knoten mit Tiefe t in dem jeweiligen Teilbaum haben. Diese Knoten sind genau die Knoten mit Tiefe $t+1$ im gesamten Baum, zusammen sind es also bis zu $2 * (2^t) = 2^{t+1}$ Knoten.

qed.

Beweis:

Beweis des Satzes A:

Aus dem Satz B folgt, dass ein ausgeglichener Binärbaum der Tiefe t maximal $2^0 + 2^1 + \ldots + 2^t = 2^{t+1} - 1$ Knoten hat. Hat also ein ausgeglichener Binärbaum n Knoten, so gilt:

$$2^t - 1 \leq n \leq 2^{t+1} - 1$$

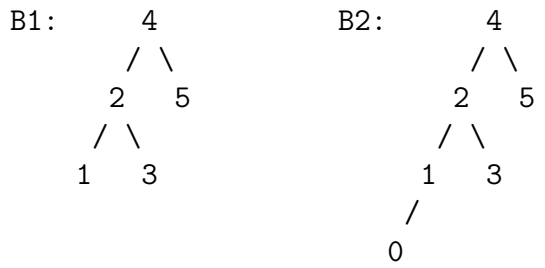
wobei t die Tiefe des Baumes ist. Daraus folgt:

$$t \leq \log_2(n+1) \leq t + 1$$

Das heißt: $t \in O(\log_2 n + 1) = O(K * \ln n)$ für ein $K \in \mathbb{N}$, d.h. $t \in O(\ln n)$.
qed.

Preis der Ausgeglichenheit

Wird die Menge durch Hinzufügen oder Löschen von Elementen verändert, so kann nach und nach der Binärbaum, der diese Menge darstellt, seine Ausgeglichenheit verlieren. Als Beispiel dafür stellt im Folgenden der Baum `B1` einen ausgeglichenen Binärbaum dar. Nach Einfügen des Wertes 0 entsteht der Baum `B2`, der offensichtlich nicht mehr ausgeglichen ist:



Ein Verfahren ist also nötig, um die Ausgeglichenheit nach jeder oder nach einigen Änderungen der Menge wiederherzustellen. Der Algorithmus dazu ist nicht trivial.[3]

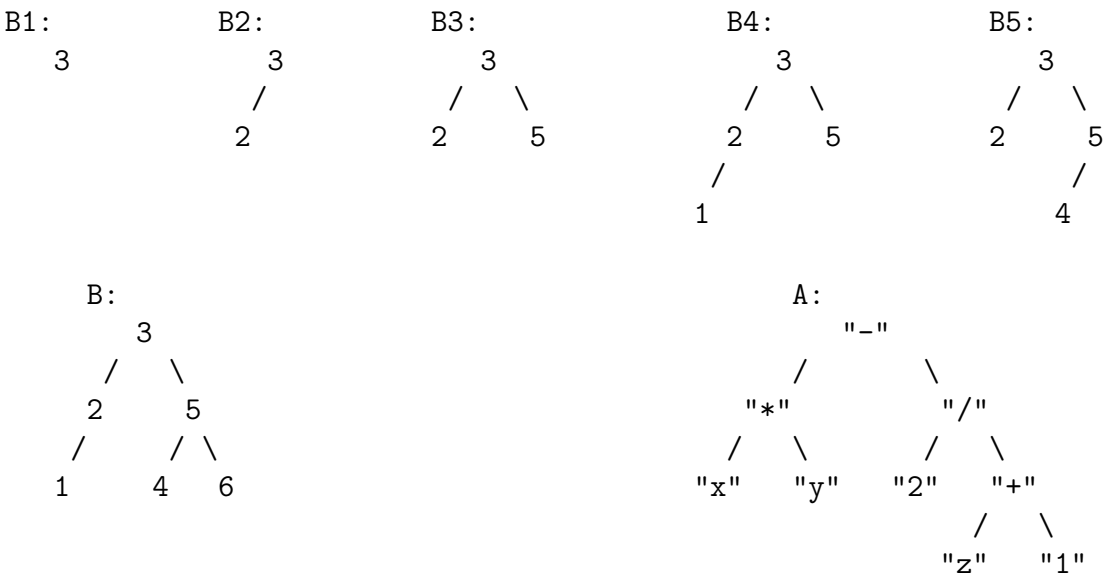
8.7 Beispiele: Grundlegende Funktionen für binäre (Such-)Bäume

Die Funktionen dieses Abschnitts beziehen sich auf den folgenden polymorphen Typ „Binärbaum mit Knotenmarkierungen":

```
- datatype 'a binbaum5 = Leer
                       | Knt1 of               'a
                       | Knt2 of 'a          * 'a
                       | Knt3 of 'a binbaum5 * 'a * 'a binbaum5;
```

[3] Hierauf wird in den Grundstudiumsvorlesungen „Informatik 2" und „Effiziente Algorithmen" ausführlicher eingegangen.

Wir nehmen an, dass in einem Ausdruck `Knt2(M1,M2)` der Knoten mit Markierung `M1` als linker Nachfolger des Knotens mit Markierung `M2` aufgefasst werden soll. Betrachten wir die Binärbäume mit folgenden graphischen Darstellungen.



Der Baum `A` soll offenbar einen arithmetischen Ausdruck repräsentieren. Diese Binärbäume können wie folgt als Ausdrücke des Typs `int binbaum5` bzw. `string binbaum5` im Fall von `A` dargestellt werden:

```
- val B1 = Knt1(           3);
- val B2 = Knt2(2,         3);
- val B3 = Knt3(Knt1(2),   3, Knt1(5));
- val B4 = Knt3(Knt2(1,2), 3, Knt1(5));
- val B5 = Knt3(Knt1(2),   3, Knt2(4,5));
- val B  = Knt3(Knt2(1,2), 3, Knt3(Knt1(4), 5, Knt1(6)));

- val A = Knt3( Knt3( Knt1("x"),
                      "*",
                      Knt1("y")
                    ),
                "-",
                Knt3( Knt1("2"),
                      "/",
                      Knt3( Knt1("z"),
                            "+",
                            Knt1("1")
                          )
                    )
              );
```

In den folgenden Abschnitten werden Funktionen angegeben, die Durchläufen durch Binärbäume (mit Knotenmarkierungen) entsprechen und die die Markierungen der Knoten in Listen aufsammeln.

8.7.1 Selektoren und Prädikate

Selektoren und Prädikate für Binärbäume lassen sich in naheliegender Weise definieren (auf das hier verwendete Konzept der `exception` gehen wir in Kapitel 10 ein):

```
- exception binbaum5_leer;
- exception binbaum5_kein_nachfolger;

- fun wurzel(Leer)          = raise binbaum5_leer
    | wurzel(Knt1(M))       = M
    | wurzel(Knt2(_, M))    = M
    | wurzel(Knt3(_, M, _)) = M;

- fun linker_baum(Leer)          = raise binbaum5_kein_nachfolger
    | linker_baum(Knt1(_))       = raise binbaum5_kein_nachfolger
    | linker_baum(Knt2(_, _))    = raise binbaum5_kein_nachfolger
    | linker_baum(Knt3(B, _, _)) = B;

- fun rechter_baum(Leer)          = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt1(_))       = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt2(_, _))    = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt3(_, _, B)) = B;

- fun ist_leer(Leer) = true
    | ist_leer(_)    = false;
```

Damit sind Zugriffe auf Bestandteile von Binärbäumen möglich:

```
- wurzel(B);
val it = 3 : int

- wurzel(rechter_baum(B));
val it = 5 : int

- wurzel(linker_baum(A));
val it = "*" : string
```

Die folgenden Definitionen sind alle mit Hilfe des Pattern Matching aufgebaut, so dass die obigen Funktionen darin nicht benötigt werden.

8.7.2 Durchlauf in Infix-Reihenfolge

Die Infix-Reihenfolge bedeutet, dass zunächst die Knoten des linken Teilbaums, dann die Wurzel und anschließend die Knoten des rechten Teilbaums aufgesammelt werden.

```
- fun infix_collect(Leer)          = nil
    | infix_collect(Knt1(M))       = [M]
    | infix_collect(Knt2(M1,M2))   = [M1,M2]
    | infix_collect(Knt3(B1,M,B2)) =
                        infix_collect(B1) @ [M] @ infix_collect(B2);
val infix_collect = fn : 'a binbaum5 -> 'a list
```

```
- infix_collect(B);
val it = [1,2,3,4,5,6] : int list

- infix_collect(A);
val it = ["x","*","y","-","2","/","z","+","1"] : string list
```

Die Bezeichnung „Infix-Reihenfolge“ wird aus dem Beispiel des Baums `A` leicht verständlich. Die Ergebnisliste entspricht dem durch den Baum `A` repräsentierten arithmetischen Ausdruck in Infix-Notation, wobei allerdings die Information über die Klammerung verloren gegangen ist.

Die rechte Seite des letzten Falls der Definition könnte eigentlich umformuliert werden: `infix_collect(B1) @ M :: infix_collect(B2)`; in der obigen Definition wurde trotzdem die umständlichere Form `infix_collect(B1) @ [M] @ infix_collect(B2)` geschrieben, weil dadurch Analogien und Unterschiede zu den folgenden Funktionen offensichtlicher werden.

8.7.3 Durchlauf in Präfix-Reihenfolge

Die Präfix-Reihenfolge bedeutet, dass die Wurzel vor den Knoten des linken Teilbaums, und diese vor den Knoten des rechten Teilbaums aufgesammelt werden.

Für die folgende Funktionsdefinition sei daran erinnert, dass in einem Ausdruck `Knt2(M1, M2)` der Knoten `M1` linker Nachfolger des Knotens `M2` ist.

```
- fun praefix_collect(Leer)          = nil
    | praefix_collect(Knt1(M))       = [M]
    | praefix_collect(Knt2(M1,M2))   = [M2,M1]
    | praefix_collect(Knt3(B1,M,B2)) =
                  [M] @ praefix_collect(B1) @ praefix_collect(B2);
val praefix_collect = fn : 'a binbaum5 -> 'a list

- praefix_collect(B);
val it = [3,2,1,5,4,6] : int list

- praefix_collect(A);
val it = ["-","*","x","y","/","2","+","z","1"] : string list
```

Man beachte, dass die Kenntnis der Stelligkeit der Operationen ausreicht, um aus der „Linearisierung“ in Präfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.

Auch hier wäre es angebracht, `[M] @ praefix_collect(B1)` zu vereinfachen zu `M :: praefix_collect(B1)`.

8.7.4 Durchlauf in Postfix-Reihenfolge

In Postfix-Reihenfolge werden zunächst die Knoten des linken Teilbaums, dann des rechten Teilbaums, und schließlich die Wurzel aufgesammelt.

```
- fun postfix_collect(Leer)          = nil
    | postfix_collect(Knt1(M))       = [M]
    | postfix_collect(Knt2(M1,M2))   = [M1,M2]
    | postfix_collect(Knt3(B1,M,B2)) =
                    postfix_collect(B1) @ postfix_collect(B2) @ [M];
val postfix_collect = fn : 'a binbaum5 -> 'a list

- postfix_collect(B);
val it = [1,2,4,6,5,3] : int list

- postfix_collect(A);
val it = ["x","y","*","2","z","1","+","/","-"] : string list
```

Man beachte, dass die Kenntnis der Stelligkeit der Operationen ausreicht, um aus der „Linearisierung“ in Postfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.

8.7.5 Infix-/Präfix-/Postfix-Reihenfolge mit Akkumulatortechnik

Die obigen Definitionen haben ein ähnliches Manko wie die Funktion `naive-reverse` in Abschnitt 5.5.8. Durch den Aufruf von `append` in jedem rekursiven Aufruf summiert sich die Gesamtanzahl der Aufrufe von „`::`“ auf einen unnötig hohen Wert. Die Funktionen sammeln zwar die Markierungen in der gewünschten Reihenfolge auf, aber verschachtelt in Listen, die dann erst wieder von `append` (bzw. „`@`“) zu einer „flachen“ Liste zusammengefügt werden müssen.

In ähnlicher Weise, wie mit Hilfe der Akkumulator-Technik aus der Funktion `naive-reverse` die Funktion `reverse` entwickelt wurde, kann man auch hier mit Hilfe der Akkumulator-Technik bessere Definitionen entwickeln:

```
- fun infix_collect(B) =
      let
        fun collect(Leer,          L) = L
          | collect(Knt1(M),       L) = M :: L
          | collect(Knt2(M1,M2),   L) = M1 :: M2 :: L
          | collect(Knt3(B1,M,B2), L) = collect(B1, M :: collect(B2, L))
      in
          collect(B, nil)
      end;
val infix_collect = fn : 'a binbaum5 -> 'a list
```

Die Reihenfolge, in der die Parameter `B1`, `M`, `B2` in den rekursiven Aufrufen weitergereicht werden, ist jeweils die gleiche wie in den vorhergehenden Definitionen. Man beachte, dass die lokale Hilfsfunktion `collect` trotz der Verwendung der Akkumulator-Technik *nicht* endrekursiv ist, weil einer der rekursiven Aufrufe innerhalb eines zusammengesetzten Ausdrucks vorkommt. Die Definition eines iterativen Prozesses zum Durchlauf durch beliebige Binärbäume ist auch äußerst schwierig und erfordert zusätzliche, aufwändige Datenstrukturen zur Verwaltung.

8.7.6 Tiefendurchlauf (Depth-First-Durchlauf)

Infix-, Präfix- und Postfix-Reihenfolge haben gemeinsam, dass die Teilbäume des Baums unabhängig voneinander durchlaufen werden. Der Unterschied der Funktionen liegt nur darin, wie das Ergebnis jeweils zusammengesetzt wird. Die Gemeinsamkeit der drei Durchlaufreihenfolgen ist als Tiefendurchlauf bekannt und kann mit Hilfe von Funktionen höherer Ordnung leicht als Abstraktion der drei ursprünglichen Funktionen definiert werden.

```
- fun depth_first_collect f0 f1 f2 f3  Leer           = f0
    | depth_first_collect f0 f1 f2 f3 (Knt1(M))       = f1(M)
    | depth_first_collect f0 f1 f2 f3 (Knt2(M1,M2))   = f2(M1,M2)
    | depth_first_collect f0 f1 f2 f3 (Knt3(B1,M,B2)) =
                    f3(depth_first_collect f0 f1 f2 f3 B1,
                       M,
                       depth_first_collect f0 f1 f2 f3 B2);

- fun infix_collect(B) =
      depth_first_collect nil
                          (fn M         => [M])
                          (fn (M1,M2)   => [M1,M2])
                          (fn (R1,M,R2) => R1 @ [M] @ R2)
                          B;

- fun praefix_collect(B) =
      depth_first_collect nil
                          (fn M         => [M])
                          (fn (M1,M2)   => [M2,M1])
                          (fn (R1,M,R2) => [M] @ R1 @ R2)
                          B;

- fun postfix_collect(B) =
      depth_first_collect nil
                          (fn M         => [M])
                          (fn (M1,M2)   => [M1,M2])
                          (fn (R1,M,R2) => R1 @ R2 @ [M])
                          B;
```

Aber auch andere nützliche Funktionen auf Binärbäumen lassen sich mit Hilfe des Tiefendurchlaufs leicht implementieren:

```
- fun anzahl_knoten(B) =
      depth_first_collect 0
                          (fn M         => 1)
                          (fn (M1,M2)   => 2)
                          (fn (R1,M,R2) => R1 + 1 + R2)
                          B;

- fun anzahl_blaetter(B) =
      depth_first_collect 0
```

```
                          (fn M         => 1)
                          (fn (M1,M2)   => 1)
                          (fn (R1,M,R2) => R1 + R2)
                          B;

- fun tiefe(B) =
      depth_first_collect 0
                          (fn M         => 1)
                          (fn (M1,M2)   => 2)
                          (fn (R1,M,R2) => 1 + Int.max(R1,R2))
                          B;

- fun element(x,B) =
      depth_first_collect false
                          (fn M         => x=M)
                          (fn (M1,M2)   => x=M2 orelse x=M1)
                          (fn (R1,M,R2) => x=M orelse R1 orelse R2)
                          B;

- anzahl_knoten(B);
val it = 6 : int

- anzahl_knoten(A);
val it = 9 : int

- anzahl_blaetter(B);
val it = 3 : int

- anzahl_blaetter(A);
val it = 5 : int

- tiefe(B);
val it = 3 : int

- tiefe(A);
val it = 4 : int

- element(5, B);
val it = true : bool

- element(7, B);
val it = false : bool

- element("2", A);
val it = true : bool

- element("3", A);
val it = false : bool
```

8.7.7 Breitendurchlauf (Breadth-First-Durchlauf)

Bei einem Breitendurchlauf werden die Knoten nach wachsender Tiefe besucht. Zuerst wird die Wurzel aufgesammelt, dann die Wurzeln der Nachfolger, dann die Wurzeln von deren Nachfolgern usw. Das Ergebnis des Breitendurchlaufs ist also für Baum `B` die Liste `[3,2,5,1,4,6]` und für Baum `A` die Liste `["-","*","/","x","y","2","+","z","1"]`.

Zur Implementierung bedienen wir uns einer Hilfsfunktion `entwurzeln`, die angewandt auf eine Liste von Bäumen von jedem Baum die Wurzel aufsammelt und die Nachfolger der Wurzel am Ende der Liste für die spätere Weiterverarbeitung einfügt. So soll aus der einelementigen Liste von Binärbäumen

```
[
  Knt3( Knt3(
              Knt1("a"),
              "*",
              Knt1("b")
            ),
        "+",
        Knt3(
              Knt1("e"),
              "-",
              Knt1("f")
            )
      )
]
```

nach dem Aufsammeln von „+“ die folgende zweielementige Liste entstehen:

```
[
  Knt3(
        Knt1("a"),
        "*",
        Knt1("b")
      ),
  Knt3(
       Knt1("e"),
       "-",
       Knt1("f")
      )
]
```

Die Funktion `entwurzeln` zerlegt also einen Baum von der Wurzel her, gerade entgegengesetzt zum Aufbau des Baums gemäß der mathematischen Definition oder mit den Konstruktoren des induktiv definierten Typs `'a binbaum5`. Der Unterschied zu den Selektoren ist, dass die Teilbäume nicht einfach als Ergebnisse geliefert werden, sondern in einer Liste nach dem Prinzip *first-in-first-out* verwaltet werden. Der Breitendurchlauf kann wie folgt implementiert werden:

```
- fun breadth_first_collect(B) =
      let
        fun entwurzeln nil                = nil
          | entwurzeln(Leer::L)           = entwurzeln(L)
          | entwurzeln(Knt1(M)::L)        = M ::entwurzeln(L)
          | entwurzeln(Knt2(M1,M2)::L)    = M2::entwurzeln(L @ [Knt1(M1)])
          | entwurzeln(Knt3(B1,M,B2)::L) = M ::entwurzeln(L @ [B1,B2])
      in
          entwurzeln(B :: nil)
      end;
val breadth_first_collect = fn : 'a binbaum5 -> 'a list

-  breadth_first_collect(B);
val it = [3,2,5,1,4,6] : int list

- breadth_first_collect(A);
val it = ["-","*","/","x","y","2","+","z","1"] : string list
```

Beim Breitendurchlauf werden die Teilbäume nicht unabhängig voneinander durchlaufen, sondern nach Tiefe verzahnt. Das verletzt die Grundannahme des Tiefendurchlaufs, so dass der Breitendurchlauf nicht mit Hilfe des Tiefendurchlaufs definiert werden kann.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 9

Pattern Matching

Der „Musterangleich“ (*Pattern Matching*) ist eine sehr nützliche Technik zur Programmentwicklung. Schon im Abschnitt 2.5.2 wurde Pattern Matching anhand eines einfachen Beispiels eingeführt. In den darauf folgenden Kapiteln sind mehrere weitere Beispiele des Pattern Matching gegeben worden. Pattern Matching erwies sich vor allem nützlich zur Definition von Prozeduren mit Parametern von zusammengesetzten Typen — siehe Abschnitt 5.3 und Kapitel 8. In diesem Kapitel wird zunächst Pattern Matching systematisch und zusammenfassend behandelt. Dann wird seine Implementierung erläutert.

9.1 Zweck des Pattern Matching

9.1.1 Muster in Wertdeklarationen

Pattern Matching dient zur Selektion der Komponenten eines zusammengesetzten Wertes und kann in Wertdeklarationen wie folgt verwendet werden:

```
- val paar = (1.52, 4.856);
val paar = (1.52,4.856) : real * real

- val (komp1, komp2) = paar;
val komp1 = 1.52 : real
val komp2 = 4.856 : real
```

Der Ausdruck `(komp1, komp2)` stellt ein „Muster“ (*Pattern*) dar, das dazu dient, eine Gestalt festzulegen. In der vorangehenden Deklaration kann das Muster `(komp1, komp2)` mit dem Wert des Namens `paar`, d.h. mit dem Vektor `(1.52, 4.856)` „angeglichen“ werden. Dieser Angleich (*matching*) liefert Bindungen für die Namen `komp1` und `komp2`. Ist der Angleich unmöglich, so scheitert die Wertdeklaration, wie etwa in:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y * 'X
  expression:    real * real
  in declaration:
    (k1,k2,k3) =
      (case (1.0,2.0)
        of (k1,k2,k3) => (k1,k2,k3))
```

9.1.2 Muster zur Fallunterscheidung in Funktionsdefinitionen

Muster und die Technik des Pattern Matching werden häufig in Fallunterscheidungen verwendet wie etwa in der folgenden polymorphen Funktionsdefinition aus dem Abschnitt 5.4.1:

```
- fun laenge nil     = 0
    | laenge(h :: L) = 1 + laenge(L);
val laenge = fn : 'a list -> int

- laenge [0,1,2,3];
val it = 4 : int
```

Die Muster `nil` und `(h :: L)` decken alle Listenarten ab: Das Muster `nil` deckt den Fall der leeren Liste ab, das Muster `(h :: L)` deckt den Fall aller nichtleeren Listen ab.

Die statische Typprüfung von SML erkennt, ob die verschiedenen Fälle einer Funktionsdeklaration etwaige Fälle nicht berücksichtigen, wie das folgende Beispiel einer polymorphen Listenfunktion aus dem Abschnitt 5.4.2 zeigt:

```
- fun letztes_element(x :: nil) = x
    | letztes_element(h :: L)   = letztes_element(L);
Warning: match nonexhaustive
          x :: nil => ...
          h :: L => ...
val letztes_element = fn : 'a list -> 'a

- letztes_element [1,2,3,4];
val it = 4 : int
```

Selbstverständlich handelt es sich dabei um keinen Fehler. Der Fall „leere Liste“ fehlt deshalb in der Definition der Funktion `letztes_element`, weil eine leere Liste kein letztes Element haben kann. Die Typprüfung, die das SML-System leistet, kennt die Semantik der Funktion aber nicht. Sie erkennt lediglich, dass die Fallunterscheidung den Fall „leere Liste“ nicht abdeckt und macht den Programmierer darauf aufmerksam, der die Semantik der Funktion kennt — oder kennen sollte.

Muster und Pattern Matching werden verwendet, wenn Funktionen, insbesondere Selektoren, über zusammengesetzten Typen, u.a. über rekursiven Typen, definiert werden. Die polymorphen Selektoren `head` (in SML als `hd` vordefiniert) und `tail` (in SML als `tl` vordefiniert) für Listen aus den Abschnitten 5.5.5 und 5.5.6 sind Beispiele dafür:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

- head [1,2,3];
val it = 1 : int

- fun tail(_ :: L) = L;
Warning: match nonexhaustive
          _ :: L => ...
val tail = fn : 'a list -> 'a list
```

```
- tail [1,2,3];
val it = [2,3] : int list
```

Auch im folgenden Beispiel werden Muster und Pattern Matching verwendet, um eine Funktion über zusammengesetzten Werten zu deklarieren. In diesem Fall handelt es sich um einen selbst definierten rekursiven Typ.

```
- datatype binbaum1 = Leer
                    | Blt of int                   (* Blt f"ur Blatt  *)
                    | Knt of binbaum1 * binbaum1; (* Knt f"ur Knoten *);
datatype binbaum1 = Blt of int
                  | Knt of binbaum1 * binbaum1
                  | Leer;

- val b = Knt(Knt(Blt(1), Blt(3)), Knt(Blt(5), Knt(Blt(8), Blt(9))));
val b = Knt (Knt (Blt #,Blt #),Knt (Blt #,Knt #)) : binbaum1
```

Dieser Binärbaum kann wie folgt dargestellt werden:

```
                        *
                      /   \
Knt(Blt(1), Blt(3))  *       *  Knt(Blt(5), Knt(Blt(8), Blt(9)))
                    / \     / \
                   1   3   5   *  Knt(Blt(8), Blt(9))
                              / \
                             8   9
```

```
- fun blaetterzahl Leer          = 0
    | blaetterzahl (Blt(_))      = 1
    | blaetterzahl (Knt(b1, b2)) =  blaetterzahl b1 + blaetterzahl b2;
val blaetterzahl = fn : binbaum1 -> int

- blaetterzahl(b);
val it = 5 : int
```

In der Definition der Funktion `blaetterzahl` kommt eine Fallunterscheidung vor, deren drei Fälle durch Muster entsprechend der Definition des Typs `binbaum1` definiert sind.

Muster und die Technik des Pattern Matching bieten sich an zur Definition von Funktionen über Typen mit endlichen Wertemengen. Dann kann man einfach für jeden der endlich vielen Werte einen Fall angeben. Ein Beispiel ist die folgende Implementierung einer logischen Disjunktion `OR`:

```
- fun OR(true,  true)  = true
    | OR(true,  false) = true
    | OR(false, true)  = true
    | OR(false, false) = false;
val OR = fn : bool * bool -> bool

- OR(OR(false, true), true);
val it = true : bool
```

Zur Auswertung eines Ausdrucks der Gestalt `OR(A,A')` wertet SML immer beide Argumente von `OR` aus, während bei einem Sonderausdruck `A orelse A'` der Teilausdruck `A'` nicht immer ausgewertet wird. Diese Frage der Auswertungsreihenfolge hängt aber ausschließlich davon ab, dass `OR` eine Funktion ist, und nicht davon, ob diese Funktion mit Hilfe von Mustern definiert ist oder auf andere Weise.

9.1.3 Muster zur Fallunterscheidung in `case`–Ausdrücken

Muster und die Technik des Pattern Matching können auch in herkömmlichen Ausdrücken verwendet werden. Die Funktion `OR` aus Abschnitt 9.1.2 kann z.B. wie folgt neu definiert werden:

```
- fun OR'(x, y) = case x  of true  => true
                           | false => y;
val OR' = fn : bool * bool -> bool

- OR'(OR'(false, true), true);
val it = true : bool
```

Man beachte, dass in SML auch bei dieser Definition stets beide Argumente ausgewertet werden. Diese Frage hängt wie gesagt nur davon ab, ob ein Ausdruck ein Sonderausdruck oder eine Funktionsanwendung ist, aber nicht davon, mit welchen Hilfsmitteln die Funktion definiert ist.

Die Behandlung von Mustern in `case`–Ausdrücken ist übrigens in vielen SML-Implementierungen die Basis, auf die alle anderen Verwendungen von Mustern intern zurückgeführt werden. Man erkennt das zum Beispiel an der Fehlermeldung bei einer Wertdeklaration, in der Muster und Wert nicht zusammenpassen:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y * 'X
  expression:    real * real
  in declaration:
    (k1,k2,k3) =
      (case (1.0,2.0)
        of (k1,k2,k3) => (k1,k2,k3))
```

Die Deklaration wurde offenbar intern in einen Ausdruck übersetzt, der mit `case` gebildet ist.

9.2 Prinzip des Pattern Matching

9.2.1 Angleichregel

Eine sogenannte *Angleichregel* ist ein Ausdruck der folgenden Gestalt:

<Muster> => <Ausdruck>

In `fun`–Deklarationen sind Angleichregeln „versteckt“, die ersichtlich werden, wenn die `fun`–Deklarationen durch ihre Bedeutung als `val`–Ausdrücke ersetzt werden. So steht z.B. die `fun`–Deklaration der Funktion `OR` aus Abschnitt 9.1.2:

```
- fun OR(true,  true)  = true
    | OR(true,  false) = true
    | OR(false, true)  = true
    | OR(false, false) = false;
val OR = fn : bool * bool -> bool
```

für die folgende `val`–Deklaration, in der Angleichregeln vorkommen:

```
- val rec OR = fn (true,  true)  => true
                | (true,  false) => true
                | (false, true)  => true
                | (false, false) => false;
val OR = fn : bool * bool -> bool
```

9.2.2 Prüfung einer Angleichregel gegen einen Wert

Eine Angleichregel

<Muster> => <Ausdruck>

wird wie folgt gegen einen Wert `W` geprüft. `W` wird mit dem Muster *<Muster>* angeglichen. Gelingt der Angleich, so werden etwaige Namen, die im Muster vorkommen, gebunden.

Der Angleich erfolgt dadurch, dass die Strukturen von Muster und Wert gleichzeitig durchlaufen, oder zerlegt, werden und dass dabei die Teilausdrücke komponentenweise und rekursiv angeglichen werden. Die Basisfälle dieses rekursiven Algorithmus liefern Namen und Konstanten, die im Muster vorkommen. Ein in einem Muster vorkommender Name kann, so lange die Typen übereinstimmen, mit jedem Wert angeglichen werden. Wird ein Name, der im Muster vorkommt, an einen Wert angeglichen, so wird der Wert an diesen Namen gebunden. Eine in einem Muster vorkommende Konstante kann nur mit derselben Konstante angeglichen werden.

Wie die vorangehende informelle Beschreibung des Pattern Matching suggeriert, stellen Konstanten und Namen die einfachsten Muster dar, was die Gestalt angeht.

Wie der Angleich genau durchgeführt wird, wird in Abschnitt 9.8 am Ende dieses Kapitels erläutert.

9.2.3 Prüfung eines Angleichmodells gegen einen Wert

Angleichregeln werden wie folgt zu einem *Angleichmodell* zusammengesetzt:

<Angleichregel> `|` `...|` *<Angleichregel>*

Die Angleichregeln werden sequenziell durchlaufen, wenn ein Wert `W` gegen sie geprüft wird. Ist der Angleich von `W` mit dem Muster einer Angleichregel des Angleichmodells erfolgreich, so wird `W` gegen die nachfolgenden Angleichregeln nicht geprüft.

9.2.4 Typkorrektheit eines Angleichmodells

Ein Angleichmodells der Gestalt

$\texttt{Muster}_1$ => $\texttt{Ausdruck}_1$ | $\texttt{Muster}_2$ => $\texttt{Ausdruck}_2$ | ...| $\texttt{Muster}_n$ => $\texttt{Ausdruck}_n$

ist nur dann korrekt typisiert, wenn:

1. $\texttt{Muster}_1$, $\texttt{Muster}_2$, ..., $\texttt{Muster}_n$ alle denselben Typ haben, und
2. $\texttt{Ausdruck}_1$, $\texttt{Ausdruck}_2$, ..., $\texttt{Ausdruck}_n$ alle denselben Typ haben.

Die Muster können aber einen anderen Typ haben als die Ausdrücke.
Diese Bedingungen können während der statischen Typprüfung überprüft werden (siehe Abschnitt 9.3 unten).

9.2.5 Herkömmliche Angleichmodelle in SML

Aus den bisher beschriebenen Prinzipien ergeben sich die beiden herkömmlichen Formen von Angleichmodellen in SML:

```
case <Ausdruck> of <Angleichmodell>
```

und

```
fn <Angleichmodell>
```

Wir erinnern daran, dass die `fun`–Deklarationen mit Pattern Matching lediglich eine „syntaktische Verzuckerung“ des vorangehenden Angleichmodells mit `fn`–Ausdrücken darstellen.

9.3 Pattern Matching und statische Typprüfung — Angleichfehler zur Laufzeit

9.3.1 Pattern Matching und statische Typprüfung

Schon während der statischen Typprüfung kann ein Angleichmodell auf Typkorrektheit überprüft werden, wie etwa in:

```
- fun f (0, true) = true
    | f (1, 2)    = false;
Error: parameter or result constraints of clauses don't agree [literal]
  this clause:      int * int -> 'Z
  previous clauses:      int * bool -> 'Z
  in declaration:
    f =
      (fn (0,true) => true
        | (1,2)    => false)
```

In einem solchen Fall wird lediglich ein Typfehler festgestellt: In diesem Beispiel besteht der Typfehler darin, dass die beiden Muster `(0, true)` und `(1, 2)` des Angleichmodells nicht denselben Typ haben.
Auch unterschiedliche Typen bei Ausdrücken in einem Angleichmodell werden statisch erkannt:

```
- fun f true  = 1
    | f false = 0.0;
Error: right-hand-side of clause doesn't agree
with function result type [literal]
  expression:  real
  result type:  int
  in declaration:
    f =
      (fn true => 1
        | false => 0.0)
```

9.3.2 Angleichfehler zur Laufzeit

Da Angleichmodelle, die nicht alle Fälle abdecken, möglich sind, kann es vorkommen, dass Angleichfehler erst zur Laufzeit erkannt werden können statt schon zur Übersetzungszeit:

```
- fun f(0) = 0
    | f(x) = if x > 0 then f(x-1) else f(~x-1);
val f = fn : int -> int

- fun g(1) = true;
Warning: match nonexhaustive
          1 => ...
val g = fn : int -> bool

- g(f(1));
uncaught exception nonexhaustive match failure
```

Ohne Auswertung des Ausdrucks `f(1)` ist nicht ersichtlich, dass `f(1)` einen anderen Wert als `1` hat. Es ist also nicht möglich, statisch festzustellen, dass kein Muster der Deklaration der Funktion `g` den Fall `g(f(1))` abdeckt.

9.4 Das Wildcard–Muster von SML

Das sogenannte *Wildcard–Muster* von SML, „_“ geschrieben, hat zwei Eigenschaften:

- Zum einen kann es an jeden Wert angeglichen werden.
- Zum anderen wird es bei einem erfolgreichen Angleich an keinen Wert gebunden.

Typischerweise wird es als Fangfall verwendet werden, wie etwa in der folgenden Funktionsdeklaration aus Abschnitt 2.5.2:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
              | 5 => true
              | 6 => true
              | 7 => true
              | 8 => true
              | 9 => true
              | _ => false;
```

Wie die folgende Funktionsdeklaration aus Abschnitt 5.5.5 zeigt, darf das Wildcard–Muster auch in einem zusammengesetzten Ausdruck vorkommen, um an Teilausdrücke angeglichen zu werden, die im definierenden Teil der Funktionsdefinition nicht von Belang sind:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

- head([1,2,3]);
val it = 1 : int
```

Da der Rest der Liste in der Definition der Funktion `head` keine Rolle spielt, ist es sinnvoll, anstatt eines Namens wie etwa `t` das Wildcard–Muster zu verwenden. So wird der Leser auf die Absicht des Programmierers aufmerksam gemacht.

9.5 Das Verbund-Wildcard–Muster von SML

Wir erinnern daran, dass Verbunde zusammengesetzte Strukuren sind, deren Komponenten Namen tragen, wie etwa im folgendem Beispiel aus Abschnitt 5.3.3:

```
- val adressbucheintrag = {Nachname          = "Bry",
                           Vornamenbuchstabe = #"F",
                           Durchwahl         = "2210"};
val adressbucheintrag =
  {Durchwahl="2210",Nachname="Bry",Vornamenbuchstabe=#"F"}
  : {Durchwahl:string, Nachname:string, Vornamenbuchstabe:char}

- adressbucheintrag = {Vornamenbuchstabe = #"F",
                       Durchwahl         = "2210",
                       Nachname          = "Bry"};
val it = true : bool
```

Wie der Test zeigt, ist für einen Verbund die Reihenfolge der Verbundkomponenten irrelevant.

Die vollständige Angabe eines Verbundes verlangt oft viel Schreibarbeit. Mit dem sogenannten *Verbund-Wildcard–Muster* leistet SML dabei dem Programmierer oder dem Leser eines Programms Abhilfe:

```
- val {Nachname = NN, ...} = adressbucheintrag;
val NN = "Bry" : string
```

So kann der Name `NN` an den Wert `"Bry"` (vom Typ Zeichenfolge) gebunden werden, ohne dass das Muster, in dem der Name `NN` vorkommt, die Struktur des Verbundes vollständig angibt.

```
- val v1 = {a = 1, b = 2};
val v1 = {a=1,b=2} : {a:int, b:int}

- val v2 = {a = 0, b = 2, c = 3};
val v2 = {a=1,b=2,c=3} : {a:int, b:int, c:int}

- val {a = N, ...} = v1;
val N = 1 : int

- val {a = N, ...} = v2;
val N = 0 : int
```

In beiden Wertdeklarationen, die ein Verbund-Wildcard–Muster enthalten, ist eindeutig, an welchen Wert der Name `N` gebunden werden soll.

Eine Verwendung des Verbund-Wildcard–Musters setzt aber voraus, dass der Typ des Verbundes, in dem das Verbund-Wildcard–Muster vorkommt, statisch, also zur Übersetzungszeit, bestimmt werden kann. Da das bei der folgenden Funktionsdeklaration nicht möglich ist, wird ein Fehler gemeldet:

```
- fun f({a = 1, ...}) = true
    | f(_)            = false;
  Error: unresolved flex record
(can't tell what fields there are besides #a)
```

9.6 Die gestuften Muster von SML

SML bietet zudem die sogenannten „gestuften Muster“ (*layered pattern*) an, die am folgenden Beispiel eingeführt werden können:

```
- val info1dozent = {Dozent = ("Bry", #"F"), Raum = "D1.04"};
val info1dozent = {Dozent=("Bry",#"F"),Raum="D1.04"}
  : {Dozent:string * char, Raum:string}

- val {Dozent = D as (N,V), Raum = R} = info1dozent;
val D = ("Bry",#"F") : string * char
val N = "Bry" : string
val V = #"F" : char
val R = "D1.04" : string
```

Mit dem Konstrukt

```
D as (N,V)
```

erfolgen neben der Bindung des Wertes `("Bry", #"F")` an den Namen `D` die Bindungen des Wertes `"Bry"` an den Namen `N` und die Bindung des Wertes `#"F"` an den Namen `V`.

9.7 Linearitätsbedingung für Muster

Die Verwendung von Mustern und der Technik des Pattern Matching verlangen, dass die Muster in dem Sinne *linear* sind, dass eine ungebundene Variable, die durch den Angleich des Musters an einen Wert gebunden werden soll, nur einmal — daher die Bezeichnung linear — im Muster vorkommt.
Die folgende polymorphe Funktion

```
- fun gleich(x, y) = (x = y);
val gleich = fn : ''a * ''a -> bool

- gleich([1,2], [1,2]);
val it = true : bool

- gleich(#"a", #"a");
val it = true : bool
```

kann also nicht wie folgt deklariert werden:

```
- val gleich = fn (x, x) => true
                |  _     => false;
Error: duplicate variable in pattern(s): x
```

Der Grund für diese Einschränkung ist, den Pattern-Matching–Algorithmus einfach und folglich effizient zu halten. Auch die Typprüfung wäre ohne diese Einschränkung betroffen, weil Mehrfachvorkommen von Variablen in Mustern zur Folge haben müssten, dass die Werte an den entsprechenden Positionen zu Gleichheitstypen gehören müssen.

9.8 Der Pattern-Matching-Algorithmus

9.8.1 Informelle Spezifikation des Pattern-Matching–Algorithmus

Im Abschnitt 9.2.2 wurde der Pattern-Matching–Algorithmus wie folgt informell erläutert:

> Der Angleich (zwischen einem Muster und einem Wert) erfolgt dadurch, dass die Strukturen von Muster und Wert gleichzeitig durchlaufen, oder zerlegt, werden und dass dabei die Teilausdrücke komponentenweise und rekursiv angeglichen werden. Die Basisfälle von diesem rekursiven Algorithmus liefern

> Namen und Konstanten, die im Muster vorkommen. Ein in einem Muster vorkommender Name kann, so lange die Typen übereinstimmen, mit jedem Wert angeglichen werden. Wird ein Name, der im Muster vorkommt, an einen Wert angeglichen, so wird der Wert an diesen Namen gebunden. Eine in einem Muster vorkommende Konstante kann nur mit derselben Konstante angeglichen werden.

Man beachte, dass Pattern Matching zwischen einem Muster `M` und einem Wert `W` erfolgt, dass aber in einem Programm jeweils ein Muster `M` und ein Ausdruck `A` gegeben sind, zum Beispiel in der Form

`val M = A` oder `case A of M =>`

Bei der Auswertung wird zunächst `A` in der aktuellen Umgebung ausgewertet zu einem Wert `W`. Danach wird der Pattern-Matching-Algorithmus auf das Muster `M` und diesen Wert `W` angewandt.

9.8.2 Umgebung (Wiederholung aus Kapitel 2)

Im Abschnitt 2.7.3 wurde die Umgebung zur Verwaltung von Bindungen von Werten an Namen wie folgt eingeführt:

> Das SML-System verwaltet mit jeder Sitzung und jeder eingelesenen Datei, d.h. Programm, eine geordnete Liste von Gleichungen der Gestalt `Name = Wert` (dargestellt als Paare `(Name, Wert)`), die Umgebung heißt. Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N = W` am Anfang der Umgebung. Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. So gilt immer als Wert eines Namens `N` derjenige Wert, der bei der letzten Deklaration von `N` angegeben wurde.

Der Pattern-Matching–Algorithmus muss sein Ergebnis in einer Form liefern, die für die Erweiterung der aktuellen Umgebung um neue Einträge geeignet ist.

9.8.3 Formale Spezifikation des Pattern-Matching–Algorithmus

Der Einfachheit halber werden im folgenden Algorithmus die gestuften Muster und das Verbund-Wildcard–Muster nicht berücksichtigt. Die entsprechende Erweiterung des Algorithmus stellt aber keine große Schwierigkeit dar.

Der Pattern-Matching-Algorithmus wird auf ein Muster `M` und einen Wert `W` angewandt.

Der Pattern-Matching–Algorithmus soll, wenn möglich, `M` und `W` angleichen und dabei Werte für die Namen, die im Muster `M` vorkommen, ermitteln. Die Bindung dieser Werte an diese Namen erfolgt aber nicht während des Pattern Matching, sondern erst danach. In anderen Worten verändert der Pattern-Matching–Algorithmus die aktuelle Umgebung nicht, er liefert nur die Bindungen, um die die Umgebung anschließend erweitert werden kann.

Der Pattern-Matching–Algorithmus soll auch feststellen, dass der Angleich von Muster `M` und Wert `W` unmöglich ist. In dem Fall sagt man, dass der Angleich von `M` und `W` gescheitert ist.

Der Pattern-Matching-Algorithmus meldet also

- entweder einen Erfolg und liefert eine (endliche) Menge von Bindungen für die (endlich vielen) Namen, die im Muster `M` vorkommen (dabei bleibt die Umgebung unverändert); Jede dieser Bindungen wird dargestellt als ein Paar `(Name, Wert)`;
- oder ein Scheitern.

Zur Angleichung eines Musters `M` und eines Werts `W` gehe wie folgt vor:

1. Falls `M` eine Konstante `k` ist, dann:
   (a) Falls `W` ebenfalls die Konstante `k` ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.
   (b) Andernfalls terminiere erfolglos.
2. Falls `M` ein Name ist, dann: liefere die einelementige Menge {`(M,W)`} von Bindungen und terminiere erfolgreich.
3. Falls `M` das Wildcard–Muster `_` ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.
4. Falls `M` zusammengesetzt ist mit (Wert-)Konstruktor `K` und Teilmustern $\mathtt{M}_1, \ldots, \mathtt{M}_n$, dann:
   (a) Falls `W` ebenfalls zusammengesetzt ist mit demselben Konstruktor `K` und Teilwerten $\mathtt{W}_1, \ldots, \mathtt{W}_n$, dann: wende für jedes $\mathtt{i} \in \{1, \ldots, n\}$ den Pattern-Matching–Algorithmus auf das Muster $\mathtt{M}_i$ und den Wert $\mathtt{W}_i$ an.
      i. Falls eine dieser Anwendungen des Pattern-Matching–Algorithmus scheitert, dann terminiere erfolglos.
      ii. Andernfalls bilde die Vereinigung aller Bindungsmengen, die diese Anwendungen des Pattern-Matching–Algorithmus liefern; liefere die so erhaltene Menge von Bindungen und terminiere erfolgreich.
   (b) Andernfalls terminiere erfolglos.

9.8.4 Beispiel einer Anwendung des Pattern-Matching–Algorithmus

`M` sei das Muster `e1 :: (e2 :: _)`,
`W` sei der Wert `1 :: (2 :: (3 :: (4 :: (5 :: nil))))`:

Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:

`K` ist der Konstruktor `::`, `M1` ist das Muster `e1`, `M2` ist das Muster `e2 :: _`

Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:

`W1` ist der Wert `1`, `W2` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`

- Anwendung des Pattern-Matching-Algorithmus auf `M1` und `W1`:
  Nebenrechnung, in der gilt:
  `M` ist das Muster `e1`, `W` ist der Wert `1`

Fall 2: `M` ist ein Name:
liefere die Menge {`(e1,1)`} und terminiere erfolgreich

Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e1,1)`}

- Anwendung des Pattern-Matching-Algorithmus auf `M2` und `W2`:
  Nebenrechnung, in der gilt:
  `M` ist das Muster `e2 :: _`, `W` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`

  Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:
  `K` ist der Konstruktor `::`, `M1` ist das Muster `e2`, `M2` ist das Muster `_`

  Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:
  `W1` ist der Wert `2`, `W2` ist der Wert `3 :: (4 :: (5 :: nil))`

  - Anwendung des Algorithmus auf `M1` und `W1`:
    Nebenrechnung, in der gilt:
    `M` ist das Muster `e2`, `W` ist der Wert `2`

    Fall 2: `M` ist ein Name:
    liefere die Menge {`(e2,2)`} und terminiere erfolgreich

    Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e2,2)`}
  - Anwendung des Algorithmus auf `M2` und `W2`:
    Nebenrechnung, in der gilt:
    `M` ist das Muster `_`, `W` ist der Wert `3 :: (4 :: (5 :: nil))`

    Fall 3: `M` ist das Wildcard–Muster `_`:
    liefere die leere Menge und terminiere erfolgreich.

    Ende der Nebenrechnung mit Erfolg, Ergebnis {}

  Fall 4(a)ii: Beide Anwendungen waren erfolgreich:
  Bilde die Vereinigung von {`(e2,2)`} und {};
  liefere {`(e2,2)`} und terminiere erfolgreich

Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e2,2)`}

Fall 4(a)ii: Beide Anwendungen waren erfolgreich:

Bilde die Vereinigung von {`(e1,1)`} und {`(e2,2)`};
liefere {`(e1,1), (e2,2)`} und terminiere erfolgreich.

Ende der gesamten Berechnung, Erfolg, Ergebnis (e1,1), (e2,2).

9.8.5 Korrektheit und Terminierung des Pattern-Matching–Algorithmus

Die Korrektheit des Pattern-Matching–Algorithmus bedeutet:

- Wenn der Pattern-Matching–Algorithmus eine Menge von Bindungen liefert, dann ergibt eine Ersetzung der Namen im Muster durch die Werte, die die Bindungen diesen Namen zuordnen, genau den Wert, mit dem das Muster angeglichen wurde.
- Wenn der Pattern-Matching-Algorithmus ein Scheitern meldet, dann gibt es keine Bindungen mit dieser Eigenschaft.

Diese Aussage lässt sich durch strukturelle Induktion beweisen, wobei für jeden möglichen (Wert-)Konstruktor ein Induktionsfall nötig ist.

Die Terminierung lässt sich ebenfalls durch strukturelle Induktion beweisen. Entscheidend dabei sind die folgenden Beobachtungen:

- Die Fälle 1, 2 und 3 des Pattern-Matching-Algorithmus, die nicht zusammengesetzte Muster behandeln, terminieren offensichtlich.
- Die Terminierung des Falles 4 des Pattern-Matching-Algorithmus, der zusammengesetzte Muster behandelt, wird induktiv bewiesen. Dabei sind die Induktionsannahmen, dass die `n` Anwendungen des Pattern-Matching-Algorithmus auf $\texttt{M}_\texttt{i}$ und $\texttt{W}_\texttt{i}$ für jedes $\texttt{i} \in \{\texttt{1}, \ldots, \texttt{n}\}$ terminieren.

9.8.6 Pattern Matching und Unifikation

Der Pattern-Matching-Algorithmus erinnert an den Unifikationsalgorithmus.

In der Tat stellt er einen Sonderfall der Unifikation dar, der darin besteht, dass nur in einem der beiden Parameter ungebundene Namen vorkommen können, namlich nur im Muster. Offenbar führt diese Einschränkung zu einem wesentlich einfacheren Algorithmus.

Die Linearitätsbedingung (siehe Abschnitt 9.7) trägt ebenfalls zur Vereinfachung gegenüber dem Unifikationsalgorithmus bei.

9.8.7 Folgen der Linearätsbedingung für den Pattern-Matching–Algorithmus

Die Linearitätsbedingung für Muster (siehe Abschnitt 9.7) macht es möglich, dass während einer Anwendung des Pattern-Matching-Algorithmus die erzeugten Bindungen unbesehen in die Ergebnismenge übernommen werden können. Der Ablauf des Pattern-Matching-Algorithmus hängt an keiner Stelle davon ab, welche Bindungen erzeugt wurden. Insbesondere sind die Bindungsmengen, die im Schritt 4(a)ii vereinigt werden, garantiert disjunkt, so dass die Vereinigung durch ein triviales Aneinanderhängen implementiert werden kann.

Ohne die Linearitätsbedingung könnten Namen mehrfach vorkommen, so dass die `n` Bindungsmengen, die im Schritt 4(a)ii vereinigt werden, mehrere Bindungen für denselben Namen enthalten könnten, die obendrein teilweise gleich und teilweise verschieden sein könnten. Bei gleichen Bindungen für einen Namen müssten die Mehrfachvorkommen erkannt und entfernt werden, bei verschiedenen Bindungen für einen Namen müssten diese erkannt und ein Scheitern gemeldet werden. Das ist zwar alles implementierbar, aber nur mit höherer Komplexität des Algorithmus.

Eine weitere Folge der Linearitätsbedingung ist, dass der Ablauf des Pattern-Matching–Algorithmus überhaupt nicht mehr von anderen Daten abhängt als vom Muster, das bereits zur Übersetzungszeit bekannt ist. Ohne die Linearitätsbedingung würden die rekursiven Aufrufe auch von den Bindungen abhängen, die erst zur Laufzeit bekannt sind. Wenn die Linearitätsbedingung eingehalten wird, kann man die rekursiven Aufrufe bereits zur Übersetzungszeit „entfalten“ in eine einzige Schachtelung von `if-then-else`. Die Struktur dieser Schachtelung hängt nur vom Muster ab. Ob diese Optimierung von SML-Implementierungen tatsächlich durchgeführt wird, ist damit nicht festgelegt, ent-

scheidend ist nur, dass durch die Entscheidung für die Linearitätsbedingung beim Design der Programmiersprache diese Optimierung ermöglicht wurde.

Neuere funktionale Sprachen mit Pattern Matching verlangen die Linearitätsbedingung noch aus einem anderen Grund. Sie haben wesentlich differenziertere Gleichheitstypen als SML, mit denen man zum Beispiel jeweils eigene Gleichheitsprädikate verbinden kann. Dann wären Mehrfachvorkommen einer Variablen in einem Muster nur sinnvoll, wenn jeweils das zugehörige Gleichheitsprädikat mit angegeben würde, wodurch die Syntax von Mustern praktisch unlesbar würde.

Die Linearitätsbedingung für Muster hat also nicht nur den Vorteil, einen einfachen Pattern-Matching-Algorithmus zu ermöglichen. Sie macht zudem die Verwendung von Mustern für Programmierer einfacher und übersichtlicher.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 10

Auswertung und Ausnahmen

Aufbauend auf den Auswertungsalgorithmen der Abschnitte 3.1.3 und 3.4.6 wird in diesem Kapitel ein Auswerter (*evaluator*) oder Interpretierer (*interpreter*) für eine einfache funktionale Programmiersprache in SML implementiert. Dann wird am Beispiel dieses Auswerters gezeigt, wie Ausnahmen verwendet werden können, um fehlerhafte Parameter von Funktions- oder Prozeduraufrufen zu behandeln. Schließlich wird die Behandlung von Ausnahmen während der Auswertung erläutert.

10.1 Die Programmiersprache SMalL

SMalL ist eine Vereinfachung von SML mit den folgenden Merkmalen:

10.1.1 Typen in SMalL

SMalL bietet nur zwei Typen an: die ganzen Zahlen und die Funktionen. SMalL lässt Funktionen höherer Ordnung zu.

Das Typsystem von SMalL ist nicht erweiterbar, d.h., der Programmierer kann keine neuen Typen definieren.

Man beachte, dass SMalL keine zusammengesetzten Typen hat. Folglich können in SMalL nur einstellige Funktionen definiert werden. Da SMalL aber Funktionen höherer Ordnung zulässt, können in SMalL mehrstellige Funktionen in curried Form (siehe Abschnitt 7.2 „Currying“) definiert werden.

Über den ganzen Zahlen bietet SMalL die folgenden vordefinierten Funktionen an:

- Multiplikation einer ganzen Zahl mit -1 (im Folgenden `Minus` genannt)
- Betrag (*absolute value*) einer ganzen Zahl
- Addition zweier ganzer Zahlen
- Subtraktion zweier ganzer Zahlen
- Multiplikation zweier ganzer Zahlen
- Ganzzahlige Division zweier ganzer Zahlen
- Rest bei einer ganzzahligen Division (im Folgenden Modulo genannt)

Einige der vordefinierten Funktionen von SMalL sind also zweistellig.

10.1.2 Verzweigung in SMalL

Zur Verzweigung bietet SMalL nur das Konstrukt `if-then-else` an. Insbesondere sind also `case`–Ausdrücke und Pattern Matching nicht in SMalL vorhanden.

Als Bedingung einer `if-then-else`–Verzweigung ist nur ein Vergleich von ganzen Zahlen erlaubt. Dafür bietet SMalL die folgenden vordefinierten Vergleichsoperatoren an:

- Gleich
- Echt kleiner
- Echt größer

sowie

- Negation von Vergleichsoperatoren

Diese Vergleichsoperatoren dürfen nur zwischen zwei SMalL-Ausdrücken stehen, deren Werte ganze Zahlen sind. Der Vergleich von Funktionen ist in SMalL (wie übrigens auch in SML) nicht möglich. Der Grund für diese Einschränkung liegt darin, dass es im allgemeinen unmöglich ist, zu erkennen, ob zwei Algorithmen, Prozeduren oder Funktionen immer dieselben Ergebnisse liefern.[1]

Da in SMalL Vergleiche ausschließlich als Bedingung von `if-then-else`–Ausdrücken vorkommen dürfen, werden keine SMalL-Ausdrücke vom Typ „Boole'scher Wert“ benötigt.

10.1.3 Globale und lokale Deklarationen in SMalL

Neben globalen Deklarationen lässt SMalL lokale Deklarationen zu. Das folgende Programm ist also in SMalL genauso wie in SML möglich:

```
val a = 1;
val f = fn x => let val a = 2
                in
                    2 * a * x
                end;
val b = a;
val c = f 1;
```

Dieses Programm bindet in SMalL wie in SML die ganze Zahl `1` an `b` und die ganze Zahl `4` an `c`.

Im Gegensatz zu SML ermöglicht SMalL keine (globalen oder lokalen) Deklarationen, die nichtsequenziell ausgewertet werden sollen (wie es in SML mit dem Konstrukt `and` möglich ist — siehe Abschnitt 4.2.8). So ist in SMalL die Definition von wechselseitig rekursiven Funktionen (siehe Abschnitt 4.2.8) nicht möglich.

Die Erweiterung des Auswerters für SMalL, der in den nächsten Abschnitten eingeführt wird, um die Behandlung von nichtsequenziell auszuwertenden Deklarationen stellt keine Schwierigkeit dar und wird nur deswegen ausgelassen, um den Auswerter möglichst klein zu halten.

[1] Die sogenannte Unentscheidbarkeit dieses Problems wurde bewiesen — siehe die Grundstudiumsvorlesung „Informatik 4“ oder die Hauptstudiumsvorlesung „Logik für Informatiker“.

10.1.4 Rekursive Funktionen in SMalL

Es ist möglich, in SMalL rekursive Funktionen zu definieren. Zum Beispiel kann in SMalL die Fakultätsfunktion definiert werden:

```
val rec factorial = fn x => if x > 0
                            then x * factorial(x - 1)
                            else 1;
```

Wie bereits im vorangehenden Abschnitt erwähnt wurde, ist der Einfachheit des SMalL-Auswerters halber in SMalL die Definition von wechselseitig rekursiven Funktionen (siehe Abschnitt 4.2.8) nicht möglich.

10.2 Die abstrakte Syntax von SMalL

10.2.1 Abstrakte Syntax versus konkrete Syntax

In dem folgenden Abschnitt 10.3 wird ein Auswerter für SMalL vorgestellt. Dieser Auswerter ist ein SML-Programm, das SMalL-Ausdrücke als Aufrufparameter erhält, sie auswertet und das Ergebnis dieser Auswertung liefert.

Die SMalL-Ausdrücke, die dieser Auswerter als Aufrufparameter erhält, sind im Gegensatz zu den herkömmlichen Programmen keine Zeichenfolgen, sondern SML-Ausdrücke. Der Auswerter für SMalL wertet also SMalL-Programme aus, die in einer anderen Syntax vorliegen als die Programme, die von den Programmierern geliefert werden. Diese andere Syntax wird „abstrakte Syntax“ genannt; die von den Programmierern verwendete Syntax wird „konkrete Syntax“ genannt (siehe Abschnitt 2.3).

Betrachten wir eine Programmiersprache PS und eine Programmiersprache AS, sowie ein AS-Programm, das die Ausführung von PS-Programmen realisieren soll. Man bezeichnet dann AS als die ausführende Programmiersprache und PS als die auszuführende Programmiersprache. Das AS-Programm erhält also ein beliebiges PS-Programm als Eingabe. Jedes PS-Programm wird vom Programmierer als Zeichenfolge geliefert und könnte im Prinzip in dem AS-Programm auch als Zeichenfolge repräsentiert sein, zum Beispiel als Wert eines AS-Typs `string`. Diese Zeichenfolge müsste das AS-Programm jedesmal analysieren, wenn es auf Teile des PS-Programms zugreift, zum Beispiel auf den Bedingungsteil eines `if-then-else`–Konstrukts im PS-Programm. Da solche Zugriffe wiederholt erforderlich sein können, ist es günstiger, das PS-Programm in eine andere Darstellung zu überführen, die den Zugriff auf Teile des PS-Programms erleichtert.

Die Überführung des PS-Programms in eine andere Darstellung geschieht üblicherweise in mehreren aufeinander aufbauenden Phasen, die je eine Aufgabe erledigen. Die ersten zwei Phasen sind die lexikalische Analyse und die Syntaxanalyse.

- Die lexikalische Analyse transformiert die Zeichenfolge, die das PS-Programm darstellt, in eine Folge von sogenannten „Token“, von denen jeder einem reservierten PS-Bezeichner (wie `let`) oder einem vom Programmierer frei gewählten Bezeichner (wie `x`) oder einer PS-Zahlkonstanten (wie `314`) oder einem PS-Operator (wie `>=`) oder einem anderen Grundsymbol der Programmiersprache PS entspricht. In dieser Folge von „Token“ ist zum Beispiel die Information über Zeilenwechsel, Einrückungen, Leerzeichen usw. nicht mehr repräsentiert.

- Die Syntaxanalyse bildet aus der Folge der „Token“ AS-Ausdrücke, die das PS-Programm darstellen. So könnte ein im PS-Programm vorkommendes `if-then-else`–Konstrukt durch einen AS-Ausdruck `verzweigung(A1,A2,A3)` dargestellt werden. In dieser Darstellung ist zum Beispiel nicht mehr repräsentiert, ob das Konstrukt im PS-Programm in der Form

  ```
  if A1 then A2 else A3
  ```

  oder etwa in der Form

  ```
  if (A1) A2 else A3 end
  ```

  geschrieben werden muss.

Zusammen erzeugen also die lexikalische Analyse und die Syntaxanalyse aus einem PS-Programm in konkreter Syntax ein entsprechendes PS-Programm in abstrakter Syntax.

Die ausführende Programmiersprache AS ist häufig eine Maschinensprache, die gar keine explizite Syntax für Ausdrücke anbietet. AS-Ausdrücke werden dann mit Hilfe von Verweisen (auch Zeiger oder Pointer genannnt — siehe Abschnitt 3.3.2) dargestellt. Aus diesem Grund wird eine abstrakte Syntax oft als „Baumsprache“ angesehen (siehe Abschnitt 2.3). Da aber Ausdrücke lediglich eine lineare Darstellung von Bäumen sind (wie im Abschnitt 8.4.4 besprochen wurde), kann jede abstrakte Syntax auch als eine Sprache von Ausdrücken angesehen und dargestellt werden, auch wenn die verwendete ausführende Programmiersprache solche Ausdrücke nicht explizit anbietet.

Auch wenn die ausführende Programmiersprache AS und die auszuführende Programmiersprache PS dieselben sind, z.B. wenn ein SML-Laufzeitystem in SML selbst implementiert ist, ist die Verwendung einer abstrakten Syntax praktisch unabdingbar, wenn auch, wie gesagt, rein theoretisch nicht zwingend erforderlich.

10.2.2 SML-Typdeklarationen für SMalL-Ausdrücke

SMalL-Ausdrücke werden durch SML-Ausdrücke vom SML-Typ `expression` repräsentiert. Die *abstrakte Syntax* von SMalL wird durch folgende SML-Typen definiert:

```
datatype unary_op      = Min
                       | Abs

datatype binary_op     = Add
                       | Sub
                       | Mult
                       | Div
                       | Mod

datatype comparison_op = ==
                       | >>
                       | <<
                       | Not    of comparison_op
```

```
datatype expression   = IntExp of int
                      | UnOp   of unary_op * expression
                      | BinOp  of binary_op * expression * expression
                      | If     of test * expression * expression
                      | Var    of string
                      | Dcl    of string * expression
                      | FnExp  of string * expression
                      | App    of expression * expression
                      | Seq    of expression * expression

    and test          = Test   of comparison_op * expression * expression
```

Man beachte hier die Verwendung des `and`–Konstrukts von SML. Es ermöglicht die Definition von Typen wie `expression` und `test`, die wechselseitig aufeinander Bezug nehmen.

10.2.3 Beispiele von SMalL-Ausdrücken in konkreter und abstrakter Syntax

```
val a = 3;

Dcl("a", IntExp 3)
```

```
val a = 3;
a;

Seq(Dcl("a", IntExp 3), Var "a")
```

```
val a = 3;
a + a;

Seq(Dcl("a", IntExp 3), BinOp(Add, Var "a", Var "a"))
```

```
val a = 1;
val b = 2;
val c = 3;
(a + b) * c;

Seq(Dcl("a", IntExp 1),
    Seq(Dcl("b", IntExp 2),
        Seq(Dcl("c", IntExp 3),
            BinOp(Mult,
                  BinOp(Add, Var "a", Var "b"),
                  Var "c"
                 )
           )
       )
   )
```

```
2 <> 3

Test(Not ==, IntExp 2, IntExp 3)
```

```
if 2 <> 3 then 1 else 0;

If(Test(Not ==, IntExp 2, IntExp 3),
   IntExp 1,
   IntExp 0)
```

```
val identity = fn x => x;

Dcl("identity", FnExp("x", Var "x"))
```

```
val rec factorial = fn x => if x > 0
                            then x * factorial(x - 1)
                            else 1;

    Dcl("factorial",
        FnExp("x",
              If(Test(>>,
                      Var "x",
                      IntExp 0
                     ),
                 BinOp(Mult,
                       Var "x",
                       App(Var "factorial",
                           BinOp(Sub,
                                 Var "x",
                                 IntExp 1
                                )
                           )
                       ),
                 IntExp 1
                )
             )
       )
```

```
(fn x => x * x) 2

App(FnExp("x", BinOp(Mult, Var "x", Var "x")), IntExp 2)
```

```
val identity = fn x => x;
identity 3;

Seq(Dcl("identity",
        FnExp("x", Var("x"))
       ),
    App(Var "identity",
        IntExp(3)
       )
    )
```

```
val rec factorial = fn x => if x > 0
                            then x * factorial(x - 1)
                            else 1;
factorial 4;

Seq(Dcl("factorial",
        FnExp("x",
              If(Test(>>,
                      Var "x",
                      IntExp 0 ),
                 BinOp(Mult,
                       Var "x",
                       App(Var "factorial",
                           BinOp(Sub,
                                 Var "x",
                                 IntExp 1 )
                          )
                      ),
                 IntExp 1
                )
             )
       ),
    App(Var "factorial", IntExp 4)
   )
```

```
val a = 0;
val f = fn x => x * x;
val a = 2;
val f = fn x => a * x;
f 3;

Seq(Dcl("a",
        IntExp 0 ),
    Seq(Dcl("f",
            FnExp("x", BinOp(Mult, Var "x", Var "x"))
           ),
        Seq(Dcl("a",
                IntExp 2 ),
            Seq(Dcl("f",
                    FnExp("x", BinOp(Mult, Var "a", Var "x"))
                   ),
                App(Var "f",
                    IntExp 3 )
               )
           )
       )
   )
```

```
val f = fn x => let val local = 3
                in
                  local * x
                end;
f 2;

Seq(Dcl("f",
        FnExp("x",
              Seq(Dcl("local",
                      IntExp 3 ),
                  BinOp(Mult, Var "local", Var "x")
                 )
             )
        ),
     App(Var "f",
         IntExp 2 )
   )
```

```
val a = 0;
val f = fn x => x * x;
val a = 2;
val f = fn x => let val a = 3
                in
                  a * x
                end;
f 3;

Seq(Dcl("a",
        IntExp 0 ),
    Seq(Dcl("f",
            FnExp("x", BinOp(Mult, Var "x", Var "x"))
           ),
        Seq(Dcl("a",
                IntExp 2 ),
            Seq(Dcl("f",
                    FnExp("x",
                          Seq(Dcl("a",
                                  IntExp 3 ),
                              BinOp(Mult, Var "a", Var "x")
                             )
                         )
                   ),
                App(Var "f",
                    IntExp 3 )
               )
           )
       )
   )
```

```
val f = fn g => (fn x => let val gx = g x
                         in
                           gx * gx
                         end);
val h = fn x => x + 1;
f h 2;
```

```
Seq(Dcl("f",
        FnExp("g",
              FnExp("x",
                    Seq(Dcl("gx",
                            App(Var "g",
                                Var "x" )
                           ),
                        BinOp(Mult,
                              Var "gx",
                              Var "gx" )
                       )
                   )
              )
       ),
    Seq(Dcl("h",
            FnExp("x",
                  BinOp(Add,
                        Var "x",
                        IntExp 1 )
                 )
           ),
        App(App(Var "f",
                Var "h" ),
            IntExp 2
           )
       )
   )
```

```
val quadrat = (fn x => x * x);
quadrat 3;
```

```
Seq(Dcl("quadrat",
        FnExp("x", BinOp(Mult, Var "x", Var "x"))
       ),
    App(Var "quadrat", IntExp 3)
   )
```

```
val f = fn g => (fn x => let val gx = g x
                         in
                           gx * gx
                         end);
val h = fn x => x + 1;
f h;
```

```
Seq(Dcl("f",
        FnExp("g",
              FnExp("x",
                    Seq(Dcl("gx",
                            App(Var "g",
                                Var "x" )
                           ),
                        BinOp(Mult,
                              Var "gx",
                              Var "gx" )
                       )
                   )
              )
       ),
    Seq(Dcl("h",
            FnExp("x",
                  BinOp(Add,
                        Var "x",
                        IntExp(1) )
                 )
           ),
        App(Var "f",
            Var "h" )
       )
   )
```

Offenbar ist die abstrakte Syntax von SMalL für Menschen schwerer lesbar als die konkrete Syntax. Dagegen kann ein SML-Programm die abstrakte Syntax von SMalL leichter verarbeiten als die konkrete Syntax.

Man beachte, dass die SML-Typdefinitionen unzulässige SMalL-Ausdrücke zulassen, wie etwa die Addition zweier Funktionsdeklarationen oder die Anwendung einer ganzen Zahl auf eine ganze Zahl.

Der Auswerter für SMalL muss also solche Fälle fehlerhafter Aufrufparameter abfangen können.

10.3 Ein Auswerter für SMalL: Datenstrukturen

10.3.1 Werte und Umgebungen

Zu den atomaren Ausdrücken in SMalL (wie in anderen Programmiersprachen) gehören vom Programmierer eingeführte Namen (Namen werden auch Variablen oder Bezeichner, englisch *identifier*, genannt). Der Wert eines Namens hängt von der Umgebung ab. Die Umgebung ist eine Liste, die vom Anfang her nach dem jeweiligen Namen durchsucht wird bis zum ersten Vorkommen des Namens (siehe Abschnitt 2.7.3 sowie den Auswertungsalgorithmus in Abschnitt 3.1.3 und Abschnitt 3.4).

Eine Umgebung kann als Liste von Gleichungen oder von Paaren `(Name, Wert)` dargestellt werden. Welche Werte in Umgebungen vorkommen können, hängt von den Typen der Sprache ab. Im Fall von SMalL gibt es demnach zwei Arten von Werten, ganze Zahlen und Funktionen (siehe Abschnitt 10.1.1).

Eine ganze Zahl ist ein Wert unabhängig von jeglichem Kontext. Eine Funktion dagegen ist ein Wert, der vom Kontext abhängt. Betrachten wir dazu die folgenden SMalL-Programme:

Programm `P1`:

```
val a = 1;
val f = fn x => a * x;
val a = 2;
```

Programm `P2`:

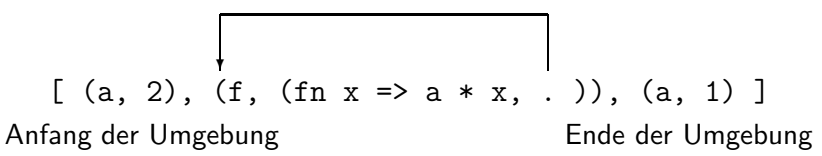
```
val a = 1;
val a = 2;
val f = fn x => a * x;
```

Das Programm `P1` definiert `f` als die Identätsfunktion auf den ganzen Zahlen. Das Programm `P2` definiert `f` als die Verdoppelungsfunktion auf den ganzen Zahlen. Die Funktionsdefinitionen in den beiden Programmen sind zwar syntaktisch gleich, aber in der Funktionsdefinition von Programm `P2` gilt die zweite Deklaration der Variablen `a`, die die erste überschattet (siehe Abschnitte 2.7, 4.2.4, 4.2.5 und 4.2.7).

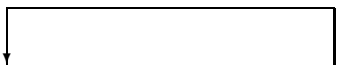
Genauso wie die Position eines Funktionsausdrucks in einem Programm (etwa `P1` oder `P2`) den Wert dieses Funktionsausdrucks bestimmt, bestimmt in einem Laufzeitsystem (etwa ein Auswerter) die Position der Funktion in der Umgebung das Ergebnis der Funktion. Die Positionen im (statischen) Programmtext entsprechen also unmittelbar den Positionen in der (dynamischen) Umgebung.

Im Fall der vorangehenden Beispiele kann man sich die erzeugten Umgebungen wie folgt veranschaulichen:

Umgebung für die Deklarationen von Programm `P1`:

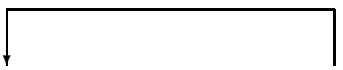


Umgebung für die Deklarationen von Programm `P2`:



Der Wert von `f`, also die zweite Komponente des Paars `(f, ...)`, muss im wesentlichen aus dem Funktionsausdruck der Deklaration von `f` bestehen, sowie zusätzlich aus einem Verweis auf die Umgebung, die bei Anwendung der Funktion gelten soll. Dieser Verweis ist jeweils durch den Pfeil veranschaulicht. Er zeigt auf die Stelle der Umgebung, an der das Paar `(f, ...)` selbst in der Umgebung steht.

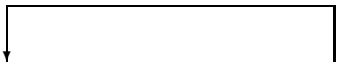
Betrachten wir nun die Funktionsanwendung `f 3`. Im Fall des Programms `P1` enthält der Wert von `f` einen Verweis auf die Umgebung



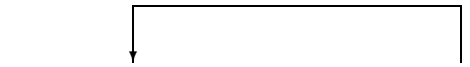
die zur Bindung des formalen Parameters `x` an den aktuellen Parameter `3` wie folgt erweitert wird:

In dieser Umgebung wird dann der Rumpf der Funktion, also `a * x` ausgewertet, was den Wert `3` ergibt.

Im Fall des Programms `P2` enthält der Wert von `f` einen Verweis auf die Umgebung



die zur Bindung des formalen Parameters `x` an den aktuellen Parameter `3` wie folgt erweitert wird:



In dieser Umgebung wird dann der Rumpf der Funktion, also `a * x` ausgewertet, was den Wert `6` ergibt.

Auf diese Weise wird das Prinzip der statischen Bindung realisiert.

10.3.2 Darstellung von SMalL-Werten und SMalL-Umgebungen in SML

SMalL-Umgebung

Eine SMalL-Umgebung wird dargestellt als eine SML-Liste von Bindungen.

SMalL-Bindung: `Bdg(name, wert)`

Eine SMalL-Bindung wird dargestellt als SML-Ausdruck der Gestalt `Bdg(name,wert)`, wobei `name` vom SML-Typ Zeichenfolge (`string`) ist und `wert` die SML-Darstellung eines SMalL-Werts ist.

SMalL-Wert „ganze Zahl": `IntVal n`

Ein SMalL-Wert vom Typ „ganze Zahl" wird als SML-Ausdruck der Gestalt `IntVal n` dargestellt, wobei `n` vom SML-Typ `int` ist.

Die Unterscheidung zwischen SMalL-Ausdrücken vom Typ „ganze Zahl" und SMalL-Werten vom Typ „ganze Zahl", also zwischen `IntExp n` und `IntVal n`, ist nicht zwingend erforderlich. Sie erlaubt aber, die statische Typprüfung von SML besser zu verwenden, was die Entwicklung, Überprüfung und Wartung des Auswerters für SMalL erleichtert.

SMalL-Wert „Funktion": `FnVal(par, body, fn_env)`

Ein SMalL-Wert vom Typ „Funktion" wird als SML-Ausdruck der Gestalt `FnVal(par, body, fn_env)` dargestellt, wobei

- `par` vom SML-Typ Zeichenfolge (`string`) ist und den Namen des (einzigen) formalen Parameters repräsentiert,
- `body` die SML-Darstellung eines SMalL-Audrucks in abstrakter Syntax ist und den Rumpf der Funktion repräsentiert,
- `fn_env` ein Verweis auf die Umgebung ist, die bei Anwendungen der Funktion gelten soll. Dieser Verweis wird durch eine SML-Referenz realisiert — zumindest in den ersten Versionen des Auswerters. Eine spätere Version des Auswerters wird diesen Verweis mit rein funktionalen Hilfsmitteln realisieren (siehe Abschnitt 10.8).

Beispiele

Darstellung der Umgebung für die Deklarationen des obigen Programms `P1`:

```
          [
           Bdg("a", IntVal 2),
fn_env1 -> Bdg("f", FnVal("x", BinOp(Mult, Var "a", Var "x"), fn_env1)),
           Bdg("a", IntVal 1)
          ]
```

wobei `fn_env1` ein Verweis (d.h. eine SML-Referenz) ist. Wenn die obige Umgebung mit `U1` bezeichnet wird, dann gilt

```
!fn_env1 = tl U1
```

Darstellung der Umgebung für die Deklarationen des obigen Programms `P2`:

```
           [
fn_env2 ->  Bdg("f", FnVal("x", BinOp(Mult, Var "a", Var "x"), fn_env2)),
            Bdg("a", IntVal 2),
            Bdg("a", IntVal 1)
           ]
```

wobei `fn_env2` ein Verweis (d.h. eine SML-Referenz) ist. Wenn die obige Umgebung mit `U2` bezeichnet wird, dann gilt

```
!fn_env2 = U2
```

10.3.3 SML-Typdeklarationen für SMalL-Werte und SMalL-Umgebungen

SMalL-Werte werden durch SML-Ausdrücke vom SML-Typ `value` repräsentiert und SMalL-Umgebungen durch SML-Listen vom SML-Typ `environment`:

```
datatype value   = IntVal of int
                 | FnVal  of string * expression * (binding list ref)

     and binding = Bdg    of string * value

type environment = binding list
```

Dabei ist `expression` der oben eingeführte SML-Typ zur Darstellung von SMalL-Ausdrücken.

Es wäre naheliegend, den Typ des Konstruktors `FnVal` in der Form `string * expression * (environment ref)` zu schreiben. Leider ermöglicht SML keine wechselseitige Bezugnahme zwischen `datatype`–Deklarationen und `type`–Deklarationen — das `and`–Konstrukt von SML verbindet mehrere `datatype`–Deklarationen miteinander oder mehrere `type`–Deklarationen miteinander, aber keine Deklarationen unterschiedlicher Art.

10.3.4 Typ des Auswerters für SMalL

Der Auswerter für SMalL ist eine zweistellige Funktion namens `eval`, die als Aufrufparameter ein Paar `(exp, env)` erhält, wobei

- `exp` ein SMalL-Ausdruck in abstrakter Syntax ist, d.h. ein SML-Ausdruck vom SML-Typ `expression`,
- `env` eine SMalL-Umgebung ist, d.h. eine SML-Liste vom SML-Typ `environment`,

und als Wert ein Paar `(val, env')` liefert, wobei

- `val` ein SMalL-Wert ist, d.h. ein SML-Ausdruck vom SML-Typ `value`,
- `env'` eine SMalL-Umgebung ist, d.h. eine SML-Liste vom SML-Typ `environment`.

Enthält der SMalL-Ausdruck `exp` keine (nicht-lokale) Deklarationen, so sind die Umgebungen `env` und `env'` identisch.

Enthält der SMalL-Ausdruck `exp` (nicht-lokale) Deklarationen, so sind die Umgebungen `env` und `env'` unterschiedlich: `env'` erweitert `env` um die Bindungen, die sich aus den Deklarationen in `exp` ergeben.

Der Typ des Auswerters `eval` für SMalL lautet also:

```
expression * environment -> value * environment
```

10.4 Ein Auswerter für SMalL: Programm `eval1`

Im Folgenden werden verschiedene Versionen des Auswerters `eval` vorgestellt, die durch schrittweise Verfeinerung entwickelt werden. Um diese Versionen zu unterscheiden, werden sie nummeriert. Die erste Version des Auswerters heißt also `eval1`. Das gesamte Programm ist in der Datei `eval1.sml` zu finden. Tests für diesen Auswerter finden sich in der Datei `eval-tests.sml`.[2]

Außer den bereits vorgestellten Typ-Deklarationen besteht der Auswerter lediglich aus der Definition der Funktion `eval1`. Diese Definition ist als Fallunterscheidung aufgebaut mit je einem Fall pro Konstruktor des Typs `expression`.

10.4.1 Auswertung von ganzen Zahlen

```
eval1(IntExp n, env) = (IntVal n, env : environment)
```

Die Auswertung eines Zahlausdrucks ergibt den entsprechenden Zahlwert, und die neue Umgebung ist gleich der alten.

10.4.2 Auswertung von unären Operationen über ganzen Zahlen

```
eval1(UnOp(op1,e), env) = let fun eval_op Min = ~
                              | eval_op Abs = abs
                          val un_op       = eval_op op1
                          val (v, _)      = eval1(e, env)
                      in
                         case v
                           of IntVal n
                                => (IntVal(un_op n), env)
                      end
```

Die Hilfsfunktion `eval_op` ordnet jedem unären SMalL-Operator einen unären SML-Operator zu. Der im SMalL-Ausdruck verwendete unäre SMalL-Operator `op1` wird mit dieser Hilfsfunktion auf den entsprechenden SML-Operator `un_op` abgebildet. Dann wird der Teilausdruck `e` des SMalL-Ausdrucks in der Umgebung `env` ausgewertet. Wenn der dabei erhaltene Wert `v` eine ganze Zahl ist, muss nur noch `un_op` auf diese ganze Zahl angewandt werden. Der dabei erhaltene Wert, zusammen mit derselben Umgebung `env`, in der der gesamte Ausdruck ausgewertet wurde, ist dann das Gesamtergebnis. Falls `e` Deklarationen enthält, können diese zwar im rekursiven Aufruf neue Bindungen erzeugen, aber mit dem rekursiven Aufruf endet auch die Gültigkeit etwaiger neuer Bindungen. Deshalb ist die vom rekursiven Aufruf zurückgelieferte Umgebung irrelevant.

Das `case`–Konstrukt dient zum Prüfen, ob `v` eine ganze Zahl ist, also die Gestalt `Intval n` hat, und zur Extraktion von `n` aus `v` mit Hilfe von Pattern Matching. Das ist nur durch Pattern Matching möglich und zum Beispiel nicht in der Form `if v ...then ...`.

Hat der Wert `v` von `e` nicht die Gestalt `IntVal n`, so liegt eine Fehler vor. Im nächsten Abschnitt wird besprochen, wie der Auswerter `eval1` zur Behandlung solcher Fehler in SMalL-Programmen erweitert werden kann.

[2] Alle in diesem Kapitel erwähnten Dateien können von der WWW-Seite des Vorlesungsskriptes heruntergeladen werden.

Die Verwendung von Pattern Matching bringt mit sich, dass die Compilierung des Auswerters Warnungen des SML-Systems erzeugt, weil das `case`–Konstrukt nicht alle Fälle des SML-Typs `value` abdeckt. Diese Warnungen sind jedoch keine Fehlermeldungen und verhindern nicht, den Auswerter für SMalL zu verwenden.

In den meisten der folgenden Abschnitte werden `case`–Konstrukte ähnlich wie im vorangehenden Programmteil verwendet, um auf Teilausdrücke eines zusammengesetzten SML-Objekts vom SML-Typ `value` zuzugreifen.

Dass die vordefinierten arithmetischen Operatoren der auszuführenden Programmiersprache SMalL einfach auf entsprechende Operatoren der ausführenden Sprache SML zurückgeführt werden, ist der übliche Ansatz zur Implementierung von Programmiersprachen.

10.4.3 Auswertung binärer Operationen über ganzen Zahlen

```
eval1(BinOp(op2,e1,e2), env)
                        = let fun eval_op Add  = op +
                                | eval_op Sub  = op -
                                | eval_op Mult = op *
                                | eval_op Div  = op div
                                | eval_op Mod  = op mod
                              val bin_op       = eval_op op2
                              val (v1, _)      = eval1(e1, env)
                              val (v2, _)      = eval1(e2, env)
                          in
                             case (v1,        v2       )
                               of (IntVal n1, IntVal n2)
                                    => (IntVal(bin_op(n1,n2)), env)
                          end
```

Dieser Fall ist völlig analog zum Fall der unären Operationen aufgebaut, nur dass es eben zwei Teilausdrücke `e1`, `e2` gibt statt einen Teilausdruck `e`. Beide werden ausgewertet zu Werten `v1`, `v2`. Wenn beide die richtige Gestalt haben, also die Zahlen `n1`, `n2` daraus extrahiert werden können, wird der entsprechende SML-Operator auf diese Zahlen angewandt.

10.4.4 Auswertung von Verzweigungen

```
eval1(If(Test(opc,e1,e2), e_then, e_else), env)
                        = let fun eval_op ==     = op =
                                | eval_op >>     = op >
                                | eval_op <<     = op <
                                | eval_op(Not t) = not o (eval_op t)
                              val comp_op        = eval_op opc
                              val (v1, _)        = eval1(e1, env)
                              val (v2, _)        = eval1(e2, env)
                          in
                             case (v1,        v2       )
                               of (IntVal n1, IntVal n2)
                                    => if comp_op(n1,n2)
                                       then eval1(e_then, env)
                                       else eval1(e_else, env)
                          end
```

Auch dieser Fall ist ziemlich ähnlich zu den beiden vorigen. Nur ist diesmal der Rumpf des `case`–Konstrukts etwas komplexer als vorher. Hier wird einfach das vordefinierte `if-then-else`–Konstrukts von SMalL auf das `if-then-else`–Konstrukt der Implementierungssprache SML zurückgeführt, so wie auch die Operatoren von SMalL auf Operatoren von SML zurückgeführt werden.

10.4.5 Auswertung von Variablen (oder Namen oder Bezeichnern)

```
| eval1(Var name, env) = let fun eval_var(name, env) =
                                 case env
                                   of Bdg(id,value)::env_tl
                                         => if name = id
                                            then value
                                            else eval_var(name, env_tl)
                           in
                               (eval_var(name,env), env)
                           end
```

Die Auswertung eines SMalL-Ausdrucks der Gestalt `Var name` erfolgt dadurch, dass die Umgebung `env` von ihrem Anfang her durchsucht wird, bis der erste Eintrag `Bdg(name, value)` gefunden wird. Dann wird der Wert `value` zusammen mit der unveränderten Umgebung `env` geliefert.

10.4.6 Auswertung von Deklarationen

```
eval1(Dcl(id,e),   env) = let val (v, _) = eval1(e, env)
                          in
                             case v
                               of FnVal(par, body, fn_env)
                                    => let
                                            val fn_env' = Bdg(id,v)
                                                          :: !fn_env
                                       in
                                            fn_env := fn_env'
                                            ;
                                            (v, Bdg(id,v)::env)
                                       end
                                | _ => (v, Bdg(id,v)::env)
                          end
```

Die Auswertung einer Deklaration besteht darin, dass die Umgebung `env` (an ihrem Anfang) um eine Bindung erweitert wird. Diese neue Bindung ordnet dem Namen `id` den Wert `v` des Ausdrucks `e` zu. Das Gesamtergebnis ist also in jedem Fall das Paar `(v, Bdg(id,v)::env)`.

Falls `v` aber ein Funktionswert ist, muss dafür gesorgt werden, dass die Umgebung, die bei Anwendungen der Funktion gelten soll, ebenfalls um die Bindung `Bdg(id,v)` erweitert wird — es könnte ja sein, dass die Funktion rekursiv ist. Diese Umgebung ist in `v` als Inhalt der SML-Referenz `fn_env` zugänglich. Die erweiterte Umgebung `fn_env'` wird also unter anderem durch Anwendung des Dereferenzierungsoperators „`!`“ berechnet, und mit der

Zuweisung `fn_env := fn_env'` wird der Inhalt der SML-Referenz durch die erweiterte Umgebung ersetzt.

Man beachte die Verwendung des Sequenzierungskonstrukts von SML, damit diese Zuweisung erfolgt, bevor das Paar `(v, Bdg(id,v)::env)` als Wert geliefert wird. Wie immer bei der Verwendung von imperativen Konstrukten muss man hier die Nebeneffekte beachten. Die Zuweisung verändert den Inhalt von `fn_env`, das ja Bestandteil von `v` ist, also wird implizit auch `v` mitverändert. Es wäre vielleicht klarer, dem veränderten `v` einen Namen `v'` zu geben und dann `(v', Bdg(id,v')::env)` zu schreiben.

Abschnitt 10.8 behandelt eine andere Implementierung, in der `fn_env` keine SML-Referenz ist und nur noch rein funktionale Hilfsmittel benötigt werden.

10.4.7 `val`–Funktionsdeklarationen versus `val-rec`–Funktionsdeklarationen

Die hier vorgenommene Behandlung von SMalL–Funktionsdeklarationen entspricht den `val-rec`–Deklarationen von SML. Zur Implementierung der SMalL–Funktionsdeklarationen entsprechend den `val`–Deklarationen von SML braucht nur die Zuweisung

```
fn_env := fn_env'
```

weggelassen zu werden. Natürlich kann man dann auch die gesamte Deklaration von `fn_env'` weglassen.

Der Unterschied zwischen `val-rec`– und `val`–Deklarationen in SML ist aus dem Vergleich der folgenden Beispiele gut erkennbar:

Beispiel 1:

```
- val f = fn x => x;
val f = fn : 'a -> 'a

- val f = fn x => f x;
val f = fn : 'a -> 'a

- f 1;
val it = 1 : int
```

Beispiel 2:

```
- val f = fn x => x;
val f = fn : 'a -> 'a

- val rec f = fn x => f x;
val f = fn : 'a -> 'b

- f 1;
<Die Auswertung terminiert nicht>
```

Die unterschiedliche Behandlungen des Verweises `fn_env` erklärt die unterschiedlichen Abläufe in den beiden Fällen.

10.4.8 Auswertung von Funktionsausdrücken

```
eval1(FnExp(par,e),env) = (FnVal(par, e, ref env), env)
```

Ein Ausdruck der Gestalt `FnExp(par,e)` steht für eine Funktion mit formalem Parameter `par` und Rumpf `e`. Der Wert dieses Ausdrucks enthält, wie in Abschnitten 10.3.1 und 10.3.2 erläutert, die gleichen Bestandteile und zusätzlich einen Verweis auf die Umgebung, die bei Anwendungen der Funktion gelten soll.

Dieser Verweis, also ein SML-Referenz, wird hier mit `ref env` erzeugt.

Falls der `FnExp`--Ausdruck für eine anonyme Funktion steht, ist die Umgebung `env`, in der der `FnExp`–Ausdruck ausgewertet wird, auch die Umgebung, die bei Anwendungen der Funktion gelten soll.

Falls der `FnExp`–Ausdruck als zweiter Teilausdruck in einem `Dcl`–Ausdruck vorkommt, ist die Funktion nicht anonym. Dann ist die Umgebung, die bei Anwendungen der Funktion gelten soll, die Umgebung `env` erweitert um die Bindung, die durch den `Dcl`–Ausdruck erzeugt wird. Diese Erweiterung wird bei der Auswertung des `Dcl`–Ausdrucks durch die Zuweisung an die hier erzeugte Referenz vorgenommen.

10.4.9 Auswertung von Funktionsanwendungen

```
eval1(App(e1,e2), env) = let val (v1, _) = eval1(e1, env)
                         in
                            case v1
                              of FnVal(par, body, fn_env)
                                 => let
                                       val (v2,_) = eval1(e2, env)
                                       val env'   = Bdg(par,v2)
                                                    :: !fn_env
                                       val (v, _) = eval1(body, env')
                                    in
                                        (v, env)
                                    end
                         end
```

Zunächst wird `e1` ausgewertet, dessen Wert `v1` eine Funktion sein muss. Ist das der Fall, wird `e2` ausgewertet, dessen Wert `v2` als aktueller Parameter an die Funktion übergeben werden muss. Die Parameterübergabe erfolgt dadurch, dass die Umgebung, die bei Anwendungen der Funktion gelten soll, also der Inhalt der Referenz `fn_env`, am Anfang um die Bindung des formalen Parameters an den Wert des aktuellen Parameters erweitert wird. In der so erhaltenen lokalen Umgebung `env'` wird schließlich der Rumpf der Funktion ausgewertet.

Verändert sich während dieser Auswertung die Umgebung, so betrifft die Änderung nur die lokale Umgebung `env'`, aber nicht die globale Umgebung `env`. Der rekursive Aufruf von `eval1` liefert zwar die veränderte lokale Umgebung zurück, aber für die Weiterverarbeitung wird diese nicht mehr benötigt.

10.4.10 Auswertung von Sequenzen

```
eval1(Seq(e1,e2),  env) = let val (_, env1) = eval1(e1, env)
                          in
                              eval1(e2, env1)
                          end
```

Der zweite Ausdruck der Sequenz wird in der Umgebung ausgewertet, die sich aus der Auswertung des ersten Ausdrucks der Sequenz ergibt. Dies ist notwendig, weil der erste Ausdruck Deklarationen enthalten kann. Der Wert einer Sequenz ist in SMalL wie in SML der Wert des zweiten Ausdrucks in der Sequenz.

Dieser Fall ist der einzige, in dem die von einem rekursiven Aufruf des Auswerters zurückgelieferte Umgebung `env1` für die Weiterverarbeitung benötigt wird. In allen bisherigen Fällen wurde, sofern überhaupt rekursive Aufrufe von `eval1` vorkamen, die zurückgelieferte Umgebung mit dem Wildcard-Muster `_` angeglichen und somit ignoriert.

10.4.11 Abhängigkeit des Auswerters `eval1` von SML

Der Auswerter `eval1` für SMalL ist von SML abhängig, weil er in SML implementiert ist.

Der Auswerter `eval1` führt jedoch grundlegende Funktionalitäten wie die Verwaltung der Umgebung und die Funktionsanwendung nicht auf dieselben Funktionalitäten von SML zurück, sondern implementiert sie selbst.

Lediglich Hilfsmittel wie die Auswertung der Addition werden auf die entsprechenden Funktionen von SML zurückgeführt. Dies ist bei der Implementierung von Programmiersprachen üblich.

So implementiert und dadurch spezifiziert `eval1` in SML einen Auswertungsalgorithmus, der von SML weitgehend unabhängig ist.

10.4.12 Gesamtprogramm des Auswerters für SMalL

Das Gesamtprogramm des Auswerters für SMalL befindet sich in der Datei `eval1.sml`.

10.4.13 Beispiele

Laden des Auswerters in SML

```
- use "eval1.sml";
[opening eval1.sml]
eval1.sml:48.34-50.66 Warning: match nonexhaustive
          IntVal n => ...

eval1.sml:64.34-66.72 Warning: match nonexhaustive
          (IntVal n1,IntVal n2) => ...

eval1.sml:79.34-83.67 Warning: match nonexhaustive
          (IntVal n1,IntVal n2) => ...

eval1.sml:87.39-91.78 Warning: match nonexhaustive
          Bdg (id,value) :: env_tl => ...
```

```
eval1.sml:115.34-124.47 Warning: match nonexhaustive
          FnVal (par,body,fn_env) => ...
datatype unary_op = Abs | Min
datatype binary_op = Add | Div | Mod | Mult | Sub
datatype comparison_op = << | == | >> | Not of comparison_op
datatype expression
  = App of expression * expression
  | BinOp of binary_op * expression * expression
  | Dcl of string * expression
  | FnExp of string * expression
  | If of test * expression * expression
  | IntExp of int
  | Seq of expression * expression
  | UnOp of unary_op * expression
  | Var of string
datatype test = Test of comparison_op * expression * expression
datatype value
  = FnVal of string * expression * binding list ref | IntVal of int
datatype binding = Bdg of string * value
type environment = binding list
val eval1 = fn : expression * environment -> value * environment
val it = () : unit
```

Die Warnungen werden gegeben, weil die Funktion `eval1` nicht alle Ausdrücke behandelt, die nach den Typdefinitionen möglich sind, sondern lediglich diejenigen, die korrekte SMalL-Ausdrücke sind. Diese Warnungen beeinträchtigen die Nutzung des Auswerters `eval1` nicht.

Auswertung von SMalL-Ausdrücken

```
- val exp_01 =
               If(Test(<<, IntExp 7, IntExp 8),
                  IntExp 1,
                  IntExp 0);
val exp_01 = If (Test (<<,IntExp #,IntExp #),IntExp 1,IntExp 0)
             : expression

- val env_01 = [];
val env_01 = [] : 'a list

- val (val_01, Env_01) = eval1(exp_01, env_01);
val val_01 = IntVal 1 : value
val Env_01 = [] : environment
```

Die Datei `eval-tests.sml` enthält eine Reihe von SMalL-Ausdrücken und Umgebungen zusammen mit den Ergebnissen der Auswertung. Es empfiehlt sich, die SMalL-Ausdrücke aus der gegebenen abstrakten Syntax in die konkrete Syntax zu überführen und die Ergebnisse der Auswertung nachzuvollziehen.

Das System SML/NJ gibt zusammengesetzte Ausdrücke normalerweise so aus, dass ab einer gewissen Schachtelungstiefe das Zeichen # als Abkürzung für ganze Teilausdrücke steht. Mit den beiden Zuweisungen

```
Compiler.Control.Print.printDepth  := 100;
Compiler.Control.Print.printLength := 100;
```

am Anfang der Datei wird SML/NJ veranlasst, diese Abkürzungen erst bei wesentlich komplexeren Ausdrücken zu verwenden als sonst.

Eine weitere Besonderheit von SML/NJ ist die Ausgabe von zyklischen Strukturen, wie sie durch die Referenzen in `FnVal`–Ausdrücken entstehen.

```
- val exp_11 =
               Seq(Dcl("identity", FnExp("x", Var "x")),
                   App(Var "identity", IntExp 3)
                   );
val exp_11 =  Seq (Dcl ("identity",FnExp ("x",Var "x")),
                   App (Var "identity",IntExp 3))
             : expression

- val env_11 = [];
val env_11 = [] : 'a list

- val (val_11, Env_11) = eval1(exp_11, env_11);
val val_11 = IntVal 3 : value
val Env_11 = [
              Bdg("identity",
                  FnVal("x",Var "x",
                        ref [Bdg("identity",FnVal("x",Var "x",%1))
                             ] as %1)
                  )
             ]
             : environment
```

Die Umgebung `Env_11` enthält eine Bindung für den Namen „identity“, nämlich einen `FnVal`–Ausdruck, dessen dritter Teilausdruck ja eine Referenz auf `Env_11` ist. Diese Referenz wird ausgegeben als `ref` gefolgt von einer Wiederholung von `Env_11`, so dass die Gefahr einer nichtterminierenden Ausgabe entsteht. Tatsächlich ist die Struktur ja zyklisch.

SML/NJ bricht die Ausgabe des Zyklus aber nach der ersten Wiederholung ab und benutzt eine symbolische Notation: `ref [...] as %1` soll bedeuten, dass die gesamte Liste mit `%1` bezeichnet wird, so dass an der Stelle, wo in der Liste das Symbol `%1` selbst vorkommt, wieder die gesamte Liste gemeint ist.

Diese Notation ist recht gut lesbar, kann aber leider von SML/NJ nur als Ausgabe erzeugt, aber nicht eingelesen werden. Erstens ist das Zeichen `%` syntaktisch gar nicht erlaubt, zweitens ist das `as`–Konstrukt syntaktisch nicht an dieser Stelle zulässig.

Behandlung von fehlerhaften SMalL-Ausdrücken durch den Auswerter `eval1`

```
- val exp_24 =
        App(IntExp 1, IntExp 1);
val exp_24 = App (IntExp 1,IntExp 1) : expression

- val env_24 = [];
val env_24 = [] : 'a list

- val (val_24, Env_24) = eval1(exp_24, env_24);
uncaught exception nonexhaustive match failure
```

Die Datei `eval-tests.sml` enthält weitere Fälle von inkorrekten SMalL-Ausdrücken. Der Auswerter `eval1` kann nur korrekte SMalL-Ausdrücke behandeln. Dies hat zwei Folgen:

- Zum einen wird bei der Kompilierung der Funktion `eval1` festgestellt, dass Fälle wie der obige, die nach den Typdeklarationen möglich sind, von der Funktion `eval1` nicht abgedeckt werden. Dadurch wurden die erwähnten Warnungen beim Laden der Datei `eval1.sml` verursacht.
- Zum anderen führt die Auswertung von Ausdrücken, die von `eval1` nicht abgedeckt werden, zu Laufzeitfehlern. Diese werden in der Form von vordefinierten, nicht abgefangenen SML-Ausnahmen mitgeteilt (siehe Abschnitt 10.6).

10.4.14 Die Start-Umgebung

Wer versucht, einige Beispielprogramme in SMalL zu schreiben, stößt früher oder später auf eine Uneinheitlichkeit zwischen vorgegebenen SMalL-Funktionen wie `Min` oder `Add`, und Funktionen, die in der Sprache SMalL selbst definiert worden sind. Letztere können mit Hilfe des Konstruktors `App` auf den Wert eines SMalL-Ausdrucks angewandt werden, wie im obigen Beispiel:

```
App(Var "identity", IntExp 3)
```

Das unäre `Minus` lässt sich aber nicht auf diese Weise anwenden. Versuche wie

```
App(Min,      IntExp 3)
App(UnOp Min, IntExp 3)
App(Var Min,  IntExp 3)
App(Var "Min",IntExp 3)
```

ergeben entweder Typfehler oder scheitern daran, dass gar keine Bindung für den Namen „`Min`“ existiert. Die gleiche Uneinheitlichkeit äußert sich darin, dass die in der Sprache SMalL definierten Funktionen an neue Namen gebunden werden können, z.B. mit `Dcl("id", Var "identity")`, und dass sie als Parameter an andere Funktionen weitergereicht werden können, dass diese Möglichkeiten aber nicht für die vorgegebenen SMalL-Funktionen bestehen. Diese Uneinheitlichkeit lässt einen grundlegenden Entwurfsfehler im Auswerter oder gar in der Sprache SMalL befürchten.

Dem ist aber nicht so. Es ist unvermeidbar, einer auszuführenden Programmiersprache (hier SMalL) gewisse Grundfunktionen vorzugeben, die nicht in dieser Sprache selbst

definiert werden können, sondern von der ausführenden Programmiersprache (hier SML) zur Verfügung gestellt werden. Man muss dann nur noch dafür sorgen, dass die Namen dieser Grundfunktionen in der gleichen Weise zur Verfügung stehen wie die Namen von in der Sprache selbst definierten Funktionen. Dies wird durch die Vorgabe einer Start-Umgebung erreicht.
Für SMalL kann die Start-Umgebung folgendermaßen definiert werden:

```
 val start_env = [Bdg("Min",
                      FnVal("x", UnOp(Min,Var "x"),
                            ref nil)),
                  Bdg("Abs",
                      FnVal("x", UnOp(Abs,Var "x"),
                            ref nil)),
                  Bdg("Add",
                      FnVal("x", FnExp("y",BinOp(Add,Var "x",Var "y")),
                            ref nil)),
                  Bdg("Sub",
                      FnVal("x", FnExp("y",BinOp(Sub,Var "x",Var "y")),
                            ref nil)),
                  Bdg("Mult",
                      FnVal("x", FnExp("y",BinOp(Mult,Var "x",Var "y")),
                            ref nil)),
                  Bdg("Div",
                      FnVal("x", FnExp("y",BinOp(Div,Var "x",Var "y")),
                            ref nil)),
                  Bdg("Mod",
                      FnVal("x", FnExp("y",BinOp(Mod,Var "x",Var "y")),
                            ref nil))
                 ]
```

Wertet man SMalL-Ausdrücke nicht in der leeren Umgebung aus, sondern in dieser Start-Umgebung, stehen die Namen der vorgegebenen Funktionen genau so zu Verfügung, als seien sie in SMalL selbst definiert worden:

```
- eval1( App(Var "Min", IntExp 1), start_env );
val it =
  (IntVal ~1,
   [Bdg("Min",FnVal("x",UnOp(Min,Var "x"),ref [])),
    Bdg("Abs",FnVal("x",UnOp(Abs,Var "x"),ref [])),
    Bdg("Add",FnVal("x",FnExp("y",BinOp(Add,Var "x",Var "y")),ref [])),
    Bdg("Sub",FnVal("x",FnExp("y",BinOp(Sub,Var "x",Var "y")),ref [])),
    Bdg("Mult",FnVal("x",FnExp("y",BinOp(Mult,Var "x",Var "y")),ref [])),
    Bdg("Div",FnVal("x",FnExp("y",BinOp(Div,Var "x",Var "y")),ref [])),
    Bdg("Mod",FnVal("x",FnExp("y",BinOp(Mod,Var "x",Var "y")),ref []))])
  : value * environment

- eval1( App(App(Var "Mult",IntExp 2), IntExp 3), start_env );
val it =
  (IntVal 6,
```

```
   [Bdg("Min",FnVal("x",UnOp(Min,Var "x"),ref [])),
    Bdg("Abs",FnVal("x",UnOp(Abs,Var "x"),ref [])),
    Bdg("Add",FnVal("x",FnExp("y",BinOp(Add,Var "x",Var "y")),ref [])),
    Bdg("Sub",FnVal("x",FnExp("y",BinOp(Sub,Var "x",Var "y")),ref [])),
    Bdg("Mult",FnVal("x",FnExp("y",BinOp(Mult,Var "x",Var "y")),ref [])),
    Bdg("Div",FnVal("x",FnExp("y",BinOp(Div,Var "x",Var "y")),ref [])),
    Bdg("Mod",FnVal("x",FnExp("y",BinOp(Mod,Var "x",Var "y")),ref []))])
  : value * environment

- eval1( Seq(Dcl("identity", FnExp("x", Var "x")),
             App(Var "identity", Var "Div")),
         start_env );
val it =
  (FnVal("x",FnExp("y",BinOp(Div,Var "x",Var "y")),ref []),
   [Bdg("identity",FnVal("x",Var "x",ref #)),
    Bdg("Min",FnVal("x",UnOp(Min,Var "x"),ref [])),
    Bdg("Abs",FnVal("x",UnOp(Abs,Var "x"),ref [])),
    Bdg("Add",FnVal("x",FnExp("y",BinOp(Add,Var "x",Var "y")),ref [])),
    Bdg("Sub",FnVal("x",FnExp("y",BinOp(Sub,Var "x",Var "y")),ref [])),
    Bdg("Mult",FnVal("x",FnExp("y",BinOp(Mult,Var "x",Var "y")),ref [])),
    Bdg("Div",FnVal("x",FnExp("y",BinOp(Div,Var "x",Var "y")),ref [])),
    Bdg("Mod",FnVal("x",FnExp("y",BinOp(Mod,Var "x",Var "y")),ref []))])
  : value * environment
```

Auch SML startet bekanntlich nicht in einer völlig leeren Umgebung, sondern gibt Bindungen für gewisse Namen durch seine Start-Umgebung vor:

```
linux% sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
- ~;
val it = fn : int -> int
```

Der Name ~ steht also von Anfang an so zur Verfügung, als sei die zugehörige Funktion in SML selbst definiert. Dass das zumindest in SML/NJ nicht der Fall ist, erkennt man daran, dass das Überschatten des Namens ~ in SML/NJ nicht ganz wie erwartet funktioniert:

```
- ~ 1;
val it = ~1 : int                (* OK *)
- ~1;
val it = ~1 : int                (* OK *)

- val ~ = fn x => x;
val ~ = fn : 'a -> 'a            (* ~ ueberschatten *)

- ~ 1;
val it = 1 : int                 (* OK *)
- ~1;
val it = ~1 : int                (* ?? *)
```

10.5 Behandlung fehlerhafter SMalL-Ausdrücke — `eval2` und `eval3`

10.5.1 Schwäche des Auswerters `eval1` für SMalL

Der Auswerter für SMalL des vorangehenden Abschnitts ist gegen fehlerhafte SMalL-Ausdrücke wie etwa

```
App(IntExp 1, IntExp 1)
```

oder

```
UnOp(Min, FnExp("x", Var "x"))
```

nicht gewappnet. Der erste Ausdruck ist inkorrekt, weil der Wert des ersten Parameters einer Funktionsanwendung eine Funktion sein muss. Der zweite Ausdruck ist inkorrekt, weil der unäre Operator `Min` nur auf ganzen Zahlen aber nicht auf Funktionen definiert ist.

Wird der Auswerter `eval1` auf einen fehlerhaften SMalL-Ausdruck angewendet, so führt dies zu einem Laufzeitfehler:

```
- eval1(UnOp(Min, FnExp("x", Var "x")), []);
uncaught exception nonexhaustive match failure
```

Die Fehlermeldungen, die das Laufzeitsystem von SML liefert, bringen die Schwäche des Auswerters für SMalL auf den Punkt: Kein Fall in der Deklaration der SML-Funktion `eval1` behandelt fehlerhafte Parameter, d.h., fehlerhafte SMalL-Ausdrücke. Der in Abschnitt 10.4.2 beschriebene Fall zur Auswertung von unären Operationen über ganzen Zahlen:

```
eval1(UnOp(op1,e), env) = let fun eval_op Min = ~
                              | eval_op Abs = abs
                          val un_op       = eval_op op1
                          val (v, _)      = eval1(e, env)
                      in
                         case v
                           of IntVal n
                                => (IntVal(un_op n), env)
                      end
```

enthält keine Behandlung für den Fall, dass der Wert `v` des Ausdrucks `e` nicht die erwartete Gestalt `IntVal n` sondern die unerlaubte Gestalt `FnVal(...)` hat.

10.5.2 Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2`

Der Auswerter `eval1` kann dadurch verbessert werden, dass dem Typ `value` weitere Werte und den `case`–Konstrukten zur Zerlegung der SMalL-Ausdrücke weitere Fälle hinzugefügt werden. Diese verbesserte Version des Auswerters heißt `eval2` und ist in der Datei `eval2.sml` definiert.

Wenn der Auswerter für SMalL verwendet wird, können zwei grundlegende Laufzeitfehler auftreten:

1. Ein SMalL-Ausdruck, d.h. ein SML-Ausdruck vom Typ `expression`, ist nicht korrekt im Sinne der Programmiersprache SMalL — wie `App(IntExp 1, IntExp 1)` oder `UnOp(Min, FnExp("x", Var "x"))`.
2. Ein SMalL-Variable hat in der Umgebung keine Bindung — wie bei der Auswertung von `UnOp(Min, Var "a")` in der leeren Umgebung.

Es bietet sich also an, zwei unterschiedliche zusätzliche Werte zu verwenden, um die beiden Fälle zu unterscheiden und so dem Benutzer eine verständliche Fehlermeldung zu geben. Die Ergänzung des SML-Typs `value` um weitere Werte kann wie folgt geschehen (die Ergänzung findet sich in der Datei `eval2.sml`):

```
datatype value  = IntVal of int
                | FnVal  of string * expression * (binding list ref)
                | Error_illegal_expression
                | Error_unbound_variable
```

Die beiden letzten Zeilen sind gegenüber der ersten Version des Auswerters neu.

Der Fall zur Auswertung von Variablen wird wie folgt erweitert:

```
eval2(Var name, env) = let fun eval_var(name, env) =
                                case env
                                 of Bdg(id,value)::env_tl
                                        => if name = id
                                           then value
                                           else eval_var(name, env_tl)
                                  | nil => Error_unbound_variable
                           in
                               (eval_var(name,env), env)
                           end
```

Der einzige Unterschied zu `eval1` ist der zweite Fall im `case`–Konstrukt.

Der Fall zur Auswertung von unären Operationen über ganzen Zahlen wird wie folgt erweitert:

```
eval2(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval2(e, env)
                          in
                            case v
                              of IntVal n
                                   => (IntVal(un_op n), env)
                               | _ => (Error_illegal_expression, env)
                          end
```

Der einzige Unterschied zu `eval1` ist wieder der zweite Fall im `case`–Konstrukt.

In allen anderen Fällen, in denen eine Erweiterung notwendig ist, ist die Erweiterung von der gleichen Art wie in diesem Fall.

10.5.3 Veränderter Auswerter für SMalL mit Sonderwerten

Das veränderte Programm findet sich in der Datei `eval2.sml`.

Gegenüber `eval1` sind lediglich die beiden neuen Zeilen zur Deklaration der Sonderwerte des Datentyps `value` hinzugekommen sowie jeweils eine neue Zeile pro `case`–Konstrukt, die einen Fangfall einführt, in dem einer der Sonderwerte geliefert wird.

Die nicht fehlerhaften SMalL-Ausdrücke in der Datei `eval-tests.sml` führen mit `eval2` zu genau den gleichen Ergebnissen wie mit `eval1`.

10.5.4 Unzulänglichkeit des veränderten Auswerters mit Sonderwerten

In einigen Fällen fehlerhafter SMalL-Ausdrücke verhält sich `eval2` einigermaßen zufriedenstellend:

```
- eval2(App(IntExp 1, IntExp 1), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(UnOp(Min, FnExp("x", Var "x")), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(Var "a", []);
val it = (Error_unbound_variable,[]) : value * environment
```

Die Fehlermeldung beschreibt jeweils zutreffend die Fehlerursache. Nicht ganz zufriedenstellend ist, dass die Fehlermeldungen als Teilausdrücke von zusammengesetzten Ausdrücken erscheinen. Die Fehlermeldung

```
Error_illegal_expression
```

wäre sicherlich der Fehlermeldung

```
(Error_illegal_expression,[])
```

vorzuziehen. Dafür zu sorgen, dass in allen Fehlerfällen nur die lesbarere Fehlermeldung als Ergebnis von `eval2` geliefert wird, erweist sich aber als erstaunlich schwierig, wenn überhaupt möglich, und verlangt vor allem eine grundlegende Veränderung des ursprünglichen Programms. Wie es auch in der Praxis oft der Fall ist, überwiegt der Nachteil des Veränderungsaufwands die Nachteile der etwas unzufriedenstellenden Fehlermeldungen, die deshalb weiterhin in Kauf genommen werden.

Aber in anderen Fällen sind die Fehlermeldungen von `eval2` einfach irreführend:

```
- eval2(UnOp(Min, Var "a"), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(BinOp(Add, BinOp(Add, Var "a", Var "b"), Var "c"), []);
val it = (Error_illegal_expression,[]) : value * environment
```

Die Fehlerursache ist in beiden Fällen eine fehlende Variablenbindung, so dass die Fehlermeldung `Error_unbound_variable` angebracht wäre.

Der Grund für die irreführende Fehlermeldung `Error_illegal_expression` liegt darin, dass die Sonderwerte rekursiv weitergereicht werden. Wenn ein rekursiver Aufruf von `eval2` den Sonderwert `Error_unbound_variable` liefert, dann führt dieser Wert in der nächsthöheren Rekursionsstufe in den Fangfall des `case`–Konstrukts, wo dann der Sonderwert `Error_illegal_expression` zurückgeliefert wird.

Offenbar ist dieses Verhalten ohne wesentliche Veränderungen der Funktion `eval2` nicht zu beheben. Solche Veränderungen müssten sicherstellen, dass in jedem Fall der Definition von `eval2` jeder mögliche Sonderwert abgefangen und wunschgemäß weitergereicht wird.

Der Fall zur Auswertung von unären Operationen über ganzen Zahlen könnte etwa wie folgt verändert werden:

```
eval2(UnOp(op1,e), env) = let fun eval_op Min = ~
                            | eval_op Abs = abs
                          val un_op       = eval_op op1
                          val (v, _)      = eval2(e, env)
                      in
                        case v
                          of IntVal n
                               => (IntVal(un_op n), env)
                           | Error_unbound_variable
                               => (v, env)
                           | Error_illegal_expression
                               => (v, env)
                           | _ => (Error_illegal_expression, env)
                      end
```

Für jeden Sonderwert wäre also ein Fall im `case`–Konstrukt hinzugekommen. Das sieht noch einigermaßen überschaubar aus. Aber in anderen Fällen der Definition von `eval2`, zum Beispiel im Fall zur Auswertung von binären Operationen über ganzen Zahlen, wären die entsprechenden Veränderungen schon umfangreicher:

```
eval2(BinOp(op2,e1,e2), env)
                       = let fun eval_op Add  = op +
                               | eval_op Sub  = op -
                               | eval_op Mult = op *
                               | eval_op Div  = op div
                               | eval_op Mod  = op mod
                           val bin_op       = eval_op op2
                           val (v1, _)      = eval2(e1, env)
                           val (v2, _)      = eval2(e2, env)
                         in
                           case (v1,        v2       )
                             of (IntVal n1, IntVal n2)
                                  => (IntVal(bin_op(n1,n2)), env)
                              | (Error_unbound_variable, _)
                                  => (v1, env)
                              | (Error_illegal_expression, _)
```

```
                                    => (v1, env)
                                | (_, Error_unbound_variable)
                                    => (v2, env)
                                | (_, Error_illegal_expression)
                                    => (v2, env)
                                | _ => (Error_illegal_expression, env)
                        end
```

Da nicht nur die beiden hier exemplarisch ausgeführten Fälle der Definition von `eval2` entsprechend verändert werden müssten, sondern sämtliche Fälle außer denen für `IntExp`, `Var`, `FnExp` und `Seq`, würde das Programm durch die Veränderung um etwa ein Drittel länger — und das bei nur zwei Sonderwerten. Für aussagekräftigere Fehlermeldungen wären aber deutlich mehr Sonderwerte erforderlich. Dann würde die ursprünglich so einfache Definition des Auswerters durch die Anzahl der zu berücksichtigenden Kombinationen von Sonderwerten derart aufgebläht, dass sie völlig unübersichtlich würde.

Die Veränderung des Programms wäre also sehr wesentlich, obwohl die Logik der Veränderung ziemlich einfach und für alle Fälle des Programms dieselbe ist. Wird ein Programm wie `eval2` in einer solchen Weise verändert, dann kann es passieren, dass der überwiegende Teil des veränderten Programms aus der Behandlung von fehlerhaften Parametern besteht. Das ist nicht zufriedenstellend.

Die Erweiterung von `eval2`, die hier skizziert wurde, wird in diesem Kapitel nicht vollständig angegeben.

10.5.5 Verbesserung des Auswerters mit SML-Ausnahmen — `eval3`

SML bietet sogenannte Ausnahmen (*exceptions*) an, um ein Programm wie den Auswerter für SMalL mit einer Behandlung von fehlerhaften Aufrufparametern zu ergänzen. Bei der Verwendung von SML-Ausnahmen sind die Veränderungen des ursprünglichen Programms minimal, so dass die Struktur und die Logik dieses Programms erhalten bleiben.

Ausnahmen sind keine Werte, sondern eine Gattung von Programmierobjekten für sich. Ausnahmen werden mit dem Konstrukt `raise` „erhoben“ oder „geworfen“, ähnlich wie Sonderwerte zur Fehlermeldung im vorangehenden Abschnitt verwendet wurden.

Die Datei `eval3.sml` enthält die Version `eval3` des Auswerters. Diese Version behandelt fehlerhafte SMalL-Ausdrücke mit Hilfe von SML-Ausnahmen.

Die Unterschiede zwischen `eval2.sml` und `eval3.sml` sind zum einen, dass statt der zwei Sonderwerte zwei SML-Ausnahmen deklariert werden. Statt:

```
datatype value  = IntVal of int
                | FnVal  of string * expression * (binding list ref)
                | Error_illegal_expression
                | Error_unbound_variable
```

lauten die Deklarationen jetzt

```
datatype value  = IntVal of int
                | FnVal  of string * expression * (binding list ref)
exception illegal_expression
exception unbound_variable
```

Zum anderen unterscheiden sich die beiden Dateien darin, dass alle Fangfälle von `eval2` der Gestalt:

```
case ...
of ...
| _ => (Error_illegal_expression, env)
```

für `eval3` die Gestalt haben:

```
case ...
of ...
| _ => raise illegal_expression
```

Entsprechend wurde

```
| nil => Error_unbound_variable
```

zu

```
| nil => raise unbound_variable
```

Ansonsten sind die Definitionen von `eval2` und `eval3` identisch. Die Struktur des Auswerters `eval2`, der sich nur geringfügig von dem ursprünglichen Auswerter `eval1` unterscheidet, bleibt also völlig erhalten.

Die Verwendung von SML-Ausnahmen anstelle von Sonderwerten zur Behandlung von fehlerhaften SMalL-Ausdrücken verursacht also keine wesentliche Verlängerung des Programms. Aber das Verhalten des Auswerters ist jetzt wie gewünscht:

```
- eval3(App(IntExp 1, IntExp 1), []);
uncaught exception illegal_expression

- eval3(UnOp(Min, FnExp("x", Var "x")), []);
uncaught exception illegal_expression

- eval3(Var "a", []);
uncaught exception unbound_variable

- eval3(UnOp(Min, Var "a"), []);
uncaught exception unbound_variable

- eval3(BinOp(Add, BinOp(Add, Var "a", Var "b"), Var "c"), []);
uncaught exception unbound_variable
```

Die Fehlermeldung (in Gestalt einer Ausnahme) beschreibt jeweils zutreffend die Fehlerursache, auch in den Fällen, in denen `eval2` eine irreführende Fehlermeldung (in Gestalt eines Sonderwerts) liefert.

Die Mitteilung „`uncaught exception`“, d.h. „nicht eingefangene Ausnahme“ bezieht sich auf die Möglichkeit, Ausnahmen zu behandeln oder „einzufangen“, von der im Auswerter `eval3` kein Gebrauch gemacht wird. Die Programmierung von „Ausnahmebehandlern“ wird im nächsten Abschnitt erläutert.

Alle nicht fehlerhaften SMalL-Ausdrücke in der Datei `eval-tests.sml` führen mit `eval3` zu genau den gleichen Ergebnissen wie mit `eval2` und mit `eval1`.

10.6 Der SML-Typ `exn` („exception“)

Dieser Abschnitt behandelt allgemein die Verwendung von SML-Ausnahmen — unabhängig von Auswertern.

10.6.1 Der vordefinierte Typ `exn`

SML bietet den vordefinierten Typ `exn` (ausgesprochen *exception* oder *Ausnahme*), dessen Werte Ausnahmewerte (*exception values*) heißen.

Eine Besonderheit dieses Typs ist, dass der Programmierer dem Typ neue (Wert-)Konstruktoren hinzufügen kann. Dies ist für keinen vordefinierten Typ, und auch für keinen vom Programmierer definierten Typ, möglich.

Die neuen (Wert-)Konstruktoren des vordefinierten Typs `exn`, die in einem Programm deklariert werden können, werden Ausnahmekonstruktoren (*exception constructors*) genannt.

10.6.2 Ausnahmekonstruktoren

Ein Ausnahmekonstruktor namens `A` wird wie folgt deklariert:

```
exception A;
```

Ausnahmekonstruktoren können konstant sein wie etwa in der folgenden Deklaration aus dem Auswerter `eval3.sml`

```
exception illegal_expression;
```

oder einen Parameter von irgend einem Typ `t` haben. Ein Ausnahmekonstruktor namens `A` mit Parameter vom Typ `t` wird wie folgt deklariert:

```
exception A of t;
```

Ausnahmen können unter Verwendung des `let`–Konstrukts von SML auch lokal deklariert werden (vgl. Abschnitt 10.6.3).

10.6.3 Ausnahmen erheben (oder werfen)

Eine Ausnahme `A` wird mit dem Ausdruck

```
raise A
```

erhoben (oder geworfen).

Ein Beispiel von einer Deklaration und einer Erhebung von einer konstanten Ausnahme ist wie folgt:

```
- exception negative_integer;
exception negative_integer

- fun factorial x = if x < 0
                    then raise negative_integer
                    else if x = 0
                         then 1
                         else x * factorial(x - 1);
val factorial = fn : int -> int
```

```
- factorial 4;
val it = 24 : int

- factorial ~4;
uncaught exception negative_integer
```

Dieses Beispiel kann wie folgt in ein Beispiel verändert werden, in dem eine Deklaration und eine Erhebung einer Ausnahme mit Parameter vorkommen:

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x = if x < 0
              then raise negative_argument(x)
              else if x = 0
                   then 1
                   else x * fac(x - 1);
val fac = fn : int -> int

-  fac ~4;
uncaught exception negative_argument
```

Die Ausnahme hat den Ausnahmeparameter `~4`, der aber in der gedruckten Mitteilung des SML-Systems nicht erwähnt wird. Wie in vielen anderen Fällen verkürzt SML/NJ die gedruckte Mitteilung, so dass die eigentliche Struktur nicht vollständig sichtbar wird.

10.6.4 Ausnahmen als Werte

Ausnahmekonstruktoren werden weitgehend wie herkömmliche (Wert-)Konstruktoren verwendet (siehe Kapitel 5 und Kapitel 8). Unter anderem können Ausnahmekonstruktoren zum Aufbau von neuen Werten und zum Pattern Matching verwendet werden. So können z.B. Listen von Ausnahmewerten gebildet werden.

Ausnahmen können unter Verwendung des `let`–Konstrukts von SML auch lokal, auch zu der Definition einer rekursiven Funktion, definiert werden, wie etwa in der folgenden Funktionsdeklaration:

```
- fun f x = let
                exception invalid_argument
            in
                if x < 0
                then raise invalid_argument
                else if x = 0
                     then 1
                     else x * f(x - 1)
            end;
val f = fn : int -> int

- f ~4;
uncaught exception invalid_argument
```

Lokale Deklarationen von Ausnahmen ermöglichen, dass unterschiedliche Ausnahmen denselben Namen tragen, was zu Verständnisproblemen führen kann. Dies sollte vermieden werden, und Ausnahmen sollten so weit wie möglich nur global deklariert werden.

10.6.5 Ausnahme versus Wert

Wie Werte von herkömmlichen Typen können Ausnahmen das Ergebnis einer Auswertung sein. Dies ist z.B. der Fall, wenn die Fakultätsfunktion `factorial` aus Abschnitt 10.6.3 auf eine negative ganze Zahl angewandt wird:

```
- factorial ~4;
uncaught exception negative_integer
```

Ausnahmen werden aber während der Auswertung (siehe die Abschnitte 3.1.3 und 3.4.6) nicht wie Werte von herkömmlichen Typen behandelt, sondern wie folgt:

> Liefert die Auswertung eines Teilausdruckes `T` eines zusammengesetzten Ausdrucks `B` eine Ausnahme `A` als Ergebnis, so wird diese Ausnahme `A` als Ergebnis der Auswertung des Gesamtausdrucks `B` geliefert, es sei denn, `A` wird von einem Ausnahmebehandler eingefangen (siehe Abschnitt 10.6.6).

Die Sonderstellung von Ausnahmen während der Auswertung kann am folgenden Beispiel beobachtet werden:

```
- exception negative_integer;
exception negative_integer

- fun factorial x = if x < 0
                    then raise negative_integer
                    else if x = 0
                         then 1
                         else x * factorial(x - 1);
val factorial = fn : int -> int

- fun is_even x = let val x_mod_2 = x mod 2
                  in
                      case x_mod_2
                        of 0 => true
                         | _ => false
                  end;
val is_even = fn : int -> bool

- is_even(factorial ~4);
uncaught exception negative_integer
```

Da SML auf einer Auswertung in applikativer Reihenfolge beruht, führt die Auswertung von `is_even(factorial ~4)` zunächst zur Auswertung von `factorial ~4`. Die Auswertung von `factorial ~4` liefert als Ergebnis die Ausnahme `negative_integer`.

Würden Ausnahmewerte bei der Auswertung wie herkömmliche Werte behandelt, so müsste anschließend `is_even(negative_integer)` ausgewertet werden. Das könnte auf zwei Weisen geschehen. Erstens könnte ein Typfehler zur Laufzeit gemeldet werden, weil `is_even` einen Aufrufparameter vom Typ `int`, aber nicht `exn` erwartet. Zweitens könnte der Ausnahmewert wie ein Sonderwert verarbeitet werden, so dass `(negative_integer mod 2)` wiederum den Wert `negative_integer` liefert. Dann würde die Auswertung von

`is_even(negative_integer)` wegen des Fangfalls im `case`–Konstrukt den Wert `false` liefern.

Die Auswertung von `is_even(factorial ~4)` ergibt aber weder einen Typfehler noch den Wert `false`, sondern den Ausnahmewert `negative_integer`. Das zeigt, dass der Ausnahmewert bei der Auswertung nicht wie ein herkömmlicher Wert behandelt wurde.

10.6.6 Ausnahmen behandeln (oder einfangen)

Ein Ausnahmebehandler (*exception handler*) ist eine Art Funktion, die aber nur Parameter vom Typ `exn` haben kann. Er hat im einfachsten Fall die Gestalt `handle A => C`.

Ein Ausdruck `B` kann mit einem Ausnahmebehandler verknüpft werden zu

```
B handle A => C
```

Liefert die Auswertung von `B` (oder von einem Teilausdruck `T` in `B`) die Ausnahme `A`, so wird der Ausnahmebehandler `handle A => C` wirksam. Das bedeutet, Ausdruck `C` wird ausgewertet und der Wert von `C` als Wert von `B` geliefert. Die Ausnahme `A` wird also nicht weitergereicht.

Liefert die Auswertung von `B` etwas anderes als die Ausnahme `A`, so hat der Ausnahmebehandler `handle A => C` keine Wirkung.

Eine häufige Veranschaulichung besteht in der Vorstellung, dass ein Teilausdruck `T` tief innerhalb von `B` die Ausnahme `A` in Richtung seiner umfassenden Ausdrücke wirft. Die Ausnahme `A` wird von den umfassenden Ausdrücken von `T` einfach durchgelassen, bis sie bei `B` vom Ausnahmebehandler eingefangen wird.

Das Beispiel der Fakultätsfunktion `factorial` kann wie folgt mit einem Behandler für die Ausnahme `negative_integer` ergänzt werden:

```
- exception negative_integer;
exception negative_integer

- fun factorial x = (if x < 0
                     then raise negative_integer
                     else if x = 0
                          then 1
                          else x * factorial(x - 1)
                    )
                     handle negative_integer => factorial(~x);
val factorial = fn : int -> int

- factorial ~4;
val it = 24 : int
```

Wird die Funktion `factorial` auf eine negative ganze Zahl `x` angewandt, so wird die Ausnahme `negative_integer` erhoben. Diese Ausnahme wird vom Behandler

```
handle negative_integer => factorial(~x)
```

eingefangen, was zur Auswertung von `factorial(~(~4))` führt.
Ein Behandler für die Ausnahme `negative_argument`, die in der Fakultätsfunktion `fac` erhoben wird, kann wie folgt spezifiziert werden:

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x = if x < 0
              then raise negative_argument(x)
              else if x = 1
                   then 1
                   else x * fac(x - 1);
val fac = fn : int -> int

- fac ~4 handle negative_argument(y) => fac(~y);
val it = 24 : int
```

Obwohl in beiden Beispielen die Anwendung der Fakultätsfunktion auf eine negative ganze Zahl `z` zur Berechnung der Fakultät des Betrags `~z` dieser Zahl führt, geschieht dies in den beiden Beispielen in unterschiedlichen Weisen:

- Die Ausnahme, die erhoben wird, wird im ersten Beispiel innerhalb des Rumpfes der Fakultätsfunktion `factorial` eingefangen, d.h. im Geltungsbereich des formalen Parameters `x` der Funktion `factorial`.
- Im zweiten Beispiel ist der Behandler außerhalb des Geltungsbereiches des formalen Parameters `x` der Fakultätsfunktion `fac` definiert. Der Wert `~4` des aktuellen Parameters des Aufrufes wird über die einstellige Ausnahme `negative_argument` an den Behandler

  ```
  negative_argument(y) => fac(~y)
  ```

  weitergereicht.

Die allgemeine Form der Definition eines Behandlers ist wie folgt:

```
handle <Muster1> => <Ausdruck1>
     | <Muster2> => <Ausdruck2>
                  .
                  .
                  .
     | <Mustern> => <Ausdruckn>
```

d.h. ein Behandler besteht aus einem Angleichmodell (siehe Abschnitt 9.2.3). Wie immer, wenn Pattern Matching verwendet wird, muss auf Schreibfehler in Mustern geachtet werden. Kommt z.B. statt eines Ausnahmenamens

```
illegal_expression
```

in einem Muster fälschlich ein Bezeichner wie

```
ilegal_expression
```

vor, so wird der Bezeichner als ungebundener Name (oder Variable) ausgelegt, der während des Pattern Matching an jede mögliche Ausnahme gebunden werden kann.

10.6.7 Prinzip der Auswertung von `raise`– und `handle`–Ausdrücken

Die Auswertung eines Ausdrucks

```
raise <Ausdruck>
```

führt zur Auswertung von `<Ausdruck>`, der ein Ausdruck vom Typ `exn` sein muss. Liefert die Auswertung von `<Ausdruck>` einen Ausnahmewert `<Ausnahme>`, so ist der Wert des Ausdrucks

```
raise <Ausdruck>
```

das sogenannte Ausnahmepaket `[ <Ausnahme> ]`. Der Begriff Ausnahmepaket dient hauptsächlich dazu, Ausnahmewerte von herkömmlichen Werten zu unterscheiden. Die Schreibweise `[...]` ist keine SML-Notation (und insb. stellt es keine Liste dar).

Bei der Auswertung eines zusammengesetzten Ausdrucks werden ja zunächst die Teilausdrücke ausgewertet. Ist einer der dabei ermittelten Werte ein Ausnahmepaket, wird die weitere Auswertung der Teilausdrücke abgebrochen und das Ausnahmepaket als Wert geliefert. Auf diese Weise wird das Ausnahmepaket als Ergebnis der Auswertungen sämtlicher umfassender Ausdrücke weitergereicht, bis ein Behandler gefunden wird.

Bei der Auswertung eines Ausdrucks

```
<Ausdruck1> handle <Muster> => <Ausdruck2>
```

wird zunächst `<Ausdruck1>` ausgewertet. Ist der Wert kein Ausnahmepaket, wird dieser Wert geliefert. Ist der Wert ein Ausnahmepaket `[ <Ausnahme> ]`, erfolgt Pattern Matching zwischen `<Muster>` und dem Ausnahmewert `<Ausnahme>`. Bei Erfolg wird `<Ausdruck2>` ausgewertet und dessen Wert geliefert, bei Misserfolg wird das Ausnahmepaket `[ <Ausnahme> ]` geliefert.

Die Auswertung von `raise`– und `handle`–Ausdrücken kann durch das folgende Beispiel illustriert werden:

```
- exception negative_zahl;

- fun f x = if x = 0
            then true
            else if x > 0
                 then f(x - 1)
                 else raise negative_zahl;
val f = fn : int -> bool

- fun g true  = "wahr"
    | g false = "falsch";
val g = fn : bool -> string

- g(f ~3) handle negative_zahl => "Fehler";
val it = "Fehler" : string
```

Die Auswertung des letzten Ausdrucks kann unter Anwendung des Substitutionsmodells wie folgt erläutert werden:

```
g(f ~3) handle negative_zahl => "Fehler"

g(if ~3 = 0
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g(if false
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g(     if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g(     if false
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g( raise negative_zahl ) handle negative_zahl => "Fehler"

g( [ negative_zahl ] ) handle negative_zahl => "Fehler"

[ negative_zahl ] handle negative_zahl => "Fehler"

"Fehler"
```

Es sei daran erinnert, dass `[ negative_zahl ]` ein sogenanntes Ausnahmepaket bezeichnet.

Ausnahmen sind ihrer Natur nach eng an die Auswertungsreihenfolge gekoppelt. Mit SML-Ausnahmen kann man herausfinden, in welcher Reihenfolge Teilausdrücke ausgewertet werden:

```
- exception a;
exception a
- exception b;
exception b

- ((raise a) + (raise b)) handle b => 2
                               | a => 1;
val it = 1 : int
```

Wäre zuerst der zweite Teilausdruck von `+` ausgewertet worden, so wäre das Gesamtergebnis `2` gewesen.

Wie man eine Ausnahmebehandlung im Stil von `raise` und `handle` so in funktionale Sprachen integrieren kann, dass sie nicht so eng an die Auswertungsreihenfolge gekoppelt ist, ist derzeit Gegenstand der Forschung.

10.6.8 Vordefinierte Ausnahmen von SML

Einige Ausnahmen sind in SML vordefiniert, z.B. die Ausnahmen `Match` und `Bind`, die bei Fehlern während des Pattern Matching erhoben werden.

Die Standardbibliothek von SML beschreibt die vordefinierten Ausnahmen (siehe Abschnitt 2.10 und „The Standard ML Basis Library“ unter:
http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/).

10.7 Erweiterung von SMalL um SMalL-Ausnahmen — `eval4`

Die Datei `eval4.sml` enthält die Version `eval4` des Auswerters. Sie ist eine Erweiterung von `eval3` um SMalL-Ausnahmen und wird in diesem Abschnitt beschrieben. Die Datei `eval4-test.sml` enthält eine auf den Auswerter `eval4` angepasste Fassung der Tests `eval-test.sml`.

Die SMalL-Ausnahmen unterscheiden sich von den SML-Ausnahmen dadurch, dass SMalL nur konstante Ausnahmen (d.h. Ausnahmen ohne Parameter) zulässt. Die Erweiterung des Auswerters `eval4` um die Behandlung von Ausnahmen mit Parametern stellt keine Schwierigkeit dar. Sie wurde unterlassen, um den erweiterten Auswerter einfach zu halten und so das Prinzip der Ausnahmebehandlung besser zur Geltung zu bringen.

10.7.1 Erweiterung der Programmiersprache SMalL — Konkrete Syntax

Die konkrete Syntax von SMalL wird um die beiden folgenden Konstrukte erweitert:

```
raise <Ausnahme>

<Ausdruck1> handle <Ausnahme> => <Ausdruck2>
```

Im Gegensatz zu SML werden SMalL-Ausnahmen, die in `raise`– oder `handle`–Ausdrücken vorkommen, nicht in `exception`–Deklarationen vereinbart. So müssen SMalL-Programme in konkreter Syntax um etwaige `exception`–Deklarationen ergänzt werden, damit sie als SML-Programme vom SML-Laufzeitsystem verarbeitet werden können. Dies ist die einzige Ergänzung, die nötig ist, damit aus einem SMalL-Programm in konkreter Syntax ein SML-Programm wird.

10.7.2 Erweiterung der Programmiersprache SMalL — Abstrakte Syntax

Die abstrakte Syntax von SMalL wird wie folgt erweitert.

Ein Ausdruck in konkreter Syntax der Gestalt

```
raise exn
```

wird dargestellt als:

```
Throw(exn')
```

Ein Ausdruck in konkreter Syntax der Gestalt

```
exp1 handle exn => exp2
```

wird dargestellt als:

```
Catch(exp1', exn', exp2')
```

wobei die Zeichen ' auf die Übersetzung in abstrakte Syntax hinweisen.

10.7.3 Erweiterung der SML-Typdeklarationen des Auswerters

Der Typ `expression` des Auswerters wird erweitert um die zusätzlichen (Wert-) Konstruktoren `Throw` und `Catch`. Sie definieren, wie gerade erklärt, die abstrakte Syntax der neuen SMalL-Ausdrücke. Namen von SMalL-Ausnahmen haben den SML-Typ Zeichenfolge (`string`).

Der Typ `value` des Auswerters wird erweitert um dem zusätzlichen (Wert-)Konstruktor `ExnVal` zur Darstellung der Ausnahmepakete.

Die Typdeklarationen für den Auswerter `eval4` sind also gegenüber den Typdeklarationen für `eval3` wie folgt erweitert:

```
datatype expression    =
                       .
                       .
                       .
                       | Throw  of string
                       | Catch  of expression * string * expression

datatype value         =
                       .
                       .
                       .
                       | ExnVal of string
```

10.7.4 Erweiterung des Auswerters

Zur Behandlung von SMalL-Ausnahmen benötigt `eval4` gegenüber `eval3` drei Erweiterungen:

- einen zusätzlicher Fall zur Auswertung von `Throw`–Ausdrücken;
- Ein zusätzlicher Fall zur Auswertung von `Catch`–Ausdrücken;
- jeder andere Fall des Auswerters, in dessen Rumpf mindestens ein SMalL-Ausdruck ausgewertet wird, dessen Wert ein SMalL–Ausnahmepaket sein könnte, bedarf einer Erweiterung des `case`–Konstrukts, die für das Weiterreichen der Ausnahmepakete gemäß dem Prinzip von Abschnitt 10.6.7 sorgt.

  Betroffen sind die Fälle zur

  - Auswertung eines `UnOp`–Ausdrucks,
  - Auswertung eines `BinOp`–Ausdrucks,
  - Auswertung eines `If`–Ausdrucks,

- Auswertung eines `Dcl`–Ausdrucks,
- Auswertung eines `App`–Ausdrucks,
- Auswertung eines `Seq`–Ausdrucks.

Nicht betroffen sind die Fälle zur

- Auswertung eines `IntExp`–Ausdrucks,
- Auswertung eines `Var`–Ausdrucks,
- Auswertung eines `FnExp`–Ausdrucks.

Fall zur Auswertung eines `Throw`–Ausdrucks

```
eval4(Throw exn, env) = (ExnVal exn, env)
```

Der Wert eines `Throw`–Ausdrucks ist einfach das Ausnahmepaket für denselben Ausnahme–Namen `exn`. Die Umgebung bleibt unverändert.

Fall zur Auswertung eines `Catch`–Ausdrucks

```
eval4(Catch(e,exn,handler), env)
                          = let val (v, _) = eval4(e, env)
                            in
                               case v
                                 of ExnVal exn1
                                      => if exn = exn1
                                         then eval4(handler, env)
                                         else (v, env)
                                  | _ => (v, env)
                            end
```

Zunächst wird der Teilausdruck `e` in der Umgebung `env` ausgewertet, was den Wert `v` ergibt. Das Paar `(v, env)` ist das Ergebnis der gesamten Auswertung, außer wenn `v` ein Ausnahmepaket mit gleichem Ausnahme-Namen wie im `Catch`–Ausdruck ist; in diesem Fall werden alle Folgen der Auswertung von `e` verworfen und statt dessen der Ausnahmebehandler in der Umgebung `env` ausgewertet.

Fall zur Auswertung eines `UnOp`–Ausdrucks

```
eval4(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval4(e, env)
                          in
                             case v
                               of IntVal n
                                    => (IntVal(un_op n), env)
                                | ExnVal _
                                    => (v, env)
                                | _ => raise illegal_expression
                          end
```

Die einzige Änderung dieses Falls gegenüber `eval3` ist der neue Fall `ExnVal _` im `case`–Konstrukt, der dafür sorgt, dass das Ausnahmepaket unverändert weitergereicht wird.

Fall zur Auswertung eines `BinOp`–Ausdrucks

```
eval4(BinOp(op2,e1,e2), env)
                        = let fun eval_op Add  = op +
                                | eval_op Sub  = op -
                                | eval_op Mult = op *
                                | eval_op Div  = op div
                                | eval_op Mod  = op mod
                            val bin_op       = eval_op op2
                            val (v1, _)      = eval4(e1, env)
                            val (v2, _)      = eval4(e2, env)
                          in
                             case (v1,        v2       )
                               of (IntVal n1, IntVal n2)
                                    => (IntVal(bin_op(n1,n2)), env)
                                | (ExnVal _,  _        )
                                    => (v1, env)
                                | (_,         ExnVal _ )
                                    => (v2, env)
                                | _ => raise illegal_expression
                          end
```

Auch hier sind lediglich die neuen Fälle `ExnVal _` für das unveränderte Weiterreichen der Ausnahmepakete hinzugekommen.

Alle anderen betroffenen Fälle von `eval4` werden in gleicher Weise erweitert, damit das Ausnahmepaket unverändert weitergereicht wird. Deshalb soll der Programmtext hier nicht wiederholt werden. Statt dessen sei auf die Datei `eval4.sml` verwiesen.

10.7.5 Beispiele

Alle SMalL-Ausdrücke in der Datei `eval-tests.sml` führen mit `eval4` zu genau den gleichen Ergebnissen wie mit `eval3`. Die Datei `eval4-tests.sml` enthält zusätzliche Beispiele mit den neuen Konstruktoren `Throw` und `Catch`, die von `eval1`, `eval2`, `eval3` nicht ausgewertet werden können.

10.7.6 Weitergabe von Sonderwerten und Ausnahmebehandlung im Vergleich

Der Auswerter `eval4` ist in seinem Prinzip sehr ähnlich zu der in Abschnitt 10.5.4 skizzierten und dann kritisierten Erweiterung von `eval2`. Während die Erweiterung von `eval2` SML-Sonderwerte weiterreichen muss, muss `eval4` SMalL-Ausnahmepakete weiterreichen. Wie ein Vergleich der entsprechenden Programmstellen zeigt, geschieht das Weiterreichen auf fast die gleiche Weise:

Fall `Unop`, Erweiterung von `eval2`:

```
                                case v
                                  of IntVal n
                                       => (IntVal(un_op n), env)
                                   | Error_unbound_variable
                                       => (v, env)
                                   | Error_illegal_expression
                                       => (v, env)
                                   | _ => (Error_illegal_expression, env)
```

Fall `Unop`, `eval4`:

```
case v
  of IntVal n
       => (IntVal(un_op n), env)
   | ExnVal _
       => (v, env)
   | _ => raise illegal_expression
```

Fall `Binop`, Erweiterung von `eval2`:

```
case (v1,        v2       )
  of (IntVal n1, IntVal n2)
       => (IntVal(bin_op(n1,n2)), env)
   | (Error_unbound_variable, _)
       => (v1, env)
   | (Error_illegal_expression, _)
       => (v1, env)
   | (_, Error_unbound_variable)
       => (v2, env)
   | (_, Error_illegal_expression)
       => (v2, env)
   | _ => (Error_illegal_expression, env)
```

Fall `Binop`, `eval4`:

```
case (v1,        v2       )
  of (IntVal n1, IntVal n2)
       => (IntVal(bin_op(n1,n2)), env)
   | (ExnVal _,  _        )
       => (v1, env)
   | (_,         ExnVal _ )
       => (v2, env)
   | _ => raise illegal_expression
```

Die Ähnlichkeit der Vorgehensweise sollte nicht überraschen, weil in beiden Versionen dasselbe Problem gelöst werden soll.

Aber in Abschnitt 10.6.5 wurde festgestellt, dass die Verwendung von SML-Ausnahmen günstiger ist als das Weiterreichen von Sonderwerten durch den Auswerter. Damit stellt sich die Frage, ob die Verwendung von SML-Ausnahmen nicht auch günstiger sein müsste als das Weiterreichen von SMalL-Ausnahmepaketen, also von `ExnVal`–Ausdrücken.

Man könnte argumentieren, dass die obigen Programmstücke für `eval4` kompakter sind als für die Erweiterung von `eval2`. Tatsächlich muss in der Erweiterung von `eval2` für jeden Sonderwert ein eigenes Muster angegeben werden, so dass die Anzahl der Kombinationen von Mustern bei Hinzunahme von weiteren Sonderwerten stark anwächst. Dagegen reicht in `eval4` das Muster `ExnVal _` zur Erfassung jedes beliebigen Ausnahmepakets, egal wie viele weitere Ausnahme–Namen hinzukommen.

Diese Argumentation ist aber nicht fair. Der Datentyp `value` muss für `eval2` nicht um zwei nullstellige (Wert-)Konstruktoren `Error_illegal_expression` und `Error_unbound_variable` erweitert werden. Ein einstelliger (Wert-)Konstruktor `ErrorVal of string` würde den gleichen Zweck erfüllen, wobei anstelle von

`Error_illegal_expression` einfach `ErrorVal "illegal_expression"` geschrieben würde und analog für `unbound_variable`. Dann hätte auch `eval2` den Vorteil, dass das einzige Muster `ErrorVal _` jeden beliebigen Sonderwert abdeckt.

Die eigentliche Antwort auf die Frage ist, dass `eval4` gar nicht den Zweck hat, gegenüber der Version mit SML-Ausnahmen eine Verbesserung zu erzielen. Das gesamte Kapitel hat ja den Zweck zu zeigen, wie typische Programmiersprachenkonstrukte ausgewertet werden können. Der Zweck von `eval4` ist es, zu zeigen, wie die grundlegende Funktionalität zur Ausnahmebehandlung implementiert werden kann. Natürlich kann man sie implementieren, indem man sie einfach auf die entsprechende Funktionalität der Implementierungssprache SML zurückführt. Aber eine explizite Implementierung trägt mehr zum Verständnis der grundlegenden Algorithmen bei, was das eigentliche Anliegen dieses Kapitels ist.

10.8 Rein funktionale Implementierung des Auswerters — `eval5`

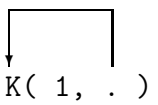
10.8.1 Verweise als Funktionen

Eine Umgebung kann eine Bindung für eine Funktion enthalten. Diese Bindung enthält als Bestandteil diejenige Umgebung, die mit der Bindung selbst beginnt, die somit wiederum die Umgebung enthält, die die Bindung enthält, usw. Offenbar liegt hier ein Zyklus vor.

Betrachten wir ein wesentlich einfacheres Beispiel für einen solchen Zyklus. Gegeben sei ein Typ `T` mit einem zweistelligen Konstruktor `K`, und wir betrachten ein Gebilde `A` der Gestalt `K(1, .)` wobei das zweite Argument von `K` das Gebilde `A` selbst sein soll.

Versucht man, `A` in der gewohnten Form niederzuschreiben, stellt man fest, dass das Niederschreiben nicht terminiert. Deshalb ist `A` auch kein Ausdruck, und deshalb wird `A` hier nur als „Gebilde“ bezeichnet.

Um `A` niederschreiben zu können, benötigt man ein zusätzliches Mittel für die Darstellung, zum Beispiel einen Pfeil wie in den Diagrammen von Abschnitt 10.3.1:



Was durch diesen Pfeil illustriert wird, nennt man abstrakt einen *Verweis*. Der Verweis ist nicht dasselbe wie `A`, sondern liefert `A`, wenn er „verfolgt“ wird. In der graphischen Illustration bedeutet „verfolgen“, dass man die Augen oder den Finger vom Anfang des Pfeils zur Spitze des Pfeils bewegt. Dadurch, dass das zweite Argument von `K` nicht direkt `A` ist, sondern etwas, das erst verfolgt werden muss, um `A` zu erhalten, ist eine Indirektion eingeführt, die den Zyklus unterbricht und eine endliche Darstellung ermöglicht.

Diese Abstraktion kann unter anderem wie folgt durch SML-Referenzen implementiert werden:

```
- datatype T = K of int * (T ref) | K0;
datatype T = K of int * T ref | K0

- fun zyklusK n = let val verweis = ref K0
                      val A = K(n, verweis)
```

```
                  in
                      verweis := A              (* Zyklus erzeugen *)
                      ;
                      A
                  end;
val zyklusK = fn : int -> T

- val A as K(n,verweis) = zyklusK 1;
val A        = K (1,ref (K (#,#)))  : T
val n        = 1                    : int
val verweis  = ref (K (1,%0)) as %0 : T ref

- val A' as K(n',verweis') = !verweis;       (* Verweis verfolgen *)
val A'       = K (1,ref (K (#,#)))  : T
val n'       = 1                    : int
val verweis' = ref (K (1,%1)) as %1 : T ref
```

In dieser Implementierung ist ein „Verweis“ eine SML-Referenz, der Zyklus entsteht mit Hilfe des Zuweisungsoperators „`:=`“, und „verfolgen“ bedeutet Anwenden des Dereferenzierungsoperators „`!`“. Das zweite Argument von `A` ist nicht `A` selbst, aber das Verfolgen dieses zweiten Arguments von `A` liefert das Gleiche wie `A`.

Eine andere Implementierung derselben Abstraktion beruht auf der Beobachtung, dass eine nullstellige Funktion nicht dasselbe ist wie ihr Wert, sondern erst auf `()` angewandt werden muss, um diesen Wert zu liefern. Ausgehend von dieser Beobachtung implementiert man das Beispiel wie folgt:

```
- datatype T  = K of int * (unit -> T) | K0;
datatype T = K of int * (unit -> T) | K0

- fun zyklusK n =let
                  fun verweis() = K(n,verweis) (* Zyklus erzeugen *)
                  val A = verweis()
                 in
                     A
                 end;
val zyklusK = fn : int -> T

- val A as K(n,verweis) = zyklusK 1;
val A        = K(1,fn) : T
val n        = 1       : int
val verweis  = fn      : unit -> T

- val A' as K(n',verweis') = verweis();      (* Verweis verfolgen *)
val A'       = K(1,fn) : T
val n'       = 1       : int
val verweis' = fn      : unit -> T
```

In dieser Implementierung ist ein „Verweis“ eine nullstellige Funktion, der Zyklus entsteht durch Rekursion (`fun` bedeutet ja `val rec`), und „verfolgen“ bedeutet Anwenden der nullstelligen Funktion auf `()`. Das zweite Argument von `A` ist nicht `A` selbst, aber das Verfolgen dieses zweiten Arguments von `A` liefert das Gleiche wie `A`.

Das Bemerkenswerte an dieser Implementierung ist, dass sie nur rein funktionale Hilfsmittel benötigt, also keine Konstrukte mit Nebeneffekten wie die Zuweisung.

Die Sichtweise, dass nullstellige Funktionen Verweisen entsprechen, wird durch die Typen-Syntax gestützt. Der Typ `unit` hat ja als einzigen Wert das leere Tupel, das nur eine syntaktische Darstellung für „nichts“ ist. Man könnte ohne weiteres die Schreibweise `(-> T)` statt `(unit -> T)` erlauben, auch wenn SML das nicht erlaubt. Dann bedeutet `(-> T)` zwar „nullstellige Funktion mit Ergebnis vom Typ `T`“, kann aber ebenso gut als „Verweis auf einen Wert des Typs `T`“ gelesen werden.

10.8.2 Ein rein funktionaler Auswerter für SMalL

Die Datei `eval5.sml` enthält die Version `eval5` des Auswerters. Sie entspricht der Version `eval4`, implementiert aber die Verweise für die Zyklen zwischen Umgebungen und Bindungen durch nullstellige Funktionen.

Insgesamt unterscheiden sich die beiden Versionen nur an vier Stellen:

1. In der `datatype`–Deklaration für den Typ `value` ändert sich

   ```
   | FnVal  of string * expression * (binding list ref)
   ```

   zu

   ```
   | FnVal  of string * expression * (unit -> binding list)
   ```

2. Im Fall für die Auswertung von Funktionsausdrücken ändert sich

   ```
   (FnVal(par, e, ref env), env)
   ```

   zu

   ```
   (FnVal(par, e, fn()=>env), env)
   ```

   Die Änderungen 1. und 2. ergeben sich daraus, dass ein Verweis jetzt keine SML-Referenz sondern eine nullstellige Funktion ist.

3. Im Fall für die Auswertung von Funktionsanwendungen ändert sich

   ```
   val env'   = Bdg(par,v2)
                :: !fn_env
   ```

   zu

   ```
   val env'   = Bdg(par,v2)
                :: fn_env()
   ```

   Die Änderung 3. ergibt sich daraus, wie ein Verweis verfolgt wird.

4. Im Fall für die Auswertung von Deklarationen ändert sich

```
case v
  of FnVal(par, body, fn_env)
       => let
              val fn_env' = Bdg(id,v)
                            :: !fn_env
          in
              fn_env := fn_env'
              ;
              (v, Bdg(id,v)::env)
          end
```

zu

```
case v
  of FnVal(par, body, fn_env)
       => let
            fun fn_env'() = Bdg(id,v'())
                            :: fn_env()
            and v'() = FnVal(par,body,fn_env')
          in
              (v'(), Bdg(id,v'())::env)
          end
```

Hier sieht der Unterschied größer aus, weil die in `eval4` implizite Veränderung von `v` zu einem neuen Wert `v'`, die als Nebeneffekt der Zuweisung stattfindet, in `eval5` explizit gemacht wird. Es wird also ein Name `v'` für den veränderten Wert eingeführt. Siehe hierzu auch die Bemerkungen in Abschnitt 10.4.6.

Der Zyklus entsteht in `eval5` durch die wechselseitige Rekursion zwischen `fn_env'` und `v'`. Da SML wechselseitige Rekursion nur zwischen Funktionen erlaubt, aber nicht zwischen einer Funktion und einer Konstanten, kann `v'` nicht als Konstante definiert werden, sondern ist ebenfalls eine nullstellige Funktion. Wo vorher `v` stand, steht also hinterher `v'()`.

Das sind alle Änderungen. Die Version `eval5` ergibt für alle Beispiele in den Dateien `eval-tests.sml` und `eval4-tests.sml` die gleichen Ergebnisse wie die Version `eval4`. Lediglich die dritten Argumente von `FnVal` werden vom SML-System anders ausgegeben. Wo vorher `ref [...]` stand, steht jetzt `fn` (wie es auch im vorigen Abschnitt erkennbar ist).

Selbstverständlich ist diese Änderung der Repräsentation unabhängig von allen vorherigen Modifikationen des Auswerters. Man kann also mit den obigen vier Änderungen jede der bisherigen Versionen des Auswerters in eine rein funktionale Implementierung umschreiben.

10.9 Meta- und Objektsprache, Bootstrapping

In diesem Kapitel wurde die Programmiersprache SML verwendet, um eine andere Programmiersprache, nämlich SMalL, zu implementieren. Unter Verwendung von Begriffen,

die aus der Linguistik stammen, sagen Informatiker, dass dabei SML als Metasprache und SMalL als Objektsprache dient.

Die *Metasprache* ist die Implementierungs- oder ausführende Sprache. Die *Objektsprache* ist die implementierte oder auszuführende Sprache.

In der Linguistik nennt man Metasprache eine (natürliche) Sprache wie etwa die englische Sprache, in der Aspekte einer anderen Sprache wie z.B. die Grammatik der klassischen arabischen Sprache beschrieben werden. Diese andere Sprache nennt man Objektsprache.

In der Informatik so wie in der Linguistik kommt es häufig vor, dass Meta- und Objektsprache dieselbe Sprache sind. Zur Implementierung einer Programmiersprache hat das sehr wesentliche Vorteile.

Man kann in einer (meist Maschinen-)Sprache den Kern (wie etwa SMalL) einer Programmiersprache (wie SML) implementieren. Dann können in dieser einfachen Kernsprache sämtliche Erweiterungen dieser Kernsprache implementiert werden, z.B. zusammengesetzte Typen, Pattern Matching oder die statische Typprüfung. In der Regel geschehen solche Erweiterungen stufenweise, so dass mehrere Zwischensprachen zwischen der Kern- und Zielsprache entstehen. Dieser Implementierungsansatz heißt *Bootstrapping*.[3]

Das Bootstrapping hat den Vorteil, dass nur die Implementierung der Kernsprache, also einer minimalen Sprache, auf jeden Computertyp und unter jedem Betriebssystem geleistet werden muss. Der Rest der Implementierung der Programmiersprache erfolgt in der Kernsprache dieser Programmiersprache selbst sowie stufenweise in den Zwischensprachen, die daraus implementiert werden. Durch das Bootstrapping wird die Portierung einer Programmiersprache auf verschiedene Computer und auf verschiedene Betriebssystemen wesentlich erleichtert.

[3]Die Technik des Bootstrapping zur Implementierung von Programmiersprachen wird in der Hauptstudiumsvorlesung „Übersetzerbau“ näher erläutert.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 11

Bildung von Abstraktionsbarrieren mit abstrakten Typen und Modulen

Auch wenn der Kern von vielen Programmen klein und übersichtlich ist, ist ein vollständiges Programm oft groß und unübersichtlich. Nicht selten umfasst ein vollständiges Programm hunderte von Prozeduren und tausende von Zeilen. Strukturierung tut also in der Programmierung Not, damit Programme für die menschliche Leserschaft — bekanntlich die wichtigste Leserschaft für Programme — übersichtlich und verständlich bleiben. Programmiersprachen bieten zwei komplementäre Mittel zur Strukturierung von großen Programmen: Die Bildung von Abstraktionsbarrieren zum Verbergen von Definitionen zum einen, die Gruppierung von Teilprogrammen zum anderen. In SML sowie in vielen modernen Programmiersprachen können Abstraktionsbarrieren durch abstrakte Datentypen und Modulen geschaffen werden. Wie die meisten modernen Programmiersprachen bietet SML Module, in SML „Strukturen“ genannt, zur Gruppierung von Teilprogrammen. In diesem Kapitel werden die Begriffe „abstrakter Typ“ und „Modul“ sowie ihre Umsetzung in SML eingeführt.

11.1 Vorzüge des Verbergens

Viele komplexe Programme haben Kerne, die wie der Auswerter `eval4` für SMalL in Kapitel 10 ziemlich klein und übersichtlich sind. In der Regel verlangen aber praktische Anwendungen, dass die Programmkerne erweitert werden.

Damit ein Auswerter für SMalL wie die Prozedur `eval4` aus Kapitel 10 in der Praxis eingesetzt werden kann, müsste er z.B. so ergänzt werden, dass er SMalL-Programme in konkreter statt abstrakter Syntax behandelt, dass er etwaige Syntaxfehler in den auszuwertenden SMalL-Programmen erkennt und meldet und dass er eine statische Typprüfung durchführt. Wäre der kleine Auswerter `eval4` nur in dieser Weise ergänzt, so ergäbe sich ein mehrfach längeres Programm als `eval4`. Dieses längere Programm wäre nicht mehr so schlicht und übersichtlich wie der Kernauswerter `eval4`.

Es wäre natürlich, eine solche Ergänzung vom Auswerter `eval4` in Teilprogramme aufzuteilen, die je eine Aufgabe implementieren würden: ein Teilprogramm wäre z.B. für die Übersetzung von SMalL-Programmen in konkreter Syntax in SMalL-Programme in abstrakter Syntax zuständig, ein anderes Teilprogramm für die Erkennung von Syntaxfehlern, ein weiteres Teilprogramm für die statische Typprüfung von SMalL-Programmen.

Eine Aufteilung in Teilprogramme reicht oft nicht aus, um große oder komplexe Program-

me übersichtlich zu machen. Es ist oft vom Vorteil, dass Definitionen etwa von Prozeduren oder Typen, die nur innerhalb eines Teilprogramms verwendet werden, außerhalb dieses Teilprogramms nicht sichtbar und nicht verwendbar sind. Das erleichtert nachträgliche Änderungen, weil die Änderungen sich dann nur innerhalb des Teilprogramms auswirken und nicht im sehr viel größeren Gesamtprogramm. Dieser Ansatz liegt auch den lokalen Deklarationen einer Prozedur zugrunde (siehe Abschnitt 4.2). Teilprogramme müssen aber oft mehrere Prozeduren und Typen umfassen, so dass lokale Deklarationen von Prozeduren unzureichend sind.

Ein *abstrakter Typ* ergänzt eine Typdefinition — in SML eine `datatype`-Deklaration — um Namen und Prozeduren, die abgerufen werden können, deren Implementierung aber außerhalb der Definition des abstrakten Typs unsichtbar ist. So können zusammen mit einem Typ grundlegende Namen und Operationen zur Verwendung des Typs zur Verfügung gestellt werden, ohne die interne Repräsentation von Werten dieses Typs zugänglich machen zu müssen.

Ein *Modul* ist ein Teilprogramm, das lokale Definitionen beinhalten kann. Mit einem Modul wird festgelegt, welche Definitionen des Moduls außerhalb des Moduls, welche nur innerhalb des Moduls verwendet werden können.

Sowohl bei abstrakten Typen als auch bei Modulen kann erreicht werden, dass nur die zur Verfügung gestellten Operationen verwendet werden. So kann die Datenverarbeitung auf die Verwendung dieser Operationen eingeschränkt werden.

Das Prinzip, nur die für die Verarbeitung notwendigen Namen und Operationen zur Verfügung zu stellen, aber die interne Repräsentation von Werten zu verbergen, ist unter dem Namen *information hiding* bekannt. Es ist eines der wichtigsten Hilfsmittel bei der Softwareentwicklung.

11.2 Abstrakte Typen in SML

11.2.1 Motivationsbeispiel: Das Farbmodell von HTML

Die Markup-Sprache (oder Auszeichnungssprache) HTML sowie die Style-Sheet-Sprache (oder Formatierungssprache) CSS1 repräsentieren Farben nach einem einfachen Modell. In diesem Modell ist eine Farbe durch Anteile an den drei Grundfarben Rot, Grün und Blau — daher die Bezeichnung RGB — definiert, wobei diese Anteile natürliche Zahlen zwischen einschließlich 0 und 255 sind.[1]

Das RGB-Tripel mit den kleinstmöglichen Komponenten (0, 0, 0) entspricht der Farbe Schwarz, das RGB-Tripel mit den größtmöglichen Komponenten (255, 255, 255) entspricht der Farbe Weiß. In der Spezifikation von HTML 4.0 sind insgesamt 16 Farbennamen definiert.[2] Diese 16 Farbennamen gelten übrigens auch für die Markup-Sprache XHTML, die HTML ersetzen soll.

Im Folgenden wird zunächst betrachtet, wie das Farbmodell von HTML durch eine `datatype`–Deklaration in SML definiert werden kann. Nachteile dieser Definition werden angesprochen. Dann wird gezeigt, wie zur Überwindung dieser Nachteile das Farbmodell

[1] Für mehr Information über das Farbmodell von HTML und CSS1 siehe den Abschnitt 6.3 „Color units“ der Spezifikation „Cascading Style Sheets, level 1, W3C Recommendation 17 Dec 1996, revised 11 Jan 1999“, abrufbar unter `http://www.w3.org/TR/1999/REC-CSS1-19990111#color-units`.

[2] siehe „HTML 4.0 Specification, W3C Recommendation, revised on 24-Apr-1998“, abrufbar unter `http://w3c1.inria.fr/TR/1998/REC-html40-19980424/types.html#h-6.5`.

von HTML als abstrakter Typ definiert werden kann.

11.2.2 Ein SML-Typ zur Definition des Farbmodells von HTML

Unter Verwendung einer `datatype`–Deklaration (siehe Abschnitt 8.3) kann in SML ein Typ `color` wie folgt definiert werden:

```
- datatype color = rgb of int * int * int;
datatype color = rgb of int * int * int
```

Nach dieser Typdeklaration können HTML-Farbnamen wie folgt deklariert werden:

```
- val Black   = rgb(  0,   0,   0);
val Black = rgb (0,0,0) : color
- val Gray    = rgb(128, 128, 128);
val Gray = rgb (128,128,128) : color
- val Red     = rgb(255,   0,   0);
val Red = rgb (255,0,0) : color
- val Green   = rgb(  0, 128,   0);
val Green = rgb (0,128,0) : color
- val Lime    = rgb(  0, 255,   0);
val Lime = rgb (0,255,0) : color
- val Blue    = rgb(  0,   0, 255);
val Blue = rgb (0,0,255) : color
- val White   = rgb(255, 255, 255);
val White = rgb (255,255,255) : color
```

Man beachte, dass die grüne Farbe mit maximalem Grünanteil `Lime` heißt. `Green` bezeichnet die grüne Farbe mit halbem Grünanteil.
Eine Funktion zum Mischen zweier Farben kann wie folgt definiert werden:

```
- exception negative_part;
- fun mix(color1, part1, color2, part2) =
         if part1 < 0 orelse part2 < 0
         then raise negative_part
         else let val parts = part1 + part2;
                  val rgb(r1, g1, b1) = color1;
                  val rgb(r2, g2, b2) = color2;
                  val r = (part1*r1 + part2*r2) div parts;
                  val g = (part1*g1 + part2*g2) div parts;
                  val b = (part1*b1 + part2*b2) div parts
              in
                rgb(r, g, b)
              end;
val mix = fn : color * int * color * int -> color
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = rgb (85,84,84) : color
- mix(rgb(1,1,1), 1, rgb(128,128,128), 1);
val it = rgb (64,64,64) : color
- mix(rgb(1,1,1), 1, White, 1);
val it = rgb (128,128,128) : color
```

Werden Farben in SML durch die vorangehende `datatype`–Deklaration definiert, so ist der Typkonstruktor `rgb` und die Tripel-Gestalt einer Farbe sichtbar. So kann z.B. die Funktion `mix` nicht nur auf Farbennamen wie etwa im Ausdruck

```
mix(Red, 1, mix(Lime, 1, Blue, 1), 2)
```

angewendet werden, sondern auch auf `rgb`–Tripel wie im Ausdruck

```
mix(rgb(1,1,1), 1, rgb(128,128,128), 1)
```

Die vorangehende Definition des Farbmodells verhindert nicht, dass ungültige `rgb`–Tripel mit negativen Komponenten oder mit Komponenten größer als 255 vom Programmierer verwendet werden:

```
- val a = rgb(~1,0,999);
val a = rgb (~1,0,999) : color
- mix(rgb(~12,~12,1200),5, rgb(~20,~20,2000),3);
val it = rgb (~15,~15,1500) : color
```

Zudem stellen die Deklarationen von Farbennamen wie `Black`, `Red`, `Blue` und `Lime` und der Funktion `mix` keinen abgeschlossenen Teil im Programm dar. Es fehlt eine syntaktische Einheit, die alle Deklarationen zusammenfasst, die mit RBG-Farben zu tun haben.

11.2.3 Ein abstrakter Typ zur Definition der Farben von HTML

SML bietet sogenannte abstrakte Typen an. Ein abstrakter Typ für die Farben — einschließlich der 16 Farbnamen aus der Spezifikation von HTML 4.0, der Ausnahme `negative_part` und der Funktion `mix` — kann wie folgt deklariert werden:

```
abstype color = rgb of int * int * int
with
     val Black   = rgb(  0,   0,   0);
     val Silver  = rgb(192, 192, 192);
     val Gray    = rgb(128, 128, 128);
     val White   = rgb(255, 255, 255);
     val Maroon  = rgb(128,   0,   0);
     val Red     = rgb(255,   0,   0);
     val Purple  = rgb(128,   0, 128);
     val Fuchsia = rgb(255,   0, 255);
     val Green   = rgb(  0, 128,   0);
     val Lime    = rgb(  0, 255,   0);
     val Olive   = rgb(128, 128,   0);
     val Yellow  = rgb(255, 255,   0);
     val Navy    = rgb(  0,   0, 128);
     val Blue    = rgb(  0,   0, 255);
     val Teal    = rgb(  0, 128, 128);
     val Aqua    = rgb(  0, 255, 255);
     exception negative_part;
     fun mix(color1, part1, color2, part2) =
            if part1 < 0 orelse part2 < 0
```

```
                then raise negative_part
                else let val parts = part1 + part2;
                         val rgb(r1, g1, b1) = color1;
                         val rgb(r2, g2, b2) = color2;
                         val r = (part1*r1 + part2*r2) div parts;
                         val g = (part1*g1 + part2*g2) div parts;
                         val b = (part1*b1 + part2*b2) div parts
                     in
                         rgb(r, g, b)
                     end;
        fun display(rgb(r,g,b)) = print(Int.toString r  ^  " "  ^
                                        Int.toString g  ^  " "  ^
                                        Int.toString b  ^  "\n");
end;
```

Diese `abstype`–Deklaration bildet eine syntaktische Einheit, in der die Deklarationen der Farbennamen, die Deklaration der Ausnahme `negative_part` und die Deklarationen der Funktionen `mix` zur Farbmischung und `display` zum Drucken eines Farbwerts eingeschlossen sind. Die Deklaration des Typs enthält also alle diese Deklarationen.

So wird ein abgeschlossenes Teilprogramm gebildet, das alle Deklarationen umfasst, die zu dem neuen Typ `color` gehören und zu dessen Definition beitragen.

Die Mitteilung des SML-Systems als Reaktion auf die obige Deklaration ist

```
type color
val Black = - : color
val Silver = - : color
val Gray = - : color
val White = - : color
val Maroon = - : color
val Red = - : color
val Purple = - : color
val Fuchsia = - : color
val Green = - : color
val Lime = - : color
val Olive = - : color
val Yellow = - : color
val Navy = - : color
val Blue = - : color
val Teal = - : color
val Aqua = - : color
exception negative_part
val mix = fn : color * int * color * int -> color
val display = fn : color -> unit
```

Daran fällt auf, dass die innere Gestalt einer Farbe, d.h. die (Wert-) Konstruktoren des neuen Typs `color`, in dieser Mitteilung nirgends erwähnt wird.

Tatsächlich ist diese innere Gestalt der Werte des Typs außerhalb der `abstype`–Deklaration nicht mehr verwendbar, wie die folgenden Beispiele zeigen:

```
- val mein_lieblingsblau = rgb(0, 85, 255);
Error: unbound variable or constructor: rgb
- mix(rgb(1, 1, 1), 1, rgb(128, 128, 128), 1);
Error: unbound variable or constructor: rgb
Error: unbound variable or constructor: rgb
```

Dagegen sind die Namen der Farben und Funktionen der `abstype`–Deklaration außerhalb der `abstype`–Deklaration verwendbar — sonst wäre ihre Deklaration auch ziemlich nutzlos. Nur die innere Gestalt ihrer Werte wird verborgen:

```
- Red;
val it = - : color
- mix(Red, 1, Red, 1);
val it = - : color
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = - : color
- display(mix(Red, 1, mix(Lime, 1, Blue, 1), 2));
85 84 84
val it = () : unit
```

Ein Programm, das den Typ `color` verwendet, kann also auf die dort definierten Farben über die zur Verfügung gestellten Namen wie `Red` oder `Lime` zugreifen und kann neue Farben mit der zur Verfügung gestellten Funktion `mix` erzeugen. Es kann aber keine Farben auf andere Weise bilden, insbesondere kann es nicht den Konstruktor `rgb` benutzen, um beliebige Zahlentripel zur Darstellung von Farben zu verwenden.

Daraus folgt, dass ungültige `rgb`–Tripel mit negativen Komponenten oder mit Komponenten größer als 255 weder vom Programmierer verwendet werden können, noch — dank der sorgfältigen Definition der Funktion `mix` — erzeugt werden können.

11.2.4 Implementierungstyp und Schnittstelle eines abstrakten Typs

Die Definition eines abstrakten Typs besteht aus einer Typdefinition und Wertdefinitionen, die auch Funktionsdefinitionen sein können. Es wird daran erinnert, dass in SML sowie in vielen funktionalen Programmiersprachen Funktionsdefinitionen Wertdefinitionen sind (siehe die Abschnitte 2.4.3 und 7.1).

Der Typ, der in einer `abstype`–Deklaration definiert wird, wird *Implementierungstyp* des abstrakten Typs genannt. Die Deklarationen, einschließlich der Deklarationen von Funktionen, die in einer `abstype`–Deklaration vorkommen, bilden die sogenannte *Schnittstelle* des abstrakten Typs.

Ein Programm, das einen abstrakten Typ verwendet, d.h., sich darauf bezieht, wird manchmal „Klient“ dieses abstrakten Typs genannt.

11.2.5 Vorteile des Verbergens mit abstrakten Typen

Unabhängigkeit der Klienten von der Implementierung eines Typs

Wird ein abstrakter Typ in einem Programm verwendet, so verlangt eine Veränderung seiner Definition keine Veränderung seiner Klienten, so lange die Schnittstelle des abstrakten

Typs unverändert bleibt.

Beispielsweise könnte es sein, dass das Farbmodell von HTML verfeinert werden soll, so dass für die drei Anteile der Grundfarben nicht nur 256 Abstufungen erlaubt sind, sondern 1024. In diesem verfeinerten Modell können die Komponenten der Tripel also Werte von 0 bis 1023 einschließlich annehmen. Jedes Tripel des ursprünglichen Modells muss komponentenweise mit 4 multipliziert werden, um das Tripel für dieselbe Farbe im verfeinerten Modell zu erhalten.

Wird das HTML-Farbmodell mit dem abstrakten Typ `color` implementiert, so reicht diese Vervierfachung der Zahlen in den Deklarationen der (benannten) Farben `Black`, `Silver`, usw., um den Klienten ohne weitere Änderungen auf das verfeinerte Farbmodell umzustellen. Könnte der Klient dagegen direkt die interne Tripel-Gestalt der Farben verwenden, müsste für eine solche Umstellung der gesamte (möglicherweise sehr große) Klient daraufhin durchsucht werden, wo ein Zahlentripel erzeugt wird, das zur Darstellung einer Farbe dient und deshalb vervierfacht werden muss.

Die Verwendung von abstrakten Typen erleichtert wesentlich die stufenweise Entwicklung und die Wartung von komplexen Programmen, weil sie lokale Veränderungen ermöglichen.

Einkapselung zur sicheren Wertverarbeitung

Zudem können Werte eines abstrakten Datentyps nur unter Verwendung der zu diesem Zweck in der Definition des abstrakten Typs zur Verfügung gestellten Namen und Funktionen erzeugt, verändert und im Allgemeinen manipuliert werden. Diese Einkapselung verhindert eine Verarbeitung der Werte eines abstrakten Typs, die bei der Implementierung des Typs nicht beabsichtigt wurde.

In der Praxis entstehen häufig Fehler, wenn Werte eines Typs in einer Weise verarbeitet werden, die bei der Implementierung des Typs nicht beabsichtigt wurde.

Die Einkapselungstechnik stellt einen sehr wesentlichen Beitrag zur Erstellung sicherer Software dar.

11.3 Beispiel: Abstrakte Typen zur Implementierung der Datenstruktur „Menge“

Im Folgenden betrachten wir endliche Mengen von ganzen Zahlen, die extensional definiert sind (siehe Abschnitt 8.6). Der Übersichtlichkeit halber werden lediglich die folgenden Namen und Mengenoperationen definiert:

- der Name `empty_set` zur Definition der leeren Menge;
- das Prädikat `is_empty` zur Überprüfung, ob eine Menge leer ist;
- das Prädikat `is_element` zur Überprüfung der Elementbeziehung;
- die Funktion `insert`, um ein Element in eine Menge einzufügen;
- die Funktion `remove`, um ein Element aus einer Menge zu entfernen;
- die Funktion `union`, um zwei Mengen zu vereinigen;
- die Funktion `display`, um die Elemente einer Menge zu drucken.

Zwei unterschiedliche abstrakte Typen mit derselben Schnittstelle werden implementiert. Der erste abstrakte Typ stellt Mengen als unsortierte Listen ohne Wiederholungen von Elementen dar. Der zweite abstrakte Typ verfeinert diese erste Darstellung mit der Berücksichtigung der kleinsten und größten Mengenelemente zur Beschleunigung der Funktionen `is_element`, `insert` und `remove`.

Ein durchaus realistisches Szenario ist, dass im Rahmen eines größeren Programmierprojekts zunächst die naheliegende, einfache erste Implementierung realisiert wird. Dann stellt sich bei der Benutzung des Programms heraus, dass es für die Anwendung zu langsam ist. Eine Analyse ergibt, dass die Elemente meistens in aufsteigender oder absteigender Reihenfolge sortiert in die Mengen eingefügt werden. Dadurch entsteht die Idee für die zweite Implementierung. Wenn die Mengen als abstrakte Typen realisiert sind, ist die Umstellung auf die zweite Implementierung möglich, ohne den Rest des Programms zu verändern.

11.3.1 Erster abstrakter Typ zur Implementierung der Menge als Liste

Die Mengen von ganzen Zahlen werden als unsortierte Listen ohne Wiederholung von Elementen dargestellt.

```
- abstype set = Set of int list
     with
          val empty_set                = Set nil;
          fun is_empty x               = (x = Set nil);
          fun is_element(_, Set nil)   = false
            | is_element(e, Set(h::t)) = (e = h)
                                         orelse
                                         is_element(e, Set t);
          fun insert(e, Set nil)       = Set(e::nil)
            | insert(e, Set(h::t))     =
                              if e = h
                              then Set(h::t)
                              else let val Set L = insert(e, Set t)
                                   in
                                      Set(h::L)
                                   end;
          fun remove(e, Set nil)       = Set nil
            | remove(e, Set(h::t))     =
                              if e = h
                              then Set t
                              else let val Set L = remove(e, Set t)
                                   in
                                      Set(h::L)
                                   end;
          fun union(Set nil,   M)      = M
           | union(Set(h::t), Set L)   =
                         if is_element(h, Set L)
                         then union(Set t, Set L)
                         else let val Set L1 = union(Set t, Set L)
```

```
                              in
                                 Set(h::L1)
                              end;
            fun display(Set nil)         = print "\n"
              | display(Set(h::t))       =
                                    (
                                     print(Int.toString(h) ^ " ")
                                     ;
                                     display(Set t)
                                    );
      end;
 type set
 val empty_set = - : set
 val is_empty = fn : set -> bool
 val is_element = fn : int * set -> bool
 val insert = fn : int * set -> set
 val remove = fn : int * set -> set
 val union = fn : set * set -> set
 val display = fn : set -> unit

- is_element(1,  insert(1, empty_set));
 val it = true : bool

 - is_empty(remove(1, insert(1, empty_set)));
 val it = true : bool

 - insert(3, insert(2, insert(1, empty_set)));
 val it = - : set

 - display(insert(3, insert(2, insert(1, empty_set))));
 1 2 3
 val it = () : unit

 - display(remove(2, insert(3, insert(2, insert(1, empty_set)))));
 1 3
 val it = () : unit

 - is_element(1,  insert(1, empty_set));
 val it = true : bool

 - val set1 = insert(3, insert(2, insert(1, empty_set)));
 val set1 = - : set

 - val set2 = insert(5, insert(4, insert(3, empty_set)));
 val set2 = - : set

 - display(union(set1, set2));
 1 2 3 4 5
 val it = () : unit
```

11.3.2 Zweiter abstrakter Typ zur Implementierung der Menge als Liste

Eine Menge von ganzen Zahlen wird nun als Tripel `(k, g, L)` dargestellt, wobei

- `k` das kleinste Element der Menge ist,
- `g` das größte Element der Menge ist,
- `L` die unsortierte Liste (ohne Wiederholung) der Mengenelemente ist.

```
- abstype set = Set of int * int * int list
              | EmptySet
    with
         val empty_set                       = EmptySet;
         fun is_empty x                      = (x = EmptySet);
         fun is_element(_, EmptySet)         = false
           | is_element(e, Set(k, g, h::t)) =
                     let fun member(_, nil)  = false
                           | member(m, h::t) = if m = h
                                               then true
                                               else member(m, t)
                     in
                        if e < k orelse e > g
                        then false
                        else member(e, h::t)
                     end;
         fun insert(e, EmptySet)             = Set(e, e, e::nil)
           | insert(e, Set(k, g, L))         =
               if e < k
               then Set(e, g, e::L)
               else if e > g
                    then Set(k, e, e::L)
                    else let fun add(e, nil)  = e::nil
                               | add(e, h::t) = if e = h
                                                then h::t
                                                else h::add(e, t)
                         in
                            Set(k, g, add(e, L))
                         end;
         fun remove(e, EmptySet)             = EmptySet
           | remove(e, Set(k, g, L))         =
               if e < k orelse e > g
               then Set(k, g, L)
               else let fun remove(e, nil)  = nil
                          | remove(e, h::t) = if e = h
                                              then t
                                              else h::remove(e,t);
                        fun min(h::nil)     = h
                          | min(h::t)       = Int.min(h, min t);
```

```
                            fun max(h::nil)     = h
                              | max(h::t)       = Int.max(h, max t);
                            val L'              = remove(e, L)
                        in
                           if L' = nil
                           then EmptySet
                           else Set(min L', max L', L')
                        end;
            fun union(s, EmptySet)       = s
              | union(EmptySet, s)       = s
              | union(Set(k1, g1, L1), Set(k2, g2, L2)) =
                   let fun member(_, nil)  = false
                         | member(m, h::t) = if m = h
                                             then true
                                             else member(m, t);
                       fun merge(nil, L)  = L
                         | merge(h::t, L) =
                              if member(h, L)
                              then merge(t, L)
                              else h::merge(t, L)
                   in
                     Set(Int.min(k1, k2),Int.max(g1, g2),merge(L1,L2))
                   end;
            fun display(EmptySet)        = print "\n"
              | display(Set(_, _, h::t)) =
                            let fun display_list(nil)  = print "\n"
                                  | display_list(h::t) =
                                       (
                                        print(Int.toString(h) ^ " ");
                                        display_list(t)
                                       );
                            in
                               display_list(h::t)
                            end
     end;
type set
val is_empty = fn : set -> bool
val is_element = fn : int * set -> bool
val insert = fn : int * set -> set
val remove = fn : int * set -> set
val union = fn : set * set -> set
val display = fn : set -> unit

- is_element(1,  insert(1, empty_set));
val it = true : bool

- is_empty(remove(1, insert(1, empty_set)));
val it = true : bool
```

```
- insert(3, insert(2, insert(1, empty_set)));
val it = - : set

- display(insert(3, insert(2, insert(1, empty_set))));
3 2 1
val it = () : unit

- display(remove(2, insert(3, insert(2, insert(1, empty_set)))));
3 1
val it = () : unit

- is_element(1,  insert(1, empty_set));
val it = true : bool

- val set1 = insert(3, insert(2, insert(1, empty_set)));
val set1 = - : set

- val set2 = insert(5, insert(4, insert(3, empty_set)));
val set2 = - : set

- display(union(set1, set2));
2 1 5 4 3
val it = () : unit
```

Dieselben Ausdrücke wie im Abschnitt 11.3.1 werden in ähnlicher Weise ausgewertet. Der einzige Unterschied liegt in der Reihenfolge der Mengenelemente in der Liste, die die Menge darstellt. Dieser Unterschied kommt daher, dass die Operationen `insert` und `remove` zum Einzufügen bzw. zum Entfernen von Elementen in unterschiedlicher Weise implementiert sind.

11.4 Module in SML

Module ermöglichen, größere Programme hierarchisch zu strukturieren. Die Modulbegriffe von SML heißen:

- Struktur,
- Signatur,
- Funktor.

SML-Strukturen sind den in den vorangehenden Kapiteln eingeführten SML-Programmen ähnlich. In einer SML-*Struktur* können Werte, eventuell Funktionen, die in SML wie in vielen funktionalen Programmiersprachen Werte sind (siehe Abschnitt 2.4.3), Typen und Ausnahmen definiert werden.

Eine SML-*Signatur* teilt mit, was eine Struktur implementiert, ohne die Implementierung preiszugeben. Eine Signatur ist also dem Typ einer Funktion ähnlich, der in abgekürzter Form (oder abstrakt) wiedergibt, was die Funktion leistet, ohne die Implementierung der Funktion preiszugeben.

Ein SML-*Funktor* ist ein parametrisiertes Modul. Ein SML-Funktor kann auch als eine Art Funktion angesehen werden, womit Strukturen auf Strukturen abbildet werden.
Die folgende Ähnlichkeit gilt also:

- Strukturen sind Werten ähnlich.
- Signaturen sind Typen ähnlich.
- Funktoren sind Funktionen ähnlich.

Im Folgenden werden SML-Strukturen, SML-Signaturen und SML-Funktoren näher erläutert.

11.4.1 SML-Strukturen

Deklarationen können in einer SML-Struktur zusammengefasst werden. Die folgende Struktur fasst z.B. alle Deklarationen zusammen, die zur Definition eines Typs „komplexe Zahlen“ benötigt werden:

```
structure Complex =
  struct
    type t                                  = real * real;
    val zero                                = (0.0, 0.0) : t;
    fun sum       ((x1,y1):t, (x2,y2):t) = (x1 + x2, y1 + y2) : t;
    fun difference((x1,y1):t, (x2,y2):t) = (x1 - x2, y1 - y2) : t;
    fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2 - y1 * y2,
                                             x1 * y2 + x2 * y1) : t;
    fun reciprocal((x,y) : t)               = let val r = x * x + y * y
                                              in
                                                  (x/r, ~y/r) : t
                                              end;
    fun quotient  (z1 : t,    z2 : t)    = product(z1, reciprocal z2)
  end;
```

Die Namen, die in einer Struktur deklariert sind, sind außerhalb dieser Struktur nicht sichtbar. Während z.B. die Funktion `reciprocal` in der Struktur verwendet werden kann, wie etwa in der Definition der Funktion `quotient`, ist sie außerhalb der Struktur unbekannt:

```
- reciprocal(1.0, 0.0);
Error: unbound variable or constructor: reciprocal
```

Ein Name `N`, der in einer Struktur namens `S` deklariert ist, kann außerhalb dieser Struktur `S` als `S.N` verwendet werden. So ist der Zugriff auf die Funktion `reciprocal` außerhalb der Struktur `Complex` durch den Namen `Complex.reciprocal` möglich:

```
- Complex.reciprocal(1.0, 0.0);
val it = (1.0,0.0) : Complex.t
```

Dient eine Struktur namens `S` wie im vorangehenden Beispiel zur Definition eines Typs, so wird häufig dieser Typ in der Struktur `t` genannt, außerhalb dieser Struktur `S.t`:

```
- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- Complex.product(i, i);
val it = (~1.0,0.0) : Complex.t
```

Zur Definition des Typs „komplexe Zahlen“ könnte ein abstrakter Typ der vorangehenden Struktur vorgezogen werden. Ein abstrakter Typ würde die Implementierung der Werte verbergen und so unbeabsichtigte Verwendungen verhindern.

Die Mitteilung, die SML liefert, wenn die Struktur `Complex` deklariert wird, ist die Signatur dieser Struktur, die aus den Mitteilungen besteht, die gegeben würden, wenn die Deklarationen der Struktur einzeln definiert würden:

```
structure Complex :
  sig
    type t = real * real
    val difference : t * t -> t
    val product : t * t -> t
    val quotient : t * t -> t
    val reciprocal : t -> t
    val sum : t * t -> t
    val zero : t
  end
```

11.4.2 SML-Signaturen

Eine SML-Signatur ist eine abstrakte Beschreibung der Deklarationen einer SML-Struktur oder mehrerer SML-Strukturen. Eine Signatur wird nicht nur vom SML-System aus einer Strukturdeklaration ermittelt, sondern kann auch vom Programmierer selbst deklariert werden.

Die folgende Signatur beschreibt z.B. jede Struktur, die einen Typ `t` sowie die grundlegenden arithmetischen Operationen über `t` implementiert:

```
- signature ARITHMETIC =
   sig
    type t
    val zero       : t
    val sum        : t * t -> t
    val difference : t * t -> t
    val product    : t * t -> t
    val reciprocal : t -> t
    val quotient   : t * t -> t
  end;
signature ARITHMETIC =
sig
```

```
    type t
    val zero : t
    val sum : t * t -> t
    val difference : t * t -> t
    val product : t * t -> t
    val reciprocal : t -> t
    val quotient : t * t -> t
end
```

Die Ausdrücke einer Signatur, die zwischen den reservierten Wörtern `sig` und `end` vorkommen, heißen *(Signatur-) Spezifikationen.*

Die Signatur `ARITHMETIC` kann wie folgt in sogenannten *Signatur-Constraints* verwendet werden, wenn eine Struktur definiert wird, die alle in der Signatur spezifizierten Komponenten deklariert:

```
- structure Rational : ARITHMETIC =
    struct
      type t                                  = int * int;
      val zero                                = (0, 1) : t;
      fun sum       ((x1,y1):t, (x2,y2):t) = (x1*y2 + x2*y1, y1*y2) :t;
      fun difference((x1,y1):t, (x2,y2):t) = (x1*y2 - x2*y1, y1*y2) :t;
      fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2, y1 * y2) : t;
      fun reciprocal((x,y) : t)             = (y,x) : t
      fun quotient  (z1 : t,    z2 : t)     = product(z1, reciprocal z2)
    end;
structure Rational : ARITHMETIC

- structure Complex : ARITHMETIC =
    struct
      type t                                  = real * real;
      val zero                                = (0.0, 0.0) : t;
      fun sum       ((x1,y1):t, (x2,y2):t) = (x1 + x2, y1 + y2) : t;
      fun difference((x1,y1):t, (x2,y2):t) = (x1 - x2, y1 - y2) : t;
      fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2 - y1 * y2,
                                               x1 * y2 + x2 * y1) : t;
      fun reciprocal((x,y) : t)             = let val r = x * x + y * y
                                              in
                                                  (x/r, ~y/r) : t
                                              end;
      fun quotient  (z1 : t,    z2 : t)     = product(z1, reciprocal z2)
    end;
structure Complex : ARITHMETIC
```

Ähnlich wie Typ-Constraints die statische Typprüfung ermöglichen, ermöglichen Signaturen das statische Zusammenfügen von Deklarationen aus unterschiedlichen Teilprogrammen, d.h. das Zusammenfügen von Deklarationen aus Teilprogrammen, ohne deren Implementierungen zu berücksichtigen.

Man beachte, dass die Typabkürzungen `Rational.t` und `Complex.t` unterschiedliche Typen, nämlich `int * int` und `real * real`, bezeichnen, obwohl beide Strukturen `Rational` und `Complex` dieselbe Signatur `ARITHMETIC` haben.

Es ist üblich, dass die Namen von Strukturen mit einem Großbuchstaben beginnen und die Namen von Signaturen ganz aus Großbuchstaben bestehen, aber natürlich erzwingt SML diese Konvention nicht.

11.4.3 Spezifikation versus Deklaration in SML-Signaturen

Eine Signatur besteht aus Spezifkationen, nicht Deklarationen. Zum Beispiel ist

```
type t
```

eine Spezifikation. Sie teilt mit, dass in der Signatur der Name `t` einen Typ bezeichnet. Diese Spezifikation teilt aber nicht mit, wie der Typ `t` definiert ist. Insbesondere teilt sie nicht mit, welche (Wert-)Konstruktoren der Typ `t` hat.

Um den Unterschied zwischen Spezifikationen und Deklarationen zu unterstreichen, darf das reservierte Wort `fun` in einer Signatur nicht vorkommen, sondern nur das reservierte Wort `val` wie etwa in:

```
val sum  : t * t -> t
```

11.4.4 `eqtype`–Spezifikationen in SML-Signaturen

In einer SML-Signatur kann die Spezifikation eines Typs `t` mit dem reservierten Wort `eqtype` statt `type` eingeführt werden, wenn die Gleichheit über `t` definiert sein muss.

In diesem Zusammenhang sei daran erinnert, dass keine Gleichheit über Funktionstypen definiert ist (siehe z.B. Abschnitt 10.1.2).

11.4.5 `datatype`–Spezifikationen in SML-Signaturen

In einer SML-Signatur kann die Spezifikation eines Typs `t` auch in Form einer `datatype`–Deklaration erfolgen. In diesem Fall ist nicht nur spezifiziert, dass `t` ein Typ ist, sondern auch, welche (Wert-)Konstruktoren der Typ `t` hat. Ein Beispiel dafür kommt in der Signatur `LIST` vor (ganz am Ende dieses Kapitels).

11.4.6 Angleich einer Struktur an eine Signatur — Strukturansichten

Eine Struktur `Struk` kann an eine Signatur `Sig` angeglichen werden, wenn alle Komponenten, die in der Signatur `Sig` spezifiziert werden, in der Struktur `Struk` deklariert sind.

Es ist aber möglich, eine Struktur an eine Signatur anzugleichen, die weniger Komponenten spezifiziert, als die Struktur deklariert. So können eingeschränkte Sichten (*views*) auf eine Struktur definiert werden.

Die folgende Signatur spezifiziert z.B. nur einen Teil der Namen, die in der Struktur `Complex` deklariert sind:

```
- signature RESTRICTED_ARITHMETIC =
   sig
    type t
    val zero       : t
    val sum        : t * t -> t
    val difference : t * t -> t
  end;
signature RESTRICTED_ARITHMETIC =
  sig
    type t
    val zero : t
    val sum : t * t -> t
    val difference : t * t -> t
  end
```

Unter Verwendung der Signatur `RESTRICTED_ARITHMETIC` erzeugt die folgende Deklaration eine Einschränkung der Struktur `Complex`, die nur die Namen zur Verfügung stellt, die in der Signatur `RESTRICTED_ARITHMETIC` vorkommen. Die Definition dieser Namen ist diejenige aus der (uneingeschränkten) Struktur `Complex`:

```
- structure RestrictedComplex :  RESTRICTED_ARITHMETIC = Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC

- val i = (0.0, 1.0) : RestrictedComplex.t;
val i = (0.0,1.0) : Complex.t

- RestrictedComplex.sum(RestrictedComplex.zero, i);
val it = (0.0,1.0) : Complex.t

- RestrictedComplex.product(RestrictedComplex.zero, i);
Error: unbound variable or constructor: product in path
       RestrictedComplex.product

- Complex.product(RestrictedComplex.zero, i);
val it = (0.0,0.0) : Complex.t
```

In der Deklaration der Struktur RestrictedComplex ist

```
: RESTRICTED_ARITHMETIC
```

ein sogenanntes *Signatur-Constraint*. Mit Signatur-Constraints kann also eine Form des Verbergens (*information hiding*) erreicht werden.

11.4.7 Parametrisierte Module in SML: SML-Funktoren

Beide Strukturen, `Rational` und `Complex`, können um eine Funktion `square` erweitert werden, die eine rationale bzw. komplexe Zahl auf ihr Quadrat abbildet. Die Deklaration einer solchen Funktion `square` wäre in beiden Strukturen identisch:

```
fun square z = product(z, z)
```

Wären die beide Strukturen `Rational` und `Complex` um die obige Deklaration ergänzt, so wären die Funktionen `Rational.square` und `Complex.square` doch unterschiedlich definiert, weil jede Deklaration sich auf die Definition der Funktion `product` aus der eigenen Struktur bezieht und `Rational.product` und `Complex.product` unterschiedlich definiert sind.

Es ist kein Zufall, dass `Rational.square` und `Complex.square` identisch deklariert werden können, auch wenn sie unterschiedlich definiert sind. Jeder Zahlentyp, der über ein Produkt verfügt, ermöglicht dieselbe Definition des Quadrats im Bezug auf dieses Produkt.

Die Erweiterung beider Strukturen `Rational` und `Complex` um eine Funktion `square` wäre nachteilhaft. Sie würde die Identität der Deklarationen beider Funktionen `square` verbergen und so etwaige Veränderungen dieser Funktionen erschweren. Genauso wie es sich anbietet, eine Teilberechnung, die in einem Algorithmus mehrfach vorkommt (siehe die Abschnitte 1.1.6 und 4.1), als Prozedur zu definieren, so bietet es sich an, Deklarationen, die in verschiedenen Strukturen identisch sind, nur einmal für die verschiedenen Strukturen anzugeben. Dazu dienen SML-Funktoren.

Ein SML-Funktor ist eine SML-Struktur, die andere SML-Strukturen als Parameter erhält, also eine sogenannte *parametrisierte Struktur*.

So kann z.B. die Funktion `square` für alle Strukturen mit der Signatur `ARITHMETIC` wie folgt definiert werden:

```
- functor Extended(S : ARITHMETIC) =
    struct
      fun square z = S.product(z, z)
    end;
functor Extended : <sig>
```

Der Funktor `Extended` stellt erst dann eine benutzbare Struktur dar, wenn der Name `S` an eine Struktur mit Signatur `ARITHMETIC`, also eine Struktur wie etwa `Rational` oder wie `Complex`, gebunden wird. Dies geschieht etwa wie folgt:

```
- structure ExtComplex = Extended(Complex);
structure ExtComplex : sig val square : S.t -> S.t end

- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- ExtComplex.square(i);
val it = (~1.0,0.0) : Complex.t

- structure ExtRational = Extended(Rational);
structure ExtRational : sig val square : S.t -> S.t end

- val minusone = (~1, 1) : Rational.t;
val minusone = (~1,1) : Rational.t

- ExtRational.square(minusone);
val it = (1,1) : Rational.t
```

Die Quadratfunktionen beider Strukturen `ExtComplex` und `ExtRational` sind, wie gewünscht, unterschiedliche Funktionen, wenn auch ihre Deklarationen im Funktor `Extended` syntaktisch identisch sind:

```
- ExtComplex.square(0.0, 1.0);
val it = (~1.0,0.0) : Complex.t

- ExtRational.square(0, 1);
val it = (0,1) : Rational.t
```

Während der (statischen) Typprüfung eines Funktors wird ausschließlich die Information verwendet, die die Signatur liefert. So entsteht die Abstraktion, die ermöglicht, dass zur Laufzeit ein Funktor auf unterschiedliche Strukturen angewendet wird. Sobald ein Funktor keine Typfehler enthält, kann er auf jede Struktur angewendet werden, die mit der Signatur des Parameters des Funktors angeglichen werden kann.

Es wäre übrigens naheliegend, die Signatur `ARITHMETIC` ohne die Funktion `quotient` zu definieren und diese Funktion in generischer Weise durch einen Funktor auf `product` und `reciprocal` zurückzuführen.

11.4.8 Generative und nichtgenerative Strukturdeklarationen

Zwei Formen von Strukturdeklarationen sind also möglich.

Zum einen kann eine Struktur wie folgt definiert werden:

```
- structure RestrictedComplex  =
    struct
      type t                              = real * real;
      val zero                            = (0.0, 0.0) : t;
      fun sum       ((x1,y1), (x2,y2)) = (x1 + x2, y1 + y2) : t;
      fun difference((x1,y1), (x2,y2)) = (x1 - x2, y1 - y2) : t;
    end;
```

Man spricht dann von einer „generativen Strukturdeklaration“. Diese Bezeichnung unterstreicht, dass die Strukturdeklaration ihre Komponenten explizit deklariert.

Zum anderen kann eine Struktur unter Verwendung von bereits deklarierten Strukturen und/oder Signaturen etwa wie folgt deklariert werden:

```
- structure RestrictedComplex :  RESTRICTED_ARITHMETIC = Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC
```

In einer solchen Strukturdeklaration werden die Komponenten der Struktur nicht explizit, sondern implizit in Bezug auf bereits vorgenommene Deklarationen definiert. Man spricht in einem solchen Fall von einer „nichtgenerativen Strukturdeklaration“.

11.4.9 Weiteres über Module in SML

Geschachtelte Strukturen

Geschachtelte Strukturen sind möglich wie etwa in:

```
- structure Nb =
     struct
        structure ComplexNb  = Complex;
        structure RationalNb = Rational
     end;
structure Nb :
  sig
    structure ComplexNb : <sig>
    structure RationalNb : <sig>
  end

- val i = (0.0, 1.0) : Nb.ComplexNb.t;
val i = (0.0,1.0) : Complex.t

- Nb.ComplexNb.product(i, Nb.ComplexNb.zero);
val it = (0.0,0.0) : Complex.t

- val minusone = (~1, 1) : Nb.RationalNb.t;
val minusone = (~1,1) : Rational.t

- Nb.RationalNb.sum(minusone, Nb.RationalNb.zero);
val it = (~1,1) : Rational.t
```

Lange Namen

Namen wie `Complex.t`, `Complex.product` und `Nb.RationalNb.sum` werden „lange Namen“ (oder „lange Bezeichner“) genannt.
Da geschachtelte Strukturen möglich sind, können lange Namen aus mehr als zwei (nichtlangen) Namen bestehen, wie etwa `Nb.RationalNb.sum`.

Teilen (*sharing*) von Deklarationen in Strukturen

In einer Struktur– oder Signaturdeklaration kann verlangt werden, dass Namen aus unterschiedichen Strukturen oder Signaturen „geteilt“ werden, d.h. als identisch gelten (siehe dazu die Literatur über SML).

11.5 Hinweis auf die Standardbibliothek von SML

Die Standardbibliothek von SML, die unter der URI

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

verfügbar ist, besteht aus Modulen. In ihr kommen Signaturdeklarationen und Spezifikationen vor. Ein Beispiel stellen die Spezifikationen der Struktur `List` dar (siehe `http://www.dina.kvl.dk/%7Esestoft/sml/list.html`).
Wenn man den Namen einer Struktur wie `List` kennt, kann man im SML-System die zugehörige Signatur herausfinden, indem man einen neuen (irrelevanten) Strukturnamen deklariert:

```
- structure Struct = List;
structure Struct : LIST
```

Die Mitteilung des SML-Systems enthält also die Information, dass die Struktur `List` die Signatur `LIST` hat. Deklariert man dafür wiederum einen neuen (irrelevanten) Signaturnamen, erfährt man aus der Mitteilung des SML-Systems die Schnittstelle der Struktur:

```
- signature SIG = LIST;
signature LIST =
  sig
    datatype 'a list = :: of 'a * 'a list | nil
    exception Empty
    val null : 'a list -> bool
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val last : 'a list -> 'a
    val getItem : 'a list -> ('a * 'a list) option
    val nth : 'a list * int -> 'a
    val take : 'a list * int -> 'a list
    val drop : 'a list * int -> 'a list
    val length : 'a list -> int
    val rev : 'a list -> 'a list
    val @ : 'a list * 'a list -> 'a list
    val concat : 'a list list -> 'a list
    val revAppend : 'a list * 'a list -> 'a list
    val app : ('a -> unit) -> 'a list -> unit
    val map : ('a -> 'b) -> 'a list -> 'b list
    val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
    val find : ('a -> bool) -> 'a list -> 'a option
    val filter : ('a -> bool) -> 'a list -> 'a list
    val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val exists : ('a -> bool) -> 'a list -> bool
    val all : ('a -> bool) -> 'a list -> bool
    val tabulate : int * (int -> 'a) -> 'a list
  end
```

Mit einer gewissen Erfahrung in der Listenverarbeitung reicht diese Information oft schon aus, um die geeignete Funktion für das gerade bearbeitete Problem zu finden.

Die oben erwähnte Seite der Standardbibliothek enthält im wesentlichen diese Signatur sowie eine kurze verbale Beschreibung aller Funktionen der Signatur.

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 12

Imperative Programmierung in SML

SML ist keine rein funktionale Programmiersprache, sondern eine funktionale Programmiersprache mit imperativen Sprachkonstrukten. Im Abschnitt 3.5 wurde der Begriff „Zustandsvariable“ eingeführt, die Zustandsvariablen mit den funktionalen Variablen verglichen und die Zustandsvariablen in SML, Referenzen genannt, behandelt. In diesem Kapitel wird zunächst an die SML-Programmierung mit Referenzen erinnert. Dann werden weitere imperative Konstrukte in SML, die while-Schleife und die Felder, erläutert. Letztlich werden die Ein- und Ausgaben in SML behandelt.

12.1 SML-Referenzen

Die Zustandvariablen in SML heißen Referenzen (siehe Abschnitt 3.5). Die SML-Referenzen sind „Zustandsvariablen mit expliziter Dereferenzierung“ (siehe Abschnitt 3.5.2).

12.1.1 Deklaration einer Referenz — Referenzierungsoperator in SML

Eine Referenz wird in SML unter Verwendung der Funktion `ref` deklariert (siehe Abschnitt 3.5.2):

```
- val kto = ref 100;
val kto = ref 100 : int ref
```

Man beachte, dass es in SML keine Möglichkeit gibt, eine Referenz zu deklarieren, ohne dabei der deklarierten Referenz einen Initialwert zuzuweisen.

Der Referenzierungsoperator in SML ist `ref`:

```
- val kto = ref 100;
val kto = ref 100 : int ref

- val v = ref kto;
val v = ref (ref 100) : int ref ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

12.1.2 Typ einer Referenz

Für jeden Monotyp `t` kann eine Zustandsvariable vereinbart werden, deren Inhalt ein Wert von Typ `t` ist. (Der Begriff „Monotyp“ wurde im Abschnitt 5.4.3 definiert.)

Die folgenden Deklarationen definieren die Referenzen `r` und `w` als Zustandsvariablen, deren Werte Funktionen vom Typ `int -> int` sind:

```
- val r = ref (fn x => 2 * x);
val r = ref fn : (int -> int) ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

Es ist in SML nicht möglich, eine Zustandsvariable, deren Inhalt ein Wert von einem Polytyp ist, zu vereinbaren, wie das folgende Beispiel zeigt. (Der Begriff „Polytyp“ wurde im Abschnitt 5.4.3 definiert.)

```
- val v = ref [];
stdIn:8.1-8.15 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val v = ref [] : ?.X1 list ref
```

Legt ein Typ-Constraint einen Typ für `[]` fest, so kann eine Zustandsvariable mit initialem Wert `[]` vereinbart werden, wie das folgende Beispiel zeigt:

```
- val v = ref ([] : string list);
val v = ref [] : string list ref
```

Referenzen, deren Inhalte Referenzen sind, sind möglich (siehe Abschnitt 3.5.3):

```
-  val w = ref (fn x:int => 0);
val w = ref fn : (int -> int) ref

- val z = ref w;
val z = ref (ref fn) : (int -> int) ref ref

- !(!z)(9);
val it = 0 : int
```

12.1.3 Dereferenzierungsoperator in SML

Der Dereferenzierungsoperator in SML ist „`!`”:

```
- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- !w(!v);
val it = 0 : int
```

Der Dereferenzierungsoperator verhält sich, als ob er wie folgt (unter Verwendung von Pattern Matching — siehe Kapitel 9) definiert wäre:

```
fun ! (ref x) = x
```

12.1.4 Sequenzierungsoperator in SML

Der Sequenzierungsoperator in SML ist „;“. Der Wert einer Sequenz von Ausdrücken (`A`$_1$; `A`$_2$; ...; `A`$_n$) ist der Wert des letzten Ausdrucks der Sequenz, also von `A`$_n$:

```
- 1; 2; 3;
val it = 1 : int
val it = 2 : int
val it = 3 : int

- val x = (1; 2; 3);
val x = 3 : int
```

Man beachte den Unterschied zum folgenden Beispiel:

```
- val x = 1; 2; 3;
val x = 1 : int
val it = 2 : int
val it = 3 : int
```

Im ersten Beispiel ist die Sequenz der definierende Ausdruck der Wertdeklaration. Im zweiten Beispiel ist die Wertdeklaration der erste Ausdruck der Sequenz.

12.1.5 Zuweisungsoperator in SML

Der Zuweisungsoperator in SML ist „:=“ (gesprochen *becomes* oder *erhält*). Der Wert einer Zuweisung ist in SML `()` (*unity*):

```
- val r = ref 1;
val r = ref 1 : int ref

- r := !r + 2;
val it = () : unit

- !r;
val it = 3 : int
```

Da die SML-Referenzen Zustandsvariablen mit expliziter Dereferenzierung (siehe Abschnitt 3.5.2) sind, kann in SML (im Gegensatz zu anderen Programmiersprachen) das vorangehende Beispiel nicht wie folgt fortgesetzt werden:

```
- r := r + 1;
Error: operator and operand don't agree [literal]
  operator domain: int ref * int ref
  operand:         int ref * int
  in expression:
    r + 1
Error: overloaded variable not defined at type
  symbol: +
  type: int ref
```

Der Inhalt der Referenz `r` kann aber wie folgt um 1 erhöht werden:

```
- r := !r + 1;
val it = () : unit

 - !r;
 val it = 4 : int
```

Da in SML der Wert `()` — vom Typ Typ `unit` — als Wert einer Zuweisung geliefert wird, hat der Zuweisungsoperator „`:=`“ den folgenden Polytyp (siehe 5.4.3):

```
'a ref * 'a -> unit.
```

12.1.6 Druckverhalten von SML bei Referenzen

SML liefert Referenzen, druckt sie aber nicht aus. Anstelle einer Referenz druckt SML das Symbol `ref` gefolgt von ihrem Inhalt:

```
- val v = ref 5;
val v = ref 5 : int ref

- v;
val it = ref 5 : int ref

-  val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- w;
val it = ref fn : (int -> int) ref
```

Dieses Druckverhalten von SML sollte nicht dazu verleiten, die Referenzen in SML für Zustandsvariablen ohne explizite Dereferenzierung zu halten!

12.1.7 Gleichheit zwischen Referenzen

Die Gleichheit für Referenzen in SML ist „`=`“.

```
- val v1 = ref 12;
val v1 = ref 12 : int ref

- val v2 = v1;
val v2 = ref 12 : int ref

- v1 = v2;
val it = true : bool
```

Sind zwei Referenzen gleich, so sind ihre Inhalte ebenfalls gleich:

```
- !v1 = !v2;
val it = true : bool
```

Die Umkehrung gilt selbstverständlich nicht: Zwei unterschiedliche Referenzen mit demselben Inhalt sind nicht gleich:

```
- ref 12 = ref 12;
val it = false : bool
```

Das folgende Beispiel mag verständlicher sein:

```
- val v1 = ref 12;
val v1 = ref 12 : int ref

- val v2 = ref 12;
val v2 = ref 12 : int ref

- v1 = v2;
val it = false : bool
```

Das Ergebnis der Auswertung des Ausdruckes `ref 12 = ref 12` zeigt, dass `ref` nicht referenztransparent (siehe Abschnitt 3.5.1), also nicht rein funktional (siehe Abschnitt 3.6.4) ist.

12.1.8 Vordefinierte SML-Prozeduren über Referenzen

Die Dereferenzierung, die Zuweisung und die Gleichheit sind die einzigen vordefinierten Prozeduren in SML über Referenzen.

Im Gegensatz zu anderen Programmiersprachen ermöglicht SML nicht, zu einer Referenz, d.h. einer Speicheradresse, eine Zahl zu addieren, um so auf eine darauffolgende Speicherzelle zuzugreifen. Diese Möglichkeit, die bei Maschinensprache sinnvoll ist, ist bei modernen höheren Programmiersprachen verpönt, weil sie zu schwer verständlichen und daher häufig fehlerhaften Programmen führt.

12.2 Iteration in SML

Zur Iteration bietet SML die While-Schleife (vgl. Abschnitt 4.3.3) an. Ein Beispiel der Verwendung der While-Schleife in SML ist wie folgt:

```
- val zaehler = ref 4;
- while ! zaehler > 0 do
     (print(Int.toString (!zaehler) ^ " "); zaehler := !zaehler - 1);
4 3 2 1 val it = () : unit
```

Ein `while`–Ausdruck in SML hat die folgende Gestalt:

```
while A1 do A2
```

Der Ausdruck `A1` ist die „Bedingung“ des `while`–Ausdrucks. Die Bedingung eines `while`–Ausdrucks muß vom Typ `bool` sein. Der Ausdruck `A2` wird *Rumpf* des `while`–Ausdrucks genannt. Sein Typ kann beliebig sein. Der Typ eines `while`–Ausdrucks ist `unit`.

Man beachte die Verwendung der Klammern und der Sequenz im Rumpf des `while`–Ausdrucks des vorangehenden Beispiels:

```
(print(Int.toString (!zaehler) ^ " "); zaehler := !zaehler - 1);
```

Die Fakultätsfunktion (siehe Abschnitt 4.3.3) kann in SML statt mittels Endrekursion auch wie folgt iterativ implementiert werden:

```
- fun fak n =
     let val zaehler = ref n;
         val akk     = ref 1;
     in
       while !zaehler <> 0 do
          (akk := !akk * !zaehler; zaehler := !zaehler - 1);
       ! akk
     end;
val fak = fn : int -> int

- fak 4;
val it = 24 : int
```

Der Ausdruck `! akk`, der unmittelbar nach dem `while`–Ausdruck im definierenden Teil der Prozedur `fak` vorkommt, dient lediglich dazu, den berechneten Wert als Wert der Anwendung der Prozedur `fak` zu liefern. Dieser Ausdruck `! akk` ist nicht Teil des Rumpfes des `while`–Ausdrucks.

12.3 SML-Felder

Ein Feld (*array*) ist einem Vektor ähnlich. Im Gegensatz zu den Komponenten eines Vektors sind jedoch die Komponenten eines Feldes veränderbar.

Felder nehmen in der imperativen Programmierung eine ähnliche Stellung wie Listen in der rein funktionalen Programmierung ein.

Die Struktur `Array` der Standardbibliothek von SML bietet Prozeduren an, mit denen Felder deklariert und verwendet werden können.

12.3.1 Deklaration eines Feldes

Felder können wie folgt deklariert werden:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array
```

Bei der Deklaration eines Feldes wird also ein (gleicher) Initialwert allen Komponenten des Feldes zugewiesen.

Felder können Komponenten von unterschiedlichen Typen haben, jedoch wie bei Listen müssen alle Komponenten eines Feldes denselben Typ haben:

```
- val int_feld = Array.array(5, 12);
val int_feld = [|12,12,12,12,12|] : int array

- val int_list_feld = Array.array(2, [1,2,3]);
val int_list_feld = [|[1,2,3],[1,2,3]|] : int list array
```

12.3.2 Zugriff auf die Komponenten eines Feldes

Es wird wie folgt auf die Komponenten eines Feldes zugegriffen:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.sub(f, 0);
val it = "a" : string

- Array.sub(f, 3);
uncaught exception subscript out of bounds
raised at: stdIn:47.1-47.10
```

Man beachte, dass die erste Komponente eines Feldes mit 3 Komponenten die Komponente mit Nummer 0, die letzte die Komponente mit Nummer 2 ist.

12.3.3 Veränderung der Komponenten eines Feldes

Die Komponenten eines Feldes können wie folgt verändert werden:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.update(f, 1, "b");
val it = () : unit

- Array.sub(f, 1);
val it = "b" : string

- Array.update(f, 2, "c");
val it = () : unit

- val zaehler = ref 0;
val zaehler = ref 0 : int ref

- while !zaehler < 3 do
     (print(Array.sub(f, !zaehler) ^ " "); zaehler := !zaehler + 1);
a b c val it = () : unit
```

12.3.4 Länge eines Feldes

Die Funktion `Array.length` liefert die Länge eines Feldes:

```
- val  f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.length f;
val it = 3 : int
```

12.3.5 Umwandlung einer Liste in ein Feld

Es ist möglich, wie folgt aus einer Liste ein Feld zu erzeugen:

```
- Array.fromList [1, 2, 3];
val it = [|1,2,3|] : int array
```

12.3.6 Umwandlung eines Feldes in eine Liste

Aus einem Feld kann wie folgt eine Liste erzeugt werden:

```
- fun array_to_list(a : 'a array) =
     let val length     = Array.length a;
         val counter_rf = ref length;
         val list_rf    = ref [] : 'a list ref
     in
       while ! counter_rf > 0 do
          ( counter_rf := ! counter_rf - 1
          ; list_rf := Array.sub(a, ! counter_rf) :: ! list_rf
          );
       ! list_rf
     end;
val array_to_list = fn : 'a array -> 'a list

- array_to_list(f);
val it = ["a","b","c"] : string list
```

12.3.7 Gleichheit für Felder

Die Gleichheit zwischen Feldern ist referenzbezogen. Folglich liefert die folgende Gleichung

```
- Array.fromList [1, 2, 3] = Array.fromList [1, 2, 3];
val it = false : bool
```

genauso wie die Gleichung

```
- ref 12 = ref 12;
val it = false : bool
```

(siehe Abschnitt 12.1) den Wert `false`.

12.3.8 Hinweis auf die Standardbibliothek von SML

Die Standardbibliothek stellt eine Sammlung von Modulen (Strukturen) zur Verfügung, die weitere Funktionen zur Programmierung mit Felder anbieten — siehe http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/.

12.4 Beispiel: Sortieren eines Feldes durch direktes Einfügen (straight insertion sort)

12.4.1 Totale Ordnung

Eine Ordnung $<$ über eine Menge M ist eine binäre Relation über M, d.h. eine Teilmenge des Kartesischen Produkts $M \times M$, die die folgenden Eigenschaften erfüllt:

- $<$ ist reflexiv: für alle $m \in M$ gilt $x < x$
- $<$ ist antisymmetrisch: für alle $m_1, m_2 \in M$, falls $m_1 < m_2$ und $m_2 < m_1$, dann $m_1 = m_2$
- $<$ ist transitiv: für alle $m_1, m_2, m_3 \in M$, falls $m_1 < m_2$ und $m_2 < m_3$, dann $m_1 < m_3$

Eine Ordnung $<$ über einer Menge M heißt *total*, wenn für alle $m_1 \in M$ und $m_2 \in M$, $m_1 < m_2$ oder $m_2 < m_1$.

12.4.2 Sortieren

Unter dem Sortieren von Elementen aus einer Menge M mit (totaler) Ordnung $<$ versteht man das Anordnen der gegebenen Elemente in auf- oder absteigender Reihenfolge bezüglich der Ordnung $<$.

Das Sortieren dient im Allgemeinen dazu, die Suche nach einem Element in der gegebenen Menge zu vereinfachen.

Das Sortieren ist in Alltagsleben wie in der Informatik eine grundlegende Tätigkeit: Zum Beispiel werden die Ergebnisse einer Klausur sortiert nach Matrikelnummern (bezüglich der Ordnung $\leq$ über den natürlichen Zahlen) veröffentlicht und Telefonbücher sind sortiert nach Familienname (bezüglich der sogenannten lexikographischen Ordnung über Buchstabenfolgen). In der Informatik beruhen viele Algorithmen auf dem Sortieren von Daten.

Man unterscheidet in der Informatik zwischen dem Sortieren von Felder, auch *internes Sortieren* genannt, und das Sortierten im Dateien, auch *externes Sortieren* genannt.

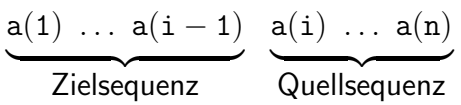
Das interne Sortieren wird verwendet, wenn Datensätze im Hauptspeicher sortiert werden. Dabei setzt man oft voraus, dass kein oder nur sehr wenig weiterer Speicherplatz als das zu sortierende Feld zur Verfügung steht. Diese Annahme war in der Vergangenheit wichtig, als der verfügbare Speicherplatz sehr klein war. Sie ist heute nur noch in manchen Fälle wie z.B. für „Mobile Computing“ oder beim „Wegwerf-Computer“ wichtig.

Das externe Sortieren geht auf die Zeit zurück, wo große Datenmengen auf Bändern gespeichert wurden, worauf ausschließlich sequenziell, also ohne sogenannte direkte Zugriffe wie auf Felder, geschrieben wurde. Um den Inhalt eines Bandes zu sortieren, verwenden externe Sortierverfahren mindestens ein weiteres Band, auf dem Zwischenergebnisse aufgenommen werden.

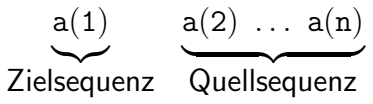
In der Praxis werden Datensätze sortiert, die aus einem sogenannten *Schlüssel* sowie weiteren Werten bestehen. Der *Schlüssel* ist ein möglicherweise künstlich hinzugefügter Teil des Datensatzes, über dessen Typ eine (totale) Ordnung definiert ist. Der Einfachheit halber betrachten wir im Folgenden Schlüssel, die ganze Zahlen sind, die totale Ordnung $\leq$ über den ganzen Zahlen, und Datensätze, die nur aus einem Schlüssel bestehen.

12.4.3 Internes Sortieren durch direktes Einfügen (straight insertion sort)

Diese Sortiermethode wird oft von Kartenspieler verwendet. Die Datensätze (etwa Karten) werden in eine Zielsequenz `a(1), ..., a(i-1)` und eine Quellsequenz `a(i), ..., a(n)` aufgeteilt:

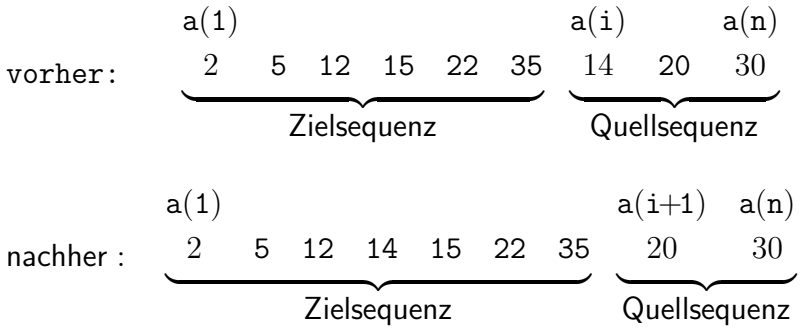


Anfangs enthält die Zielsequenz nur `a(1)`, die Quellsequenz enthält alle Datensätze von `a(2)` bis `a(n)`:



Wir nehmen an, dass nach aufsteigenden Schlüsseln sortiert werden soll.

Beginnend mit `i = 2` wird der erste Datensatz `a(i)` der Quellsequenz in die Zielsequenz an dieser Stelle eingefügt, dass sich eine um einen Datensatz längere sortierte Zielsequenz ergibt. Dafür müssen alle Elemente der Zielsequenz, die echt größer als `a(i)` sind, um eine Stelle nach rechts verschoben werden, wie das folgende Beispiel zeigt:



Für jeden Wert von `i` zwischen `2` und `n` wird `a(i)` wie folgt in die Zielsequenz eingefügt.

Angefangen mit `k = i - 1` bis `k = 1` wird `a(i)` mit `a(k)` verglichen. Gilt `a(k) > a(i)`, so werden die Werte von `a(k)` und `a(i)` vertauscht.

Um das Vertauschen zu ermöglichen, muss einer der Werte von `a(k)` und `a(i)` in einer Zustandsvariable aufgenommen werden. Da `a(i)` in der Regel mit mehreren Feldkomponenten `a(k)` verglichen wird, empfehlt es sich, den Wert von `a(i)` „zwischenzuspeichern“. Sei die Zustandsvariable dazu `z` genannt.

Es ergibt sich also der folgende Algorithmus, der in einem nicht weiter formalisierten Pseudo-Code (mit Zustandsvariablen mit expliziter Dereferenzierung) ausgedruckt ist:

```
i := 2;
while ! i <= n do
   z := ! a(!i);
   k := ! i;
   while ! k >= 1 and ! z < ! a(!k-1) do
          a(!k) := ! a(!k-1);
          k := ! k - 1
   end-while;
   a(!k) := ! z;
   i := ! i + 1
end-while
```

Die Abbruchbedingung

```
! k >= 1 and ! z < ! a(!k-1)
```

der inneren `while`–Schleife kann in

```
! a(!i) < ! a(!k-1)
```

vereinfacht werden, wenn anstelle der Zustandsvariable `z` eine Feldkomponente `a(0)` verwendet wird. Ist der Wert von `a(0)` derselbe wie der Wert von `a(!i)`, so ist die Bedingung

```
! a(i) < ! a(k - 1)
```

verletzt, wenn `k` = 1.
So ergibt sich der folgende (verbesserte) Algorithmus:

```
i := 2;
while !i <= n do
   a(0) := ! a(!i);
   k := ! i;
   while ! a(!i) < ! a(!k-1) do
          a(!k) := ! a(!k-1);
          k := ! k - 1
   end-while;
   a(!k) := ! a(0);
   i := ! i + 1
end-while
```

Dieser Algorithmus kann wie folgt in SML implementiert werden, wobei die zu sortierende Sequenz als Liste ein- und ausgegeben wird. Als Initialwerte der Feldkomponente `a(i)` und des Zählers `k` wird willkürlich 0 gewählt:

```
- fun direktes_einfuegen(sequenz : int list) : int list =
         let val a = Array.fromList(0 :: sequenz);
             val i = ref 2;
             val k = ref 0;
             val n = Array.length a - 1;
```

```
        in
          while ! i <= n do
             ( Array.update(a, 0, Array.sub(a, ! i));
               Array.update(a, 0, Array.sub(a, 0));
               k := ! i;
               while Array.sub(a, 0) < Array.sub(a, ! k - 1) do
                  ( Array.update(a, ! k,  Array.sub(a, ! k - 1));
                    k := ! k - 1
                  );
               Array.update(a, ! k, Array.sub(a, 0));
               i := ! i + 1
             );
          tl(array_to_list a)
        end;
val direktes_einfuegen = fn : int list -> int list
```

Die Prozedur `array_to_list` ist im Abschnitt 12.3.6 definiert.

```
- direktes_einfuegen([5, 2, 4, 1, 3, 9, 7, 6, 0, 8]);
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

12.4.4 Komplexität des internen Sortierens durch direktes Einfügen

Während des `i`-ten Durchlaufs der äußeren While-Schleife werden mindestens `1` und höchstens $\texttt{i} - 1$ Schlüssel verglichen. Sind alle Permutationen von `n` Schlüsseln in der sortierenden Sequenz gleich wahrscheinlich, so werden im Durchschnitt während des `i`-ten Durchlaufs $\texttt{V} = \texttt{i}/2$ Schlüssel verglichen.

Die Anzahl der Feldkomponenten, die während des `i`-ten Durchlaufs einen neuen Wert erhalten, beträgt $\texttt{V} + 2$, weil am Anfang des Durchlaufs `a(0)` den Wert von `a(i)` erhält.

So sind die Gesamtzahlen der Schlüsselvergleiche und Aktualisierungen von Feldkomponenten (Übung!):

	Mindestzahl	Durchschnitt	Höchstzahl
Schlüsselvergleiche	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2$
Aktualisierungen	$3(n-1)$	$(n^2+9n-10)/4$	$(n^2+3n-3)/2$

Man kann (leicht) zeigen, dass die Mindestzahlen vorkommen, wenn die zu sortierende Sequenz bereits sortiert ist, die Höchstzahlen, wenn die zu sortierende Sequenz in umgekehrter Reihenfolge angeordnet ist.

Wird als Größe des Problems die Länge der zu sortierenden Sequenz angesehen, und als Zeiteinheit ein Vergleich oder eine Aktualisierung, so ist der durchschnittliche Zeitbedarf des Sortierens durch direktes Einfügen $O(n^2)$.

Das Sortieren durch direktes Einfügen ist kein guter Sortieralgorithmus. Ein wesentlich besserer durchschnittlicher Zeibedarf wird von Sortieralgorithmen erreicht, die die falsch angeordneten Schlüssel in einem Schritt über größere Entfernungen bewegen.[1]

[1] Weitere Sortieralgorithmen werden in der Grundstudiumvorlesung Informatik 2 eingeführt.

12.5 Ein- und Ausgaben in SML

12.5.1 Datenströme (streams)

Das Paradigma von SML — wie auch von vielen anderen Programmiersprachen — für die Ein- und Ausgaben ist das Paradigma des „Datenstroms“ (*stream*). Ein Datenstrom stellt in SML eine Folge von Datensätzen vom Typ Zeichen, Zeichenfolgen oder auch Binärzahl dar, deren Länge unbegrenzt ist.

Die Datensätze eines Datenstroms werden nacheinander zwischen dem „Datenerzeuger“ (*producer*) des Datenstroms und dem „Datenverbraucher“ (*consumer*) des Datenstroms verschickt.

Man unterscheidet zwischen „Eingabedatenströme“ (*input streams*) und „Ausgabedatenströme“ (*output streams*).

Der Datenerzeuger eines Eingabedatenstroms kann bei einer interaktiven SML-Sitzung die Tastatur des Terminals sein. Der Datenerzeuger eines Eingabedatenstroms kann auch eine Datei sein, deren Inhalt sequenziell eingelesen wird. Der Datenverbraucher eines Eingabedatenstroms ist das Programm, in dem dieser Eingabedatenstrom deklariert ist.

Der Datenerzeuger eines Ausgabedatenstroms ist das Programm, in dem dieser Ausgabedatenstroms deklariert ist. Der Datenverbraucher eines Ausgabedatenstroms kann bei einer interaktiven SML-Sitzung der Bildschirm sein. Der Datenverbraucher eines Ausgabedatenstroms kann auch ein Drucker oder eine Datei sein, in die die Datensätze des Ausgabedatenstroms geschrieben werden.

Ein Datenstrom wird in SML durch eine Deklaration erzeugt. Dabei erhält ein Datenstrom einen Namen, der ein gewöhnlicher SML-Name (oder SML-Bezeichner) ist. Die Deklaration eines Eingabedatenstroms spezifiziert zudem einen Datenerzeuger, die Deklaration eines Ausgabedatenstroms einen Datenverbraucher.

Mit der Deklaration eines Eingabe- oder Ausgabedatenstroms wird auf Ressourcen des zugrundeliegenden Betriebssystems zugegriffen. Man sagt, dass der Datenstrom „geöffnet“ wird.

Damit sparsam mit Betriebssystemressourcen umgegangen werden kann, können (und sollen!) Datenströme, die nicht mehr benötigt werden, geschlossen werden.

In SML stellen Datenströme Typen dar, die über vordefinierte Prozeduren verfügen. Mit diesen Prozeduren können Datenströme deklariert, d.h. erzeugt, und geschlossen werden und kann die Datenübertragung mit einem Datenstrom gesteuert werden.

Die vordefinierten Datenstromprozeduren sind selbstverständlich nicht referenztransparent (siehe Abschnitt 3.5.1, 3.6.2 und 4.1.3), weil der Aufruf zu unterschiedlichen Zeitpunkten desselben Ausdrucks zur Steuerung eines Datenstroms unterschiedliche Folgen haben mag. Die Datenstromprozeduren sind folglich nicht rein funktional. Sie sind also im strengen Sinne keine Funktionen (siehe die Abschnitte 3.6.2 und 4.1.3).

SML bietet Datenströme von den zwei folgenden Arten an:

- Datenströme von Zeichen (SML-Typ `char`) oder Zeichenfolgen (SML-Typ `string`) — auch „Textströme“ genannt;
- Datenströme von Binärdaten dargestellt als Worte einer Länge von 8 Bits (= 1 Byte) — auch „Binärströme“ genannt.

Die Datenstromprozeduren für beide Datenstromarten sind in der SML-Standard-

bibliothek definiert:

Das Modul (oder Struktur) `TextIO` der SML-Standardbibliothek enthält die Prozeduren zur Verwendung von Textströmen. Das Modul `BinIO` der SML-Standardbibliothek enthält die Prozeduren zur Verwendung von Binärströmen.

Im Folgenden werden einige Prozeduren zur Verwendung von Textströmen eingeführt.

Viele Module (oder Strukturen) der SML-Standardbibliothek enthalten besonders ausgereifte Funktionen und Prozeduren zur Textverarbeitung, womit grundlegende Dienste der Textverarbeitung und der Programmübersetzung wie etwa Worterkennung, Formatierung, Übersetzung von Zahlen in Zeichenfolgen und lexikalische Analyse leicht implementiert werden können.

12.5.2 Ausgabestrom

Ein Ausgabetextstrom namens `strom1` mit einer Datei namens `datei1` als Datenverbraucher kann wie folgt deklariert (oder geöffnet) werden:

```
- val strom1 = TextIO.openOut "datei1";
val strom1 = - : TextIO.outstream
```

Die Mitteilung von SML auf diese Deklaration zeigt, dass der Typ eines Ausgabetextstroms `TextIO.outstream` ist.

Existiert die Datei `datei1` noch nicht, so wird sie mit dieser Deklaration erzeugt. Die Datei `datei1` mag aber schon vorhanden sein, wenn die vorangehende Deklaration ausgewertet wird. Die Verwendung einer vorhandenen Datei als Datenverbraucher eines Ausgabetextstroms führt aber zum Verlust des vorherigen Inhalts dieser Datei.

Der Ausgabetextstrom `strom1` kann wie folgt verwendet werden, um in die Datei `datei1` die Zeichenfolge „Erster Schreibtest“ zu schreiben:

```
- TextIO.output(strom1, "Erster Schreibtest");
val it = () : unit
```

Die Zeichen neue Zeile (*newline*) und *Tab* werden in SML „`\n`“ bzw. „`\t`“ genannt. Das Zeichen \ wird in SML mit „`\\`“ erzeugt (siehe Abschnitt 5.2.4). Die Folge:

\ gefolgt von white-space-Zeichen gefolgt von \

ermöglicht es, white-space–Zeichen wie *newline*, *tab* oder Leerzeichen, die zur lesbareren Darstellung eines Programms nützlich sind, innerhalb einer SML-Zeichenfolge zu ignorieren (siehe Abschnitt 5.2.4).

Ausgabeströme übergeben ihre Daten an ihren Datenverbraucher nicht immer sofort. Verzögerungen können auftreten, die durch die Verwaltung der auf dem Prozessor gleichzeitig laufenden Prozesse durch das Betriebssystem bedingt sind. Die Prozedur `TextIO.flushOut` bewirkt, dass alle vom Ausgabestrom noch nicht ausgegebenen Datensätze an den Verbraucher des Ausgabestroms weitergeleitet werden:

```
- TextIO.flushOut strom1;
val it = () : unit
```

Der Ausgabetextstrom `strom1` kann wie folgt geschlossen werden:

```
- TextIO.closeOut strom1;
val it = () : unit
```

12.5.3 Eingabestrom

Sei `datei2` eine Datei, deren Inhalt die Zeichenfolge

```
"Erster Lesetext\n"
```

ist.
Eine Eingabetextstrom namens `strom2` mit einer Datei `datei2` als Datenerzeuger kann wie folgt deklariert (oder geöffnet) werden:

```
- val strom2 = TextIO.openIn "datei2";
val strom2 = - : TextIO.instream
```

Die Mitteilung von SML auf diese Deklaration zeigt, dass der Typ eines Eingabetextstroms `TextIO.instream` ist.
Der Eingabetextstrom `strom2` wird wie folgt verwendet, um das erste Zeichen des Stromes, d.h. in diesem Fall der Datei `datei2` zu erhalten:

```
- val zeichen = TextIO.input1(strom2);
val zeichen = SOME #"E" : TextIO.elem option
```

Das Ergebnis ist ein Wert des zusammengesetzten Typs `TextIO.elem option`. Der Polytyp (siehe Abschnitt 5.4.3) `option` verfügt über zwei (Wert-)Konstruktoren `SOME` und `NONE`. Mit diesem Polytyp kann jeder Typ `'a` so erweitert werden, dass unter zulässigen und unzulässigen Werten unterschieden werden kann.
`elem` ist der abstrakte Typ der Datensätze eines Eingabe- oder Ausgabetextstroms.
Eine beliebige Anzahl n, im Folgenden $n = 6$, von Datensätzen aus dem Eingabetextstrom `strom2` kann wie folgt erhalten werden:

```
- val zeichenfolge = TextIO.inputN(strom2, 6);
val zeichenfolge = "rster " : TextIO.vector
```

Man beachte, dass die 6 Zeichen nach dem bereits ausgegebenen ersten Zeichen „`E`“ geliefert werden. Die gelieferte Zeichenfolge hat den Typ `TextIO.vector`, der nicht mit den herkömmlichen SML-Vektoren (siehe Abschnitt 5.3.1) verwechselt werden darf.
Die übrigen Datensätze des Eingabetextstroms `strom2` können wie folgt erhalten werden:

```
- val rest = TextIO.input(strom2);
val rest = "Lesetest\n" : TextIO.vector
```

Der Eingabetextstrom `strom2` wird wie folgt geschlossen:

```
- TextIO.closeIn strom2;
val it = () : unit
```

12.5.4 Standard–Ein- und -Ausgabetextströme `TextIO.stdIn` und `TextIO.stdOut`

Die sogenannten Standard-Ein- und -Ausgaben sind in SML — wie in anderen Programmiersprachen auch — Textströme. Sie heißen `TextIO.stdIn` und `TextIO.stdOut`.
So kann die Zeichenfolge `"abcd\n"` am Bildschirm ausgegeben werden.

```
- val bildschirm = TextIO.stdOut;
val strom3 = - : TextIO.outstream

- TextIO.output(bildschirm, "abcd\n");
abcd
val it = () : unit
```

Zeichenfolgen können wie folgt von der Tastatur eingelesen werden:

```
- val tastatur = TextIO.stdIn;
val tastatur = - : TextIO.instream

- TextIO.input(tastatur);
123456\n
val it = "123456\n" : TextIO.vector
```

12.5.5 Die vordefinierte Prozedur `print`

SML bietet auch die vordefinierte Prpozedur `print`, die wir uns schon mehrmals benutzt haben (siehe z.B. Abschnitte 11.2.3, 11.3.1 und 12.2), um Zeichenfolgen an die Standardausgabe zu leiten:

```
- print "abc\n";
abc
val it = () : unit
```

Obwohl `print` im Modul (Struktur) `TextIO` definiert ist, ist diese Prozedur nicht nur durch ihren langen Namen (siehe Abschnitt 11.4) `TextIO.print` aufrufbar, sondern auch unter dem Namen `print`, wie das folgende Beispiel zeigt:

```
- TextIO.print "abc\n";
abc
val it = () : unit

- print "abc\n";
abc
val it = () : unit
```

12.5.6 Beispiel: Inhalt einer Datei einlesen und an die Standardausgabe weiterleiten

Die folgende Prozedur liest eine Datei namens `datei` ein und leitet ihren Inhalt an die Standardausgabe `TextIO.stdOut`, d.h. an den Bildschirm, weiter:

```
- fun cat datei =
   let val datei = TextIO.openIn datei;
       val zeichenfolge = ref "";
   in
     while ( zeichenfolge := TextIO.inputN(datei, 1)
           ; ! zeichenfolge <> ""
           )
     do
        TextIO.output(TextIO.stdOut, ! zeichenfolge);
     TextIO.closeIn datei
   end;
val cat = fn : string -> unit
```

Sei `datei2` eine Datei, deren Inhalt die Zeichenfolge

```
"Erster Lesetext\n"
```

ist.

```
- cat "datei2";
Erster Lesetest
val it = () : unit
```

12.5.7 Hinweis auf das Modul `TextIO` der Standardbibliothek von SML

Die vorangehenden Abschnitte sind eine oberflächige Einführung in die Verwendung von Textströmen in SML. Für weitere Informationen dazu siehe das Modul `TextIO` der Standardbibliothek von SML (unter http://www.dina.kvl.dk/%7Esestoft/sml/text-io.html).

© François Bry (2001, 2002)
Dieses Lehrmaterial wird ausschließlich zur privaten Verwendung angeboten. Eine nichtprivate Nutzung (z.B. im Unterricht oder eine Veröffentlichung von Kopien oder Übersetzungen) dieses Lehrmaterials bedarf der Erlaubnis des Autors.

Kapitel 13

Formale Beschreibung der Syntax und Semantik von Programmiersprachen

Für jede Programmiersprache müssen

1. die Syntax der Programmiersprache (d.h., welche Programme zur Auswertung zulässig sind) und
2. die Semantik der Programmiersprache (d.h., wie zulässige Programme auszuwerten sind)

festgelegt werden. Syntax und Semantik einer Programmiersprache müssen präzis definiert werden, damit zum einen Programmierer die Programmiersprache zielsicher verwenden können, zum anderen alle Implementierungen dieser Programmiersprache dasselbe „Verhalten“ aufweisen, d.h. dieselben Programme zulassen und die zugelassenen Programme „in gleicher Weise“ auswerten. Zur präzisen Definition von Syntax und Semantik einer Programmiersprache werden formale, d.h. mathematische, Methoden eingesetzt. Dieses Kapitel führt in diese Methoden ein.

13.1 Formale Beschreibung der Syntax einer Programmiersprache

13.1.1 Syntax versus Semantik

Nicht jeder Ausdruck ist ein zulässiger SML-Ausdruck. So wird z.B. der folgende Ausdruck von einem SML-System abgelehnt:

```
- val 2mal x => 2 * x;
Error: syntax error: deleting DARROW INT
```

In der Tat weist dieser Ausdruck mehrere Mängel auf:

1. Zur Deklaration einer Funktion in SML darf das reservierte Wort `val` nur dann verwendet werden, wenn der definierende Ausdruck ein `fn`–Ausdruck ist. Andernfalls

muss zur Deklaration einer Funktion in SML das reservierte Wort `fun` verwendet werden.

2. Der Name `2mal` ist in SML nicht zulässig, weil er mit einer Zahl anfängt.

3. In SML darf das Symbol `=>` nur innerhalb von `fn`– oder `case`–Ausdrücken vorkommen.

Alle diese Mängel betreffen den Aufbau des Ausdrucks, nicht seine vermeintliche Bedeutung, die leicht zu erraten ist.

Inkorrekt gebildete Ausdrücke können festgestellt werden, bevor sie ausgewertet werden. Das Aufspüren von inkorrekt gebildeten Ausdrücken kann also statisch (man sagt auch zur Übersetzungszeit) erfolgen (vgl. Abschnitt 2.7, 4.2 und 6.2).

Die Syntax einer Programmiersprache bestimmt, ob Ausdrücke dieser Programmiersprache korrekt aufgebaut sind. Syntaxfehler wie in der Deklaration von `2mal` können immer (in endlicher Zeit) festgestellt werden. Die syntaktische Korrektheit eines Ausdrucks, d.h. die Fehlerfreiheit des Aufbaus dieses Ausdrucks, kann ebenfalls immer (in endlicher Zeit) festgestellt werden.

Weil sowohl Syntaxfehler als auch die syntaktische Korrektheit in endlicher Zeit feststellbar sind, sagt man, dass die syntaktische Korrektheit (und die syntaktische Inkorrektheit) von Ausdrücken eine entscheidbare Eigenschaft ist. Eine Eigenschaft E heißt *entscheidbar*, wenn es einen Algorithmus gibt, der in endlicher Zeit antwortet, ob E erfüllt oder nicht erfüllt ist.

Im Gegensatz zu der Deklaration von `2mal` ist der folgende Ausdruck korrekt gebildet, was die SML-Mitteilung belegt:

```
- fun fak1 x = if x > 0
               then x * fak1 x - 1
               else 1;
val fak1 = fn : int -> int
```

Dieser Ausdruck ist jedoch keine korrekte Deklaration der Fakultätsfunktion: Aufgrund der von SML festgelegten Präzedenzen (siehe Abschnitt 2.3) steht er für die folgende Deklaration:

```
- fun fak1 x = if x > 0
                then x * (fak1 x) - 1
                else 1;
val fak1 = fn : int -> int
```

Die Anwendung der Funktion `fak1` auf eine echt positive ganze Zahl `x` führt also zum rekursiven Aufruf von `fak1 x`. Folglich terminiert die Auswertung von `fak1 x` nicht.

Die Nichtterminierung der Funktion `fak1`, wenn sie auf echt positive Zahlen angewandt wird, kann experimentell beobachtet werden. Ein Experiment wie die Auswertung von `fak1 4` lässt vermuten, dass eine solche Auswertung nicht terminiert, kann dies aber nicht feststellen. So lange eine Auswertung schon gedauert haben mag, kann sie im nächsten Moment beendet werden.

Die Nichtterminierung der Funktion `fak1`, wenn sie auf echt positive natürliche Zahlen angewandt wird, kann auch durch den Versuch, einen Terminierungsbeweis zu finden,

festgestellt werden. Ein solcher Versuch würde zweifelsohne den Mangel der Deklaration erkennen lassen, so dass man anstatt der Terminierung die Nichtterminierung beweisen würde.

Der Mangel in der Deklaration der Funktion `fak1` ist nicht syntaktischer, sondern semantischer Natur. Die beabsichtigte (mathematische) Funktion, nämlich die Fakultätsfunktion, wird durch die vorangehende SML-Funktion `fak1` nicht implementiert.

Ein zentraler Satz der Informatik besagt, dass die Terminierung eines Programms eine sogenannte *semi-entscheidbare* Eigenschaft ist, was heißt:

1. Ein Algorithmus ist möglich, der die Terminierung eines Programms überprüft und nach endlicher Zeit die Terminierung meldet, falls das Programm terminiert.

2. Kein Algorithmus ist möglich, der die Terminierung eines Programms überprüft und nach endlicher Zeit meldet, ob das Programm terminiert oder nicht terminiert.

Für jeden Algorithmus, der die Terminierung von Programmen überprüft, wird es also immer Programme geben, die nicht terminieren und dessen Nichtterminierung der Algorithmus nicht feststellt.

So unterscheiden sich die Mängel grundsätzlich, die in den Deklarationen von `2mal` und `fak1` auftreten:

1. Die syntaktische Korrektheit eines Ausdrucks (oder Programms) ist entscheidbar.

2. Viele wichtige semantische Eigenschaften von Programmen, wie u.a. die Terminierung, sind hingegen semi–entscheidbar.

13.1.2 Ziele von formalen Beschreibungen der Syntax von Programmiersprachen

Formale Beschreibungen der Syntax von Programmiersprachen dienen Programmierern als Gebrauchsanweisungen. Sie dienen auch als Grundlage für die Implementierung der Programmiersprache. Sie ermöglichen zudem, automatisch aus der (formalen) Syntaxbeschreibung einen Algorithmus abzuleiten, der die Korrektheit von Ausdrücken gemäß dieser Syntax überprüft.

Automatisch heißt hier, dass eine Prozedur höherer Ordnung (siehe Kapitel 7) eine Syntaxbeschreibung `SB` als Aufrufparameter erhält und daraus einen „Syntaxprüfer“ gemäß `SB`, d.h. eine Prozedur `syntax_prüfer(SB)`, liefert.

Angewandt auf einen Ausdruck `expr` überprüft das Programm `syntax_prüfer(SB)`, ob `expr` der Syntaxbeschreibung `SB` entspricht oder nicht.

13.1.3 Lexikalische Analyse und Symbole versus Syntaxanalyse und Programme

Die Syntaxprüfung eines Programms (gemäß einer gegebenen Syntaxbeschreibung) besteht aus den folgenden zwei Phasen:

- Die *lexikalische Analyse*, d.h. die Überprüfung der Syntax der reservierten Namen (in SML wie etwa `fun`, `val`, `case`, `*`, `div`) und vom Programmierer frei ausgewählten

Namen (wie etwa `2mal`, `fak1`). Namen werden in diesem Kontext auch *Symbole* oder *Lexeme* (Englisch *token*) genannt.

- Die *syntaktische Analyse* (oder *Syntaxanalyse*), d.h. die Überprüfung der korrekten Anordnung der Symbole (oder Lexeme) in einem Programm. Während der Syntaxanalyse wird auch aus dem Quellprogramm in konkreter Syntax ein Programm in abstrakter Syntax erzeugt (vgl. Abschnitt 10.2.1).

Ein Programm zur lexikalischen Analyse wird Symbolentschlüssler (auch *Lexer*, *Scanner*) genannt. Ein Programm zur Syntaxanalyse wird Zerteiler (oder *Parser*) genannt.

13.1.4 EBNF-Grammatiken zur formalen Syntaxbeschreibungen

Die sogenannten alphabetischen Namen von SML fangen mit einem (kleinen oder großen) Buchstaben an, der gefolgt wird von endlich vielen (auch null) Buchstaben (`a ... z A ... Z`), Ziffern (`0 1 2 ... 9`), Underscore (`_`) oder Hochkommata (single quote: `'`) (siehe Abschnitt 2.4.8).

Eine Notation verwendend, die zuerst zur Beschreibung der Syntax der Programmiersprache Algol 60 (Algol ist eine Kürzel von Algorithmic Language and 60 bezeichnet den Geburtsjahrgang 1960 von Algol), kann die Definition der alphabetischen Namen von SML wie folgt mit vier sogenannten EBNF-Regeln formalisiert werden. EBNF steht für *Erweiterte Backus-Naur-Form* (nach den Namen der Erfinder der Notation):

Startsymbol:		AlphaName
AlphaName	::=	Anfangszeichen Folgezeichen * .
Anfangszeichen	::=	Buchstabe .
Folgezeichen	::=	Buchstabe | "0" | "1" | "2" | ... | "9" | "_" | "'" .
Buchstabe	::=	"a" | "b" | "c" | ... | "x" | "y" | "z" | "A" | "B" | "C" | ... | "X" | "Y" | "Z" .

Dabei ist AlphaName das Startsymbol, und die EBNF-Regeln haben paarweise unterschiedliche linke Seiten (vor dem Zeichen ::=). Das Zeichen | im rechten Teil einer EBNF-Regel trennt Alternativen.

Die erste Regel besagt, dass ein AlphaName aus genau einem Buchstabe gefolgt von beliebig vielen (eventuell null) Folgezeichen besteht.

Die zweite Regel definiert ein Folgezeichen als einen Buchstaben oder 0 oder 1 ... oder 9.

Die dritte Regel definiert einen Buchstaben in ähnlicher Weise.

Die Symbole

AlphaName	Anfangszeichen	Folgezeichen	Buchstabe
	::= * . " |		

gehören zur Spezifikationssprache EBNF („::=“ wird manchmal als *becomes* gelesen; „*“ wird Kleene'scher Stern genannt). Man nennt sie Metasymbole, weil sie einer Sprache angehören, die zur Spezifikation einer anderen Sprache, der Sprache der SML-Namen, verwendet wird. Die spezifizierte Sprache, hier die Sprache der SML-Namen, wird Objektsprache genannt.

Das Zeichen „. . . “ hingegen ist eine Kürzung, die nicht zur Spezifikationssprache EBNF gehört. Diese Kürzung wird hier verwendet, weil die vollständigen Regeln sehr lang sind.

Das Metasymbol „*“ bedeutet eine n-fache ($n \in \mathbb{N}$, eventuell $n = 0$) Wiederholung des vorstehenden Symbols. Das Metasymbol „.“ markiert das Ende einer EBNF-Regel. Das Zeichen „”“ wird verwendet, um Symbole der Objektsprache einzuführen wie etwa in „4“. Das Symbol „|“ bezeichnet eine Alternative.

Weitere Metasymbole sind „+“ und „[ ]“, die die folgende Bedeutung haben:

Das Metasymbol „+“ bedeutet eine n-fache ($n \in \mathbb{N}$ und $n \geq 1$) Wiederholung des vorstehenden Symbols.

[A] drückt eine Option aus: [A] bedeutet kein oder ein Vorkommen von A.

Die Klammern „)“ und „(“ sind weitere Metasymbole, womit „|“, „*“ und „+“ auf zusammengesetzte Ausdrücke angewandt werden können wie etwa in:

Ternärzahl ::= ”0” | (”1” | ”2”) (”0” | ”1” | ”2”) * .

Diese EBNF-Regel definiert Zahlen zur Basis 3 ohne führende Null. Die nächste Regel definiert Zahlen zur Basis 4, eventuell mit führender Null oder Vorzeichen:

Quatärzahl ::= [”+” | ”-”] (”0” | ”1” | ”2” | ”3”) * .

Metasymbole wie

AlphaName Anfangszeichen Folgebuchstabe

im ersten Beispiel sind *Nichtterminalsymbole*. Wie sie heißen, ist genauso belanglos, wie die frei gewählten Namen eines SML-Programms. Genauso wie die Fakultätsfunktion in einem Programm `fak1` oder `@&*` heißen mag, können die Nichtterminalsymbole anders heißen. (Wie in Programmen ist bei einer Umbenennung von Nichtterminalsymbolen in EBNF-Regeln wichtig, dass sie überall stattfindet und auseinander hält, was verschieden ist.)

Die Symbole, die zwischen zwei ” vorkommen, heißen *Terminalsymbole*.

Aus den folgenden EBNF-Regeln

Startsymbol: Binaerzahl
Binaerzahl ::= Binaerziffer Binaerziffer * .
Binaerziffer ::= ”0” | ”1”.

werden unendlich viele (Objekt-)Symbolen (oder Worte) einer Objektsprache unter anderem die folgenden hergeleitet:

0 1 01 10 001 010 100 101 . . .

Eine Menge von EBNF-Regeln, die paarweise unterschiedliche linke Teile haben, zusammen mit der Angabe eines Startsymbols wird EBNF-Grammatik genannt. (Die Angabe des Startsymbols darf nicht vergessen werden!)

Der folgende Algorithmus beschreibt die Herleitung eines (Objekt-)Symbols aus einer EBNF-Grammatik:

Herleitungsalgorithmus für EBNF-Grammatiken:

1. Wähle eine Alternative im rechten Teil der Regel, dessen linker Teil das Startsymbol ist.
2. So lange ein Ausdruck der Gestalt (S) mit S Terminal- oder Nichterminal-Symbol vorkommt, wähle willkürlich einen solchen Ausdruck (S) und ersetze ihn durch S.
3. So lange ein Nichterminalsymbol oder ein Ausdruck der Gestalt expr *, expr + oder [expr] vorkommt, wähle (willkürlich) einen der folgenden Schritte 3.1, 3.2, 3.3 oder 3.4 und fahre mit diesem Schritt fort:
   - 3.1 Fall Nichterminalsymbol:
     - 3.1.1 Wähle (willkürlich) ein Nichtterminalsymbol NT.
     - 3.1.2 Wähle (willkürlich) eine Alternative A in der rechten Seite der EBNF-Regel mit linker Seite NT.
     - 3.1.3 Ersetze das Nichtterminalsymbol NT durch die Alternative A.
   - 3.2 Fall Ausdruck A der Gestalt expr * :
     Wähle (willkürlich) einen der folgenden Schritte 3.2.1 oder 3.2.2:
     - 3.2.1 Streiche A = expr * .
     - 3.2.2 Ersetze A = expr * durch expr expr * .
   - 3.3 Fall Ausdruck A der Gestalt expr + :
     Wähle (willkürlich) einen der folgenden Schritte 3.3.1 oder 3.3.2:
     - 3.3.1 Ersetze A = expr + durch expr .
     - 3.3.2 Ersetze A = expr + durch expr expr + .
   - 3.4 Fall Ausdruck A der Gestalt [expr]: Wähle (willkürlich) einen der folgenden Schritte 3.4.1 oder 3.4.2:
     - 3.4.1 Streiche A = [expr] .
     - 3.4.2 Ersetze A = [expr] durch expr .
4. Fahre bei Schritt 2 fort.
5. Liefere das Ergebnis.

Beispiel einer Anwendung des Herleitungsalgorithmus:
EBNF-Grammatik:

Startsymbol:	Binaerzahl
Binaerzahl	::= Binaerziffer Binaerziffer * .
Binaerziffer	::= ”0” | ”1”.

Im Folgenden wird immer der am weitesten rechts stehende Ausdruck entsprechend des Algorithmus bearbeitet:

Binaerzahl				
Binaerziffer	Binaerziffer *			(Schritt 1)
Binaerziffer	Binaerziffer	Binaerziffer *		(Schritt 3.2.2)
Binaerziffer	Binaerziffer	Binaerziffer	Binaerziffer *	(Schritt 3.2.2)
Binaerziffer	Binaerziffer	Binaerziffer		(Schritt 3.2.1)
Binaerziffer	Binaerziffer	1		(Schritt 3.1.2)
Binaerziffer	0	1		(Schritt 3.1.2)
0	0	1		(Schritt 3.1.2)

Ein anderer Ablauf des Herleitungsalgorithmus, der ebenfalls die Binärzahl 001 liefert, ist wie folgt:

Binaerzahl				
Binaerziffer	Binaerziffer *			(Schritt 1)
0	Binaerziffer *			(Schritt 3.1.2)
0	Binaerziffer	Binaerziffer *		(Schritt 3.2.2)
0	0	Binaerziffer *		(Schritt 3.1.2)
0	0	Binaerziffer	Binaerziffer *	(Schritt 3.2.2)
0	0	Binaerziffer		(Schritt 3.2.1)
0	0	1		(Schritt 3.1.2)

Der Herleitungsalgorithmus liefert den Grund für die Bezeichnungen Terminal- und Nichtterminalsymbole: Während eines Ablaufs des Herleitungsalgorithmus bleiben die Terminalsymbole erhalten und werden die Nichtterminalsymbole nach und nach ersetzt.

Im Schritt 2 können mehrere Ausdrücke (S) zur Wahl stehen. Der Herleitungsalgorithmus legt nicht fest, welcher gewählt werden soll. Man sagt, dass der Herleitungsalgorithmus *nichtdeterministisch* ist.

Man kann sich leicht davon überzeugen, dass die Wahl im Schritt 2 die Reihenfolge der Ersetzungen beeinflusst, jedoch nicht das am Ende des Ausführung des Algorithmus hergeleitete (Objekt-)Symbol.

Wie der Schritt 2 enthält der Schritt 3.4 eine nichtdeterministische Wahl, nämlich die Wahl einer Alternative. Im Gegensatz zur Wahl im Schritt 2 beeinflußt die Wahl im Schritt 3.4 das hergeleitete (Objekt-)Symbol.

Um beide Arten von Nichtdeterminismus zu unterscheiden, spricht man von *don't care*-Nichtdeterminismus (Schritt 2) und *don't know*-Nichtdeterminismus (Schritt 3.4).

13.1.5 Kontextfreie Grammatiken zur formalen Beschreibung der Syntax von Programmen

Die `local`–Ausdrücke von SML (siehe Abschnitt 4.2.2) haben die folgende Gestalt:

```
local
   "Deklarationen"
in
   "Deklarationen"
end
```

wobei `"Deklarationen"` jeweils eine oder mehrere SML-Deklarationen bezeichnet. Man beachte, dass `local`–Ausdrücke eine Art dreielementige Klammerung darstellen: `local` ist wie eine öffnende Klammer, `in` ist ein Zwischenteil und `end` ist eine schließende Klammer.

Die Gestalt der `local`–Ausdrücke kann wie folgt mit einer EBNF-Grammatik spezifiziert werden:

```
Localausdruck ::= "local"
                  ( Deklaration [ ";" ] ) +
                  "in"
                  ( Deklaration [ ";" ] ) +
                  "end".
```

(Der Einfachheit halber wird in dieser EBNF-Regel die Möglichkeit der Reihung von SML-Deklarationen mit `and` (siehe Abschnitt 4.2.8) zur nichtsequenziellen Auswertung nicht berücksichtigt.)

Weitere EBNF-Regeln sind zur Spezifikation des Nichtterminalsymbols Deklaration nötig. Sie werden hier postuliert, jedoch nicht explizit angegeben.

Im obigen EBNF-Regel kommen +,), (,] und [vor. Wie in Abschnitt 13.1.4 bereits erwähnt, sind sie Metasymbole der Sprache der EBNF-Regeln. Die Klammern) und (dienen dazu, den Geltungsbereich des Operators + anzugeben.

Die folgende EBNF-Grammatik definiert arithmetische Ausdrücke über Binärzahlen:

```
Startsymbol:        ArithmAusd
ArithmAusd    ::=   [ "+" | "−" ] Term  |  Term ( "+" | "−" ) Term .
Term          ::=   Faktor | Faktor ( "*" | "/" ) Faktor .
Faktor        ::=   Binaerzahl  |  "(" ArithmAusd ")" .
Binaerzahl    ::=   Binaerziffer Binaerziffer * .
Binaerziffer  ::=   "0" | "1" .
```

Die folgenden Ausdrücke werden mit der vorangehenden EBNF-Grammatik hergeleitet:

```
10
+ 01
(10 * 111)
- (10 * 111)
+ ( + 01 + - (10 * 111) )
```

Es fällt auf, dass die vorangehende EBNF-Grammatik EBNF-Regeln mit wesentlich komplizierteren rechten Seiten aufweist als die EBNF-Grammatik zur Definition der Syntax der alphanumerischen Namen von SML (siehe Abschnitt 13.1.4).

Wenn jede Alternative einer rechten Seite jeder EBNF-Regel einer EBNF-Grammatik G

- ein Terminalsymbol oder
- ein Terminalsymbol gefolgt von einem Nichtterminalsymbol gefolgt von *

ist, so heißt die EBNF-Grammatik *regulär*. So ist die EBNF-Grammatik zur Definition der Syntax der alphanumerischen Namen von SML eine reguläre Grammatik. Die vorangehende EBNF-Grammatik zur Definition der Syntax von arithmetische Ausdrücken ist aber nicht regulär.

Reguläre EBNF-Grammatiken ermöglichen die Spezifikation von Bezeichnern wie die alphanumerischen Namen von SML. Sie ermöglichen aber keine Form der Klammerung, wie sie in `local`–Ausdrücke oder in den vorangehenden arithmetischen Ausdrücke vorkommt.

EBNF-Grammatiken, seien sie regulär oder nicht, werden *kontextfreie* Grammatiken genannt. Die Bezeichnung kontextfrei wurde vom Linguist Noam Chomsky in den 50er Jahren eingeführt. Sie geht darauf zurück, dass im Algorithmus zur Herleitung eines (Objekt-)Symbols aus einer EBNF-Grammatik (siehe Abschnitt 13.1.3) ein Nichterminalsymbol ohne Berücksichtigung des Kontextes, in dem es vorkommt, ersetzt wird.

Kontextfreie Grammatiken sind zu einfach, um viele Aspekte von natürlichen Sprache zu formalisieren. Wenn z.B. aus der folgenden kontextfreien Grammatik mit Startsymbol Satz

Satz	::=	Subjekt Verb .
Subjekt	::=	"Sie" | "Er" .
Verb	::=	"sieht" | "liest" .

korrekte deutsche Sätze wie etwa „Sie sieht“ und „Er liest“ hergeleitet werden, verlangt im folgenden Beispiel die Übereinstimmung in Genus und Numerus vom Subjekt und Verb die Berücksichtigung des Kontexts:

Satz	::=	Subjekt Verb .
Subjekt	::=	"Er" | "Wir" .
Verb	::=	"sehen" | "liest" .

damit inkorrekte Sätze wie „Er sehen“ und „Wir liest“ nicht hergeleitet werden. Die Berücksichtigung eines Kontexts ist mit kontextfreien Grammatiken und dem Herleitungsalgorithmus aus Abschnitt 13.1.3 nicht möglich.

Obwohl die Syntax von Programmiersprachen mit kontextfreien Grammatiken beschrieben werden, verlangt die Syntax der meisten Programmiersprachen die Berücksichtigung von Kontexten. In SML setzt z.B. die Verwendung eines Namens oder einer Ausnahme voraus, dass dieser Name oder diese Ausnahme deklariert wurde, wie die folgenden Beispiele belegen:

```
- 2 + drei;
Error: unbound variable or constructor: drei

- raise ausnahme;
Error: unbound variable or constructor: ausnahme
```

Um die Kontextabhängigkeiten einer Programmiersprache zu spezifizieren, wird eine kontextfreie Grammatik für diese Programmiersprache mit Zusatzbedingungen ergänzt. Auch diese Zusatzbedingungen werden statisch überprüft.

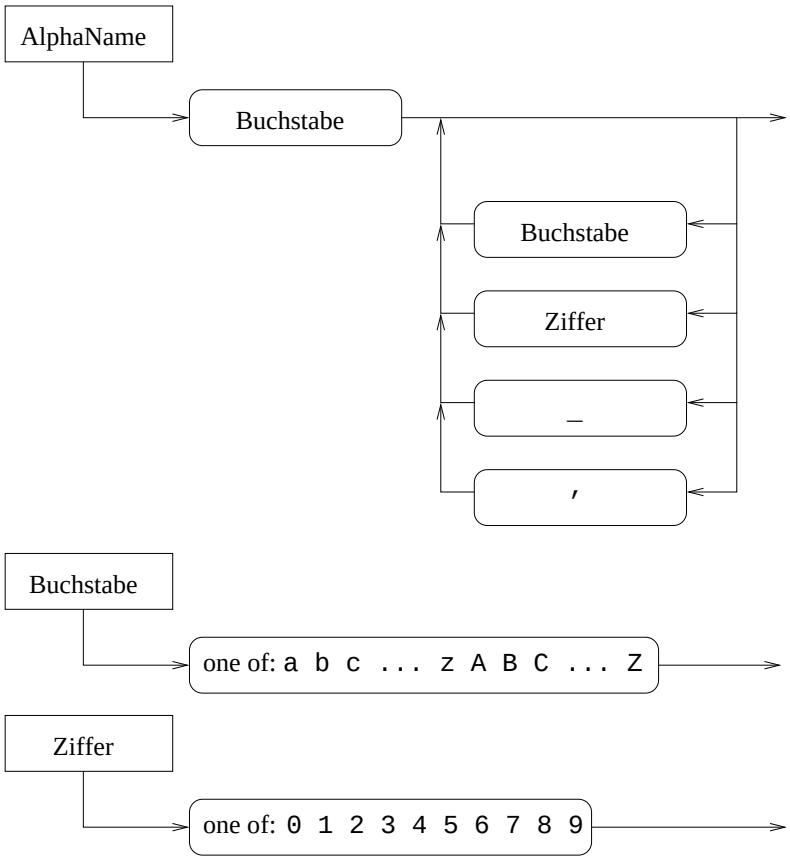
Die Grammatiken, die in der Informatik verwendet werden, gehen auf die Linguistikforschung zurück.[1]

[1] Das Buch *Steven Pinker: The Language Instinct. Penguin Books, 1994* gibt einen fazinierenden, leicht verständlichen Einblick in diese Forschung, die der theoretischen Informatik nahe steht.
Lehrveranstaltungen der Computerlinguistik (siehe das Centrum für Informations- und Sprachverarbeitung, CIS, der LMU) führen in Informatikmethoden der Verarbeitung natürlichen Sprachen wie etwa automatische Übersetzung, automatische Briefbeantwortung und Suchmaschinen ein.

13.1.6 Syntaxdiagramme

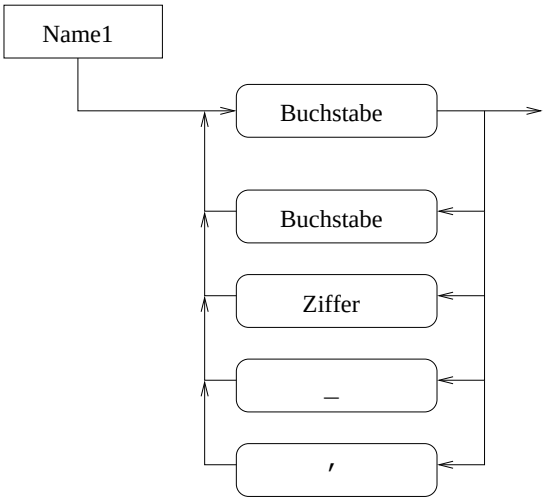
Anstelle von EBNF-Regel werden auch sogenannte Syntaxdiagramme verwendet. Die folgenden Diagramme geben z.B. die Definition der Syntax von alphanumerischen Namen von SML wieder:

> Die alphabetischen Namen von SML fangen mit einem (kleinen oder großen) Buchstaben an, der gefolgt wird von endlich vielen (auch null) Buchstaben (a ... z A ... Z), Ziffern (0 1 2 ... 9), Underscore (_) oder Hochkommata (single quote: ').



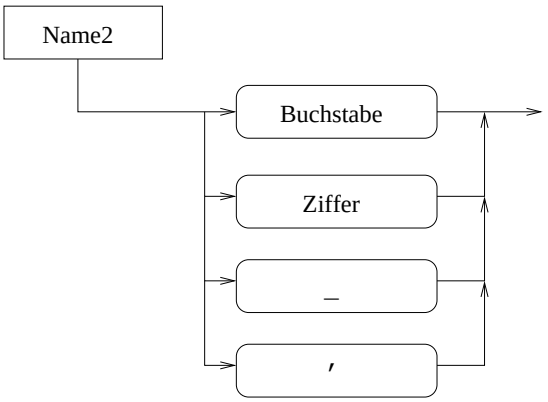
Man vergleiche diese Diagramme mit der EBNF-Grammatik zur Definition der Syntax von alphanumerischen Namen von SML in Abschnitt 13.1.3.

Zum besseren Verständnis von Syntaxdiagrammen vergleichen wir das vorangehende Syntaxdiagramm zur Definition der alphanumerischen Namen in SML mit den folgenden Syntaxdiagrammen (die sich auf die vorangehenden Syntaxdiagramme für Buchstabe und Ziffer beziehen):



Man überzeuge sich, dass Namen, die durch diese Diagramm definiert sind, im Gegensatz zur den alphanumerischen Namen von SML immer eine ungerade Anzahl von Zeichen enthalten.

Das folgende Diagramm definiert Namen, die aus einem einzigen Buchstaben oder aus einer einzigen Ziffer bestehen oder die gleich _ oder ' sind:



Syntaxdiagramme können für den menschlichen Leser übersichtlicher als EBNF-Regeln sein. Deswegen kommen sie in einigen Einführungen in Programmiersprachen vor.[2]

13.2 Formale Beschreibungen der Semantik einer Programmiersprache: Ziele und Ansätze

Eine formale Beschreibung der Semantik einer Programmiersprache wird dazu verwendet, um präzis festzulegen,

- was Programme in dieser Programmiersprache berechnen oder
- wie Programme in dieser Programmiersprache ausgeführt werden sollen.

[2]siehe z.B. die Seiten 457–467 in *Lawrence Paulson: ML for the Working Programmer. MIT Press, 2nd Edition, 1996.*

Der Unterschied zwischen beide Aspekte muss unterstrichen werden. Es kann z.B. sinnvoll sein zu untersuchen, ob ein Programm die Fibonacci-Zahlen (siehe Abschnitt 4.3.5) berechnet, ohne sich dabei dafür zu interessieren, ob das Programm einen iterativen oder baumrekursiven Prozess (siehe Abschnitt 4.3.5) implementiert.

Ein Ansatz zur Festlegung, was Programme in einer gegebenen Programmiersprache S berechnen, ist der Ansatz der *denotationellen Semantik*.

Wird der Rechenvorgang während der Ausführung eines Programms in einer gegebenen Programmiersprache beschrieben, so spricht man von einer *operationalen Semantik* dieser Programmiersprache. Operationale Semantiken von Programmiersprachen können in unterschiedlichen Formalismen definiert werden. Ein Ansatz dazu beruht auf sogenannten *Abstrakten Zustandsmaschinen* (Abstract State Machines, ASM). Dabei werden Abstraktionsebenen unterschiedlicher Grobkörnigkeit ausgewählt und mathematische Funktionen spezifiziert, die den Auswertern aus Kapitel 10 sehr ähnlich sind.[3]

Zur Beschreibung sowohl des „was“ wie des „wie“ einer Berechnung mit imperativen Programmen werden sogenannten *axiomatische Semantiken* bevorzugt. Dabei handelt es sich um logische Kalküle.

Im nächsten Abschnitt wird in die grundlegenden Konzepte der *denotationellen Semantik* eingeführt. Dieser Ansatz eignet sich zur Beschreibung der Semantik einer rein funktionalen Programmiersprache gut, weil er lediglich beschreibt, was Programme berechnen, jedoch nicht wie berechnet wird. Wie rein funktionale Programme ausgewertet werden, lässt sich mit einem ziemlich einfachen Auswertungsalgorithmus beschreiben (siehe Kapitel 10). So liefert eine denotationelle Semantik für eine rein funktionale Programmiersprache eine nützliche, abstrakte Ergänzung des Auswertungsalgorithmus.

13.3 Einführung in die denotationelle Semantik funktionaler Programmiersprachen

In diesem Abschnitt wird erläutert, wie eine denotationelle Semantik für eine rein funktionale Programmiersprache definiert wird. Imperative Sprachkonstrukte (siehe Kapitel 12) werden im Rahmen dieser kurzen Einführung nicht behandelt.

13.3.1 Mathematische Funktionen zur Repräsentation von Programmfunktionen

Eine denotationelle Semantik für eine Programmiersprache S hat das Ziel, eine Antwort auf die Frage zu liefern, welche mathematische Funktion eine Funktionsdeklaration in der Programmiersprache S entspricht — man sagt, welche mathematische Funktion die Denotation („Inhalt“, Bedeutung) der Funktionsdeklaration ist.

Wir wollen hier zunächst anhand von einfachen Beispiele davon überzeugen, dass diese Frage im Allgemeinen keineswegs trivial zu beantworten ist.

[3] Zum Ansatz der abstrakten Zustandsmaschinen siehe:
R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine: Definition, Verifikation, Validation. Springer Verlag, 2001,
E. Börger: High Level System Design and Analysis Using ASMs. in: Lecture Notes in Computer Science (LNCS) 1012, Springer Verlag, 1999,
ASM web site: http://www.eecs.umich.edu/gasm/.

Betrachten wir wieder einmal die Deklaration von `fak1` aus Abschnitt 13.1.1:

```
- fun fak1 x = if x > 0
               then x * (fak1 x) - 1
               else 1;
val fak1 = fn : int -> int
```

Es ist noch leicht zu erkennen, dass `fak1` einer (mathematischen) Funktion entspricht, die jede negative ganze Zahl auf 1 abbildet. Es liegt nahe, als Denotation von `fak1` eine partielle (mathematische) Funktion zu berücksichtigen, die auf den echt positiven ganzen Zahlen nicht definiert ist.
Auch die folgende SML-Funktion `p2` stellt kein großes Verständnisproblem dar:

```
- fun p2(x, y) = y;
val p2 = fn : 'a * 'b -> 'b
```

`p2` implementiert offenbar die zweite Projektion für Paare, d.h. die folgende (mathematische) Funktion:

$$\begin{array}{rccc} proj_2: & A \times B & \rightarrow & B \\ & (x,y) & \mapsto & y \end{array}$$

Welcher mathematischen Funktion entspricht aber das folgende Programm?

```
- val p2fak1 = fn (x, y) => p2(fak1 x, y);
```

Beruht die Programmiersprache wie SML auf der Auswertung in applikativer Reihenfolge, so implementiert `p2fak1` die folgende partielle mathematische Funktion:

$$f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$
$$f(x,y) = \begin{cases} y & \text{falls } x \leq 0 \\ \textit{undefiniert} & \text{sonst} \end{cases}$$

Für eine Programmiersprache, die auf einer Auswertung in normaler Reihenfolge oder auf einer verzögerten Auswertung beruht, implementiert aber `p2fak1` die mathematische Funktion $proj_2$.
Der Ansatz „denotationelle Semantik“ ermöglicht, Deklarationen wie den obigen eine Denotation für jede Auswertungsart zu geben.

13.3.2 Programmfunktionen versus mathematische Funktionen

Untersucht man die denotationelle Semantik einer Programmiersprache, so müssen die folgenden Begriffe auseinander gehalten werden:

- Funktionsdeklaration
- Programmfunktion
- mathematische Funktion

Die Funktionsdeklaration ist der Ausdruck (aus einem Programm), dem eine Bedeutung (Denotation) gegeben werden soll.

Die Programmfunktion ist unser intuitives, oft unpräzises, manchmal fehlerhaftes Verständnis der Funktionsdeklaration — also die Intuition einer mathematischen Funktion. Wäre diese Intuition schon eine formale Definition, so wäre sie die Denotation!

Eine mathematische Funktion hingegen ist ein formal definiertes Objekt.

Das Ziel einer denotationellen Semantik einer funktionalen Programmiersprache S ist die Feststellung einer mathematischen Funktion für jede Funktionsdeklaration in der Programmiersprache S.

Aus der Sicht eines Programmierers kann das Ziel einer denotationellen Semantik die Feststellung einer mathematischen Funktion für jede Programmfunktion sein. Diese Sicht ist etwas unpräzis, weil sie die Funktionsdeklaration (in einer bestimmten Programmiersprache) nicht explizit erwähnt. Sie hat auch den Nachteil, Bezug auf einen intuitiven Funktionsbegriff, nämlich die Programmfunktion, zu nehmen.

13.3.3 Werte von (mathematischen) Funktionen zur Repräsentation von Programmausdrücken ohne (Programm-) Werte

Betrachten wir nochmals die Deklaration von `fak1` aus Abschnitt 13.1.1. Ist `n` eine echt positive ganze Zahl, so terminiert die Auswertung von `fak1 n` nicht. Die (intuitive) Programmfunktion f, die der Funktionsdeklaration `fak1` entspricht, ist also über die echt positiven ganzen Zahlen nicht definiert. Anders ausgedrückt: $f(\mathtt{n})$ hat keinen Wert, wenn $\mathtt{n} \in \mathbb{Z}$ und $\mathtt{n} > 0$.

Programmausdrücke ohne Werte können in zwei Weise denotiert — d.h., mit einer Bedeutung versehen — werden:

1. Das mathematische Gegenstück eines Programmausdrucks ohne Wert ist undefiniert.
2. Das mathematische Gegenstück eines Programmausdrucks ohne Wert ist ein Sonderwert $\perp$ (üblicherweise geschrieben als ein auf dem Kopf stehendes T, gesprochen *bottom*).

Der erste Ansatz hat den Nachteil, Berechnungen im mathematischen Modell zu erschweren. Mit dem zweiten Ansatz kann man die Undefiniertheit eines Programmausdrucks feststellen, wenn der Programmausdruck den Sonderwert als Denotation erhält. Dieser zweiter Ansatz wird deswegen gewählt.

Das Beispiel der Deklaration von `p2fak1` in Abschnitt 13.3.1 zeigt diesen Vorteil:

> Sei $\perp$ die Denotation von `fak1 4` ($\perp$ ist der Sonderwert, der „undefiniert“ bedeutet).
> Ist $proj_2$ die Denotation von `p2`, so ist $proj_2(\perp, 4)$ die Denotation von `p2fak1 4`.
> Je nach dem, ob die betrachtete Programmiersprache die Auswertung in applikativer Reihenfolge oder die verzögerte Auswertung verwendet, definiert man
> $proj_2(\perp, 4) = \perp$ (Auswertung in applikativer Reihenfolge)
> oder
> $proj_2(\perp, 4) = 4$ (verzögerte Auswertung)

13.3.4 Syntaktische und semantische (Wert-)Domäne, Semantikfunktionen

Jedem Monotyp (siehe Abschnitt 5.4.3) der Programmiersprache wird eine Menge, *semantische (Wert-)Domäne* oder kurz *Domäne* genannt, als Denotation zugewiesen, die ein bottom-Element als Sonderelement zur Repräsentation von undefinierten Programmwerten enthält (siehe Abschnitt 13.3.2). Falls nötig, werden die bottom-Elemente unterschiedlicher semantischer Domänen dadurch unterschieden, dass das bottom-Element einer semantischen Domäne D als $\perp_D$ bezeichnet wird.

Die Denotation des Typs `int` ist z.B. $\mathbb{Z} \cup \{\perp\}$.

Die Denotation des Typs `real` ist z.B. $\mathbb{G} \cup \{\perp\}$, wobei $\mathbb{G}$ die Menge der Gleitkommazahlen ist (eine echte Teilmenge der rationalen Zahlen: $\mathbb{G} \subset \mathbb{Q}$).

Die Denotation des Typs `bool` kann z.B. die Menge $\{t, f, \perp\}$, die Menge $\{1, 0, \perp\}$ oder sogar die Menge $\{true, false, \perp\}$ sein. Wird die letzte dieser drei Mengen ausgewählt, so soll zwischen z.B. `true` als Ausdruck der Programmiersprache und dessen Denotation, ebenfalls *true* notiert, unterschieden werden.

In Abschnitt 13.3.5 werden wir sehen, dass über die Denotation eines Typs eine Ordnung mit bestimmten Eigenschaften benötigt wird.

Die Programmausdrücke, die denselben Typ besitzen, bilden eine sogenannte *syntaktische (Wert-)Domäne*.

Die Abbildung (d.h. mathematische Funktion), die die Elemente einer syntaktischen Domäne, d.h. die Ausdrücke eines bestimmten Typs `t`, auf ihre Denotation abbildet, d.h. auf Elemente der semantischen (Wert-)Domäne von `t`, heißt eine *Semantikfunktion*.

Die Definition einer denotationellen Semantik für eine funktionale Programmiersprache besteht in der Festlegung von syntaktischen und semantischen Domänen sowie von Semantikfunktionen.

13.3.5 Strikte und nicht-strikte (mathematische) Funktionen

Beruht eine funktionale Programmiersprache auf einer Auswertung in applikativer Reihenfolge (siehe Abschnitt 3.2), so terminiert die Auswertung der Funktionsanwendung

```
funk(arg1, ..., argi, ..., argn)
```

nicht, wenn die Auswertung des `i`-ten Arguments `argi` nicht terminiert.

Sind die Denotationen der Ausdrücke `funk`, `arg1`, ..., `argi`, ..., `argn` die (mathematischen) Objekte f (eine n-stellige Funktion), $d_1, \ldots, d_i, \ldots d_n$, so muss also gelten (siehe Abschnitt 13.3.2):

$$d_i = \perp :$$
$$f(d_1, \ldots, \perp, \ldots, d_n) = \perp$$

Beruht aber die Programmiersprache auf der verzögerten Auswertung (siehe Abschnitt 3.2), so kann es sein, dass

$$f(d_1, \ldots, \perp, \ldots, d_n) \neq \perp$$

Die folgende Definition führt Begriffe ein, die die vorangehende Unterscheidung erleichtern.

Definition (strikte und nicht-strikte (mathematische) Funktionen)
Seien $D_1, \ldots, D_n$ und D Mengen, die $\perp$ als Element haben, f eine n-stellige (mathematische) Funktion von $D_1 \times \ldots \times D_n$ in D und $i \in \{1, \ldots, n\}$.

- f heißt *strikt im i-ten Argument*, falls für alle $d_1 \in D_1, \ldots, d_{i-1} \in D_{i-1}, d_{i+1} \in D_{i+1}, \ldots, d_n \in D_n$ gilt:

$$f(d_1, \ldots, d_{i-1}, \perp, d_{i+1}, \ldots, d_n) = \perp$$

- f heißt *strikt*, wenn f in allen seinen Argumenten strikt ist.
- f heißt *nicht-strikt*, wenn f in mindestens einem seiner Argumente nicht strikt ist.

Die Denotation einer Funktionsdeklaration in einer Programmiersprache, die auf der Auswertung in applikativer Reihenfolge beruht, muss also eine strikte Funktion sein.

Die Denotation einer Funktionsdeklaration in einer Programmiersprache, die auf der verzögerten Auswertung beruht, muss keine strikte Funktion sein. Sie kann aber eine strikte Funktion sein. So ist z.B. die Denotation des Ausdrucks

```
fun id x = x;
```

in einer Programmiersprache mit verzögerter Auswertung die (mathematische) Funktion:

$$\begin{array}{lccc} i: & A & \to & A \\ & x & \mapsto & x \end{array}$$

die strikt ist (weil $i(\perp) = \perp$).

Nicht selten werden die Begriffe Denotation und Programmfunktion im informellen Sprachgebrauch verwechselt. Es wird z.B. von strikten bzw. nicht-strikten Programmfunktionen gesprochen. Die Verwendung der Begriffe „strikte“ und „nicht-strikte Funktionen“ ist aber nur dann richtig, wenn es sich um (mathematische) Funktionen handelt, diese Funktionen zudem Ursprungs- und Bildmengen haben, in denen das Sonderelement $\perp$ vorkommt.

13.3.6 Approximationsordnung

Betrachten wir die folgende Funktionsdeklaration in SML:

```
fun fak2 x = if x = 0
             then 1
             else x * fak2 (x - 1);
```

Ist `n` eine positive ganze Zahl, so liefert der Aufruf `fak2 n` die Fakultät von `n`. Ist aber `n` eine echtnegative ganze Zahl, so terminiert die Auswertung von `fak2 n` nicht. Die folgende (mathematische) Funktion

$$f0 : \mathbb{Z} \cup \{\bot\} \to \mathbb{Z} \cup \{\bot\}$$

$$n \mapsto \begin{cases} n! & \text{falls } n \geq 0 \\ \bot & \text{falls } n < 0 \\ \bot & \text{falls } n = \bot \end{cases}$$

ist eine mögliche Denotation für `fak2`, weil sie die folgenden Bedingungen erfüllt:

1. $f0$ bildet $\mathbb{Z} \cup \{\bot\}$ in $\mathbb{Z} \cup \{\bot\}$ ab;
2. $f0(\bot) = \bot$;
3. Wenn $f0(n) \neq \bot$, dann $f0(n) = n!$.

Der dritte Fall der Definition von $f0(n)$ ist notwendig, damit die Anwendung von `fak2` auf Programmausdrücke ohne Werte abgedeckt ist, $f0$ eine totale Funktion von $\mathbb{Z} \cup \{\bot\}$ in $\mathbb{Z} \cup \{\bot\}$ ist und $f0$ strikt ist, was die Auswertungsreihenfolge von SML verlangt.
Die folgende (mathematische) Funktion $f24$ ist aber auch eine mögliche Denotation für `fak2`:

$$f24 : \mathbb{Z} \cup \{\bot\} \to \mathbb{Z} \cup \{\bot\}$$

$$n \mapsto \begin{cases} n! & \text{falls } n \geq 24 \\ \bot & \text{falls } n < 24 \\ \bot & \text{falls } n = \bot \end{cases}$$

Zugegebenermaßen entspricht die Grenze von 24 statt 0 nicht unserer Erfahrung als Programmierer. $f24$ erfüllt aber genauso wie $f0$ die Bedingungen für eine Denotation der SML-Funktionsdeklaration von `fak2`:

1. $f24$ bildet $\mathbb{Z} \cup \{\bot\}$ in $\mathbb{Z} \cup \{\bot\}$ ab;
2. $f24(\bot) = \bot$;
3. Wenn $f24(n) \neq \bot$, dann $f24(n) = n!$.

Offenbar gibt es für jede positive ganze Zahl n eine (mathematische) Funktion fn, die als Denotation von `fak2` genauso gut (oder genauso schlecht) wie $f24$ in Frage kommt.
Auch die folgende (mathematische) konstante Funktion f_{inf}, die jedes Element von $\mathbb{Z} \cup \{\bot\}$ auf $\bot$ abbildet, kommt als Denotion von `fak2` genauso gut (oder genauso schlecht) wie alle Funktionen fn in Frage:

$$\begin{array}{lccc} f_{inf} : & \mathbb{Z} \cup \{\bot\} & \to & \mathbb{Z} \cup \{\bot\} \\ & n & \mapsto & \bot \end{array}$$

Wie jede Funktion fn erfüllt f_{inf} die Bedingungen 1., 2. und 3.:

1. f_{inf} bildet $\mathbb{Z} \cup \{\bot\}$ in $\mathbb{Z} \cup \{\bot\}$ ab;
2. $f_{inf}(\bot) = \bot$;
3. Wenn $f_{inf}(n) \neq \bot$, dann $f_{inf}(n) = n!$.

Benötigt wird also eine formale Begründung, warum die Denotation von `fak2` $f0$ und nicht $f24$ oder f_{inf} sein soll. Der Begriff *Approximationsordnung* liefert eine solche Begründung.
Wir erinnern zunächst daran, wie eine Ordnung definiert ist (siehe Abschnitt 8.6.5 und 12.4.1):

> Eine (partielle) Ordnung über einer Menge M ist eine reflexive, transitive und antisymmetrische binäre (d.h. zweistellige) Relation $\leq$ über M.
> Gilt für jedes $m1 \in M$ und jedes $m2 \in M$, $m1 \leq m2$ oder $m2 \leq m1$, so ist $\leq$ eine totale Ordnung über M.

Ist D eine semantische Domäne, also eine Menge, die $\bot$ enthält, so wird die folgende Ordnung $\sqsubseteq$ über D definiert:

Definition

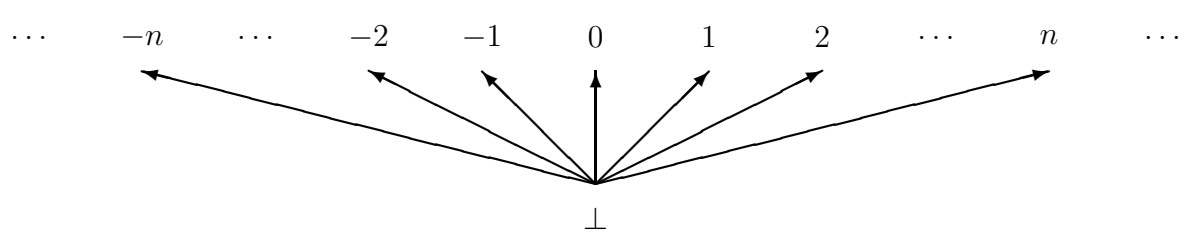
1. $\bot \sqsubseteq d$ für alle $d \in D$, d.h. $\bot$ ist Minimum der Ordnung $\sqsubseteq$;
2. $d \sqsubseteq d$ für alle $d \in D$.

Über $\mathbb{Z} \cup \{\bot\}$, die Denotation des SML-Typs `int`, ist die Approximationsordnung $\sqsubseteq$ wie folgt definiert:

Definition

1. $\bot \sqsubseteq n$ für alle $n \in \mathbb{Z}$;
2. $n \sqsubseteq n$ für alle $n \in \mathbb{Z}$;
3. $\bot \sqsubseteq \bot$.

Man beachte, dass zwischen ganzen Zahlen n und m die Beziehung $n \sqsubseteq m$ nur dann gilt, wenn $n = m$.
Diese Approximationsordnung über $\mathbb{Z} \cup \{\bot\}$wird wie folgt graphisch dargestellt:



Dabei wird $a \sqsubseteq b$ durch eine gerichtete Kante von a nach b ausgedrückt. In dieser graphischen Darstellung werden die Beziehungen $a \sqsubseteq a$ nicht dargestellt (sie gelten jedoch).

Wird $\perp$ als „undefiniert“ ausgelegt, so drückt die Ordnung $\sqsubseteq$ einen *Definiertheitsgrad* aus: $\perp$ ist weniger definiert als jeder andere Wert, jeder Wert ist genau so viel definiert wie er selbst.

Die (mathematische) Funktion $f24$ ist weniger definiert als die (mathematische) Funktion $f0$, weil sie weniger Elemente der Urspungsmenge als $f0$ auf definierte Werte abbildet — anders ausgedrückt, weil sie mehr Elemente der Urspungsmenge auf $\perp$ abbildet.

Die folgende Definition formalisiert diesen Begriff „Definiertheitsgrad“.

Definition (Semantische Domäne)

- Eine semantische (Wert-)Domäne ist eine Menge D mit $\perp \in D$ (oder $\perp_D \in D$), über die eine Ordnung $\sqsubseteq$ (oder $\sqsubseteq_D$) definiert ist, für die gilt:

  1. $\perp \sqsubseteq d$ für alle $d \in D$;
  2. $d \sqsubseteq d$ für alle $d \in D$.

  (Eventuell gelten keine weitere Ordnungsbeziehungen in D als diejenigen, die von der Definition einer semantischen Domäne verlangt werden (siehe auch Abschnitt 13.3.7: „flache Domäne“)

- Sind $D_1, \ldots, D_n$ semantische Domänen, so ist eine Ordnung $\sqsubseteq$ über das kartesische Produkt $D_1 \times \ldots \times D_n$ wie folgt definiert:

  $$(d_1, \ldots, d_n) \sqsubseteq (e_1, \ldots, e_n)$$
  $$\Longleftrightarrow$$
  $$d_1 \sqsubseteq e_1 \text{ und } \ldots \text{und } d_n \sqsubseteq e_n$$

  (Man kann leicht beweisen (Übung!), dass diese Definition tatsächlich eine Ordnung über $D_1 \times \ldots \times D_n$ definiert.)

Definition (Approximationsordnung für Vektoren und Funktionen)
Sind D_1, …, D_n und D semantische Domänen so ist eine Ordnung $\sqsubseteq$ über der Menge der (totalen) Funktionen von $D_1 \times \ldots \times D_n$ in D wie folgt definiert:

$$f \sqsubseteq g$$
$$\Longleftrightarrow$$
für alle $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ gilt: $f(d_1, \ldots, d_n) \sqsubseteq g(d_1, \ldots, d_n)$

Diese Ordnung über der Menge der (totalen) Funktionen von $D_1 \times \ldots \times D_n$ in D heißt Approximationsordnung.
(Man kann leicht beweisen (Übung!), dass diese Definition tatsächlich eine Ordnung über der Menge der (mathematischen) Funktionen von $D_1 \times \ldots \times D_n$ in D definiert.)

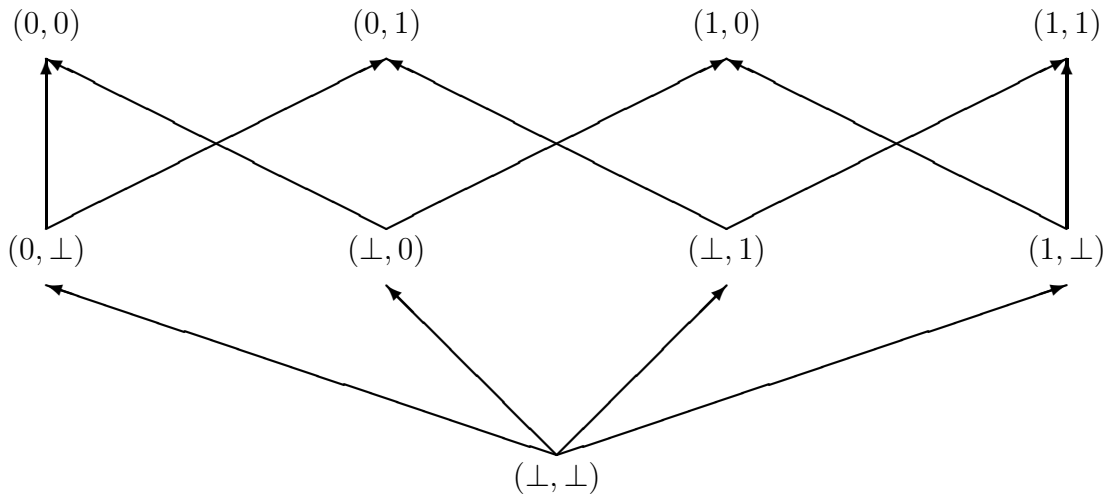
Man beachte, dass die Definitionen zusammen auch den Fall abdecken, wenn die Bildmenge der Funktionen ein Kartesisches Produkt ist.

Man beachte auch, dass im Allgemeinen über (mathematische) Funktionen keine totale Ordnung vorliegt. Betrachten wir die semantische Domäne $D = \{0, 1, \bot\}$ mit der partiellen Ordnung:

$$\begin{array}{ll} \bot \sqsubseteq 0 & 0 \sqsubseteq 0 \\ \bot \sqsubseteq 1 & 1 \sqsubseteq 1 \end{array}$$

Für diese Ordnung sind 0 und 1 unvergleichbar: Es gilt weder $0 \sqsubseteq 1$ noch $1 \sqsubseteq 0$.

Die nach der obigen Definition auf $D \times D$ induzierte Ordnung kann wie folgt graphisch dargestellt werden, wobei eine gerichtete Kante von a nach b wieder die Beziehung $a \sqsubseteq b$ bedeutet und für Beziehungen $a \sqsubseteq a$ keine Kanten dargestellt werden:



Das Paar $(\bot, \bot)$ ist das bottom-Element von $D \times D$, d.h. $\bot_{D \times D} = (\bot_D, \bot_D)$.

Die oben eingeführte Ordnung über $D \times D$ ist nicht total: die Vektoren $(0, \bot)$ und $(1, 1)$ sind z.B. für diese Ordnung nicht vergleichbar.

13.3.7 Denotation einer Funktionsdeklaration

Die Approximationsordnung ermöglicht, zwischen $f0$, $f24$, fn und f_{inf} als Denotationskandidaten für die SML-Funktionsdeklaration `fak2` zu unterscheiden: Bezüglich der Approximationsordnung ist $f0$ maximal. In der Tat gilt:

$f_{inf} \sqsubseteq fn \sqsubseteq fm \sqsubseteq f0$ für alle $n, m \in \mathbb{N}$ mit $n \geq m$.

Eine derjenigen unter den möglichen Denotationen einer Funktionsdeklaration, die bezüglich der Approximationsordnung maximal ist, soll also als Denotation ausgewählt werden. Dies bedeutet, dass als Denotation ein Denotationskandidat auszuwählen ist, der am wenigsten undefiniert ist. Dies ist sicherlich ein vernünftiges Kriterium: Die formale Bedeutung einer Funktionsdeklaration soll eine (mathematische) Funktion sein, die so weit definiert ist, wie die Funktionsdeklaration es überhaupt ermöglicht.

So vernünftig diese Definition auch sein mag, es stellen sich doch die folgenden Fragen:

- Existiert immer eine maximale (mathematische) Funktion unter den Denotationskandidaten einer Funktionsdeklaration?

- Falls es immer mindestens eine maximale (mathematische) Funktion unter den Denotationskandidaten gibt, ist sie eindeutig (d.h. gibt es nur eine einzige maximale Funktion)?

Beide Fragen können mit ja beantwortet werden, so dass der vorangehende Ansatz zur Denotation einer Funktionsdeklaration einen Sinn ergibt.

Für nichtrekursive Funktionsdeklarationen ist dies nicht schwer zu zeigen. Der Beweis für rekursive Funktionsdeklarationen ist aber keineswegs unmittelbar. Das Prinzip dieses Beweises, das auch für die Programmierpraxis interessant ist, ist am Ende dieses Kapitel in Abschnitt 13.3.10 erläutert.

13.3.8 Semantische Domäne

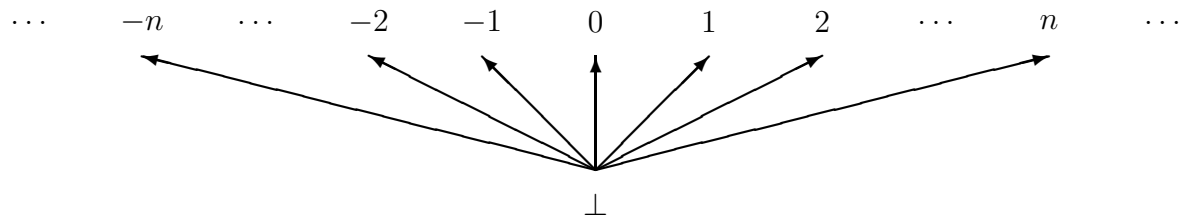
Der vorangehende Abschnitt hat als semantische Domäne für einen Vektor ein kartesisches Produkt von Domänen betrachtet. Je nach dem, ob die Auswertung der Programmiersprache, deren Semantik spezifiziert wird, eine Auswertung in applikativer Reihenfolge oder eine verzögerte Auswertung ist, werden die semantischen Domänen für zusammengesetzte Typen unterschiedlich definiert.

Flache Domäne

Sei $\mathbb{Z} \cup \{\bot\}$ die Domäne, die die Denotation des Typs `int` ist. Die Ordnung $\sqsubseteq$ über $\mathbb{Z} \cup \{\bot\}$ ist wie folgt definiert:

$$\bot \sqsubseteq n \text{ für alle } n \in \mathbb{Z}$$

Graphisch kann diese Ordnung, die nicht mit $\leq$ verwechseln werden darf, wie folgt dargestellt werden:



Eine solche Ordnung, die nur Vergleiche mit $\bot$ bietet, wird *flache Ordnung* genannt.

Die Ordnung über die semantische Domäne, die Denotation eines nichtzusammengesetzten Monotyps (wie etwa `int`, `real`, `bool`) ist, ist immer eine flache Ordnung.

Vollständigkeit der semantischen Domäne

Ist D eine semantische (Wert-)Domäne, so ist eine $\sqsubseteq$–Kette in D eine (abzählbar) unendliche Folge

$$d_0, d_1, d_2, \ldots, d_n, \ldots$$

von Elementen von D, so dass:

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$$

d.h. für alle $i \in \mathbb{N}$ gilt: $d_i \sqsubseteq d_{i+1}$. Eine obere Schranke (*upper bound*) einer $\sqsubseteq$–Kette ist ein Wert s, so dass für jedes $i \in \mathbb{N}$ gilt: $d_i \sqsubseteq s$.

Eine obere Schranke k der Kette heißt kleinste obere Schranke (*least upper bound*), wenn $k \sqsubseteq s$ für jede obere Schranke s der Kette gilt.

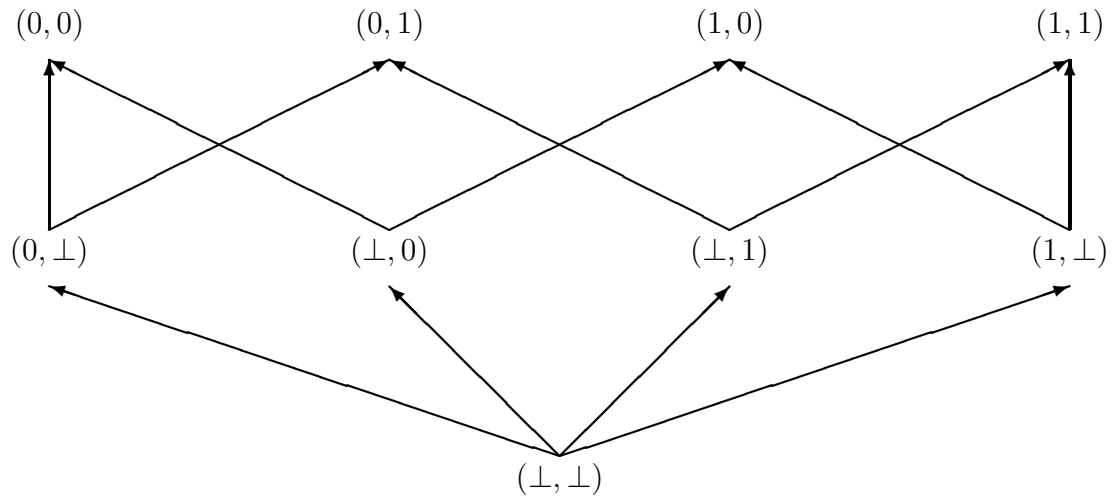
Beispiel: Die $\leq$–Kette 0, 1, 2, ..., n, ... hat keine obere Schranke in $\mathbb{N}$.

In der flachen Domäne $D = \mathbb{Z} \cup \{\bot\}$, die zuvor eingeführt wurde, hat jede Kette in D eine kleinste obere Schranke in D, weil jede Kette nur zwei unterschiedliche Elemente enthält (ab einem gewissen Index n gilt immer $d_n = d_{n+1} = \ldots = d_{n+m} = \ldots$).

Es wird von einer semantischen Domäne D verlangt, dass jede $\sqsubseteq$–Kette, informell *Approximationskette* genannt, in D eine kleinste obere Schranke in D besitzt.

Diese Bedingung wird *Vollständigkeitsbedingung* für semantische Domänen genannt. Die Vollständigkeitsbedingung für semantische Domänen stellt sicher, dass jede *Approximationskette* ein Limes innerhalb der semantischen Domäne hat.

Betrachten wir die Denotation $D = \{0, 1, \bot\}$ des Typs `bool` (vgl. Abschnitt 13.3.6) und die Denotation $D \times D$ des Typs `bool * bool`:



Offenbar hat jede $\sqsubseteq$–Kette in $D \times D$ eine kleinste obere Schranke in D.

Für endliche semantische Domänen stellt die Vollständigkeit keine strenge Bedingung dar. Zur Denotation von Funktionsdeklarationen werden aber unendliche semantische Domänen verwendet. Eine mögliche Denotation für die folgende Deklaration

```
fun pluseins n = n + 1;
```

kann aber z.B. die folgende unendliche semantische Domäne sein:

$$D = \{(n, n+1) | n \in \mathbb{Z}\} \cup \{(\bot, \bot)\}$$

Eine Kette über D kann wie folgt definiert werden:

Für $m \in \mathbb{N}$:

$$f_m : \; D \to D$$
$$n \mapsto \begin{cases} \bot & \text{falls } n \in \mathbb{Z} \text{ und } n \geq m \\ n+1 & \text{falls } n \in \mathbb{Z} \text{ und } n < m \\ \bot & \text{falls } n = \bot \end{cases}$$

Offenbar gilt: $f_m \sqsubseteq f_{m+1}$ für alle $m \in \mathbb{N}$.
Man kann zeigen (Übung!), dass die folgende Funktion die kleinste obere Schranke der Kette $(f_m)_{m \in \mathbb{N}}$ ist:

$$\begin{array}{lccll} f_\infty : & D & \to & D & \\ & n & \mapsto & n+1 & \text{falls } n \in \mathbb{Z} \end{array}$$

Diese Funktion f_∞ ist ein Element der Menge der Funktionen von D in D. (Man kann zeigen, dass diese Funktionsmenge vollständig ist.)

Die Domäneprodukte $\times$ und $\otimes$

Sind $D1$ und $D2$ semantische Domänen, so bezeichnet $D1 \times D2$ die semantische Domäne mit bottom-Element $(\bot, \bot)$ und mit der Approximationordnung, die gemäß Abschnitt 13.3.6 von den Ordnungen von $D1$ und $D2$ induziert wird.
Die Produktdomäne $D1 \times D2$ ist sinnvoll, wenn eine Programmiersprache mit verzögerter Auswertung betrachtet wird. Vektoren der Gestalt $(\bot, d2)$ mit $d2 \neq \bot$ und $(d1, \bot)$ mit $d1 \neq \bot$ ermöglichen, Funktionsdeklarationen wie

```
fun p2 (x : int, y : int) = y;
```

als Denotation eine nicht-strikten Funktion wie

$$\begin{array}{lccl} proj_2 : & \mathbb{Z} \cup \{\bot\} \times \mathbb{Z} \cup \{\bot\} & \to & \mathbb{Z} \cup \{\bot\} \\ & (x, y) & \mapsto & y \end{array}$$

zu geben, für die gilt: $proj_2(\bot, y) = y$.
Wird aber eine Programmiersprache betrachtet, die auf Auswertung in applikativer Reihenfolge beruht, dann werden als mögliche Denotationen nur strikte Funktionen berücksichtigt, so dass Vektoren der Gestalt $(\bot, d2)$ mit $d2 \neq \bot$ und $(d1, \bot)$ mit $d1 \neq \bot$ unerwünscht sind. Sie werden in der folgenden Definition einer Produktdomäne ausgeschlossen:

$$D1 \otimes D2 = \{(d1, d2) | d1 \in D1, d1 \neq \bot, d2 \in D2, d2 \neq \bot\} \cup \{(\bot, \bot)\}$$

Das bottom-Element von $D1 \otimes D2$ ist $(\bot, \bot)$ — d.h. $(\bot_{D1}, \bot_{D2})$ — und die Approximationsordnung von $D1 \otimes D2$ wird wie in Abschnitt 13.3.6 definiert.

Die Domänevereinigungen $\oplus$ und $+$

In der verquickenden (oder verschmelzenden) Vereinigung $D1 \oplus D2$ zweier semantischer Domänen $D1$ und $D2$ werden die bottom-Elemente von $D1$ und $D2$ gleichgesetzt und die

anderen Elemente auseinandergehalten. Haben $D1$ und $D2$ einen Schnitt, der nicht nur $\bot$ enthält, so müssen also die Elemente von $D1$ (oder $D2$) umbenannt werden. Die Ordnung über $D1 \oplus D2$ ist einfach die Vereinigung der Ordnungen über $D1$ und $D2$. Es heißt also, dass außer $\bot$ kein Element von $D1$ mit einem Element von $D2$ in $D1 \oplus D2$ verglichen werden kann.

Im Gegensatz zur verquickenden Vereinigung hält die unterscheidende Vereinigung $D1 + D2$ von zwei semantischen Domänen $D1$ und $D2$ jedes Element von $D1$ einschließlich $\bot_{D1}$ von jedem Element von $D2$ einschließlich $\bot_{D2}$ auseinander. Die Ordnung über $D1 + D2$ ist die Vereinigung der Ordnungen von $D1$ und $D2$. Damit $D1 + D2$ über ein Minimum verfügt, wird $D \cup D2$ ein Element $\bot$ hinzugefügt, das weder in $D1$ noch in $D2$ vorkommt. Für dieses (neue) bottom-Element $\bot$ gilt:

$$\begin{array}{ll} \bot \sqsubseteq d1 & \text{für alle } d1 \in D1 \quad (\text{u.a. } \bot \sqsubseteq \bot_{D1}) \\ \bot \sqsubseteq d2 & \text{für alle } d2 \in D2 \quad (\text{u.a. } \bot \sqsubseteq \bot_{D2}) \end{array}$$

Die Domänevereinigungen werden zur Denotation von Typen mit Varianten verwendet wie etwa:

```
datatype t = k1 of t1 | ... | kn of tn;
```

Sind $D_1, \ldots, D_n$ die Denotationen (d.h. semantischen Domänen) der Typen `t1`, ..., `tn`, so ist bei einer Programmiersprache mit Auswertung in applikativer Reihenfolge die verquickende Vereinigung $D_1 \oplus \ldots \oplus D_n$ die Denotation von `t`. Man kann sich leicht davon überzeugen, dass $\oplus$ assoziativ ist, so dass der Ausdruck $D_1 \oplus \ldots \oplus D_n$ einen Sinn ergibt.

Angehobene Domäne

Ist D eine semantische Domäne, so erhält man die angehobene (*lifted*) semantische Domäne $D_\bot$ dadurch, dass D um ein (neues) $\bot$ Element ergänzt wird, für dass gilt:

$$\begin{array}{ll} \bot \neq d & \text{für alle } d \in D \text{ (u.a. } \bot \neq \bot_D) \\ \bot \sqsubseteq d & \text{für alle } d \in D \text{ (u.a. } \bot \sqsubseteq \bot_D) \end{array}$$

Man beachte, dass für alle Domänen $D1$ und $D2$ gilt:

$$\begin{array}{rcl} D1 + D2 & = & D1_\bot \oplus D2_\bot \\ (D1 \times D2)_\bot & = & D1_\bot \otimes D2_\bot \end{array}$$

Die Domänenanhebung ermöglicht eine andere Denotation eines Typs mit Varianten. Betrachten wir dazu wieder die `datatype`–Deklaration:

```
datatype t = k1 of t1 | ... | kn of tn;
```

Der Wertkonstruktor `k1` hat den Typ `t1 -> t`. Ist D_1 die Denotation des Typs `t1`, so kann die semantische Domäne der Werte des Typs `t` der Gestalt `k1(.)` die Denotation $(\{k1\} \times D_1)_\bot$ erhalten. So ist die Denotation eines Wertes der Gestalt `k1(.)` ein Paar $(k1, d_1)$ mit $d_1 \in D_1$. Diese Denotation ermöglicht, den Wert $(k1, \bot_{D_1})$ vom Wert $\bot$ (als Element der Denotation von `t`) zu unterscheiden.

Anstatt eines Paares $(k1, d_1)$ wird auch $k1(d_1)$ geschrieben.

Die Denotation von `t` ist dann:

$$(\{k1\} \times D_1)_\bot \oplus \ldots \oplus (\{kn\} \times D_n)_\bot$$

Wenn $D_1 = \ldots = D_n$, dann gilt (Übung!):

$$(\{k1\} \times D_1)_\bot \oplus \ldots \oplus (\{kn\} \times D_n)_\bot = (\{k1, \ldots, kn\} \times D)_\bot$$

Semantische (Wert-)Domäne zur Denotation von Polytypen

Die Denotation des Polytyps `'a list` besteht aus den (unendlich vielen) Denotationen der Listentypen, die sich aus der Bindung der Typvariable `'a` an einem Typausdruck ergibt, also den Denotationen der Typen `int list`, `bool list`, `real list`, `int list list`, usw.

Die Verallgemeinerung dieses Ansatzes auf beliebige Polytypen stellt keine prinzipielle Schwierigkeit dar.

Ein polymorpher Ausdruck wie die leere Liste `nil` (oder `[]`) gehört zu jedem Listentyp. Ihre Denotation muss also Element der Denotation jedes Listentyps sein.

Die Denotation eines Polytyps besteht also in einer (unendlichen) Menge von Denotationen, deren Durchschnitt die Menge der polymorphen Ausdrücke des Polytyps ist.

Semantische Domäne zur Denotation von rekursiven Typen

Beruht die Programmiersprache auf die Auswertung in applikativer Reihenfolge, so ist die Denotation der folgenden Deklaration eines rekursiven Typs (siehe Abschnitt 8.4)

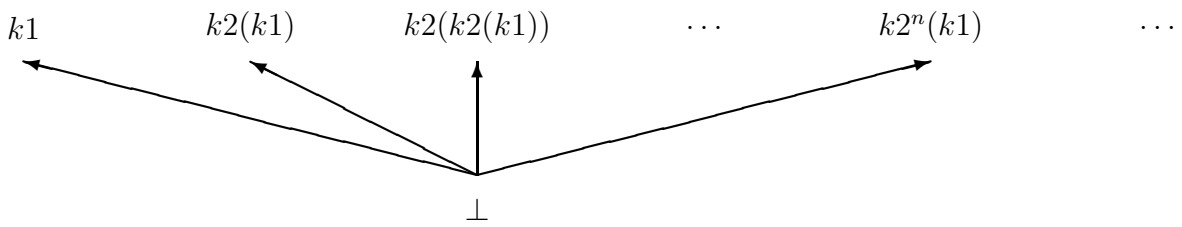
```
datatype t = k1 | k2 of t;
```

eine Menge D, die die folgende rekursive Gleichung erfüllen muss (vgl. auch mit „angehobenen Domänen“):

$$D = \{k1\}_\perp \oplus (\{k2\} \times D)_\perp$$

Diese Gleichung werden dadurch erstellt, dass die semantischen Domänen gemäß den vorangehenden „Anweisungen“ definiert werden.

Eine solche rekursive Gleichung stellt aber keine Definition dar! Es kann jedoch bewiesen werden, dass die folgende flache Domäne die passende Lösung dieser rekursiven Gleichung ist:



Ferner kann bewiesen werden, dass jede rekursive Gleichung, die sich aus der Deklaration eines rekursiven Typs ergibt, eine Lösung hat. So kann also jeder Deklaration eines rekursiven Typs eine semantische Domäne zugewiesen werden.

Für eine Programmiersprache, die auf der verzögerten Auswertung beruht, wird eine andere, kompliziertere Art von (ebenfalls rekursiven) Gleichungen verwendet, die gewährleisten, dass der rekursive Typ `t` die Deklaration von „unendlichen“ (d.h. zyklischen) Werten ermöglicht. Auch diese komplizierteren rekursiven Gleichungen haben immer Lösungen, so dass ebenfalls für Programmiersprachen mit verzögerter Auswertung den Deklarationen von rekursiven Typen semantische Domänen zugewiesen werden können.

Funktionsdomäne, Monotonie und Stetigkeit

Es bietet sich an, als Denotation einer Funktionsdeklaration mit Typ `t1 -> t2` die Menge der (mathematischen) Funktionen von $D1$ in $D2$ zu betrachten, wenn $D1$ die Denotation von `t1` und $D2$ die Denotation von `t2` ist.

Nicht alle (mathematische) Funktionen sind aber als Denotation von Funktionsdeklarationen sinnvoll, weil nicht alle (mathematische) Funktionen berechnet werden können. Die Denotation eines Funktionstyps `t1 -> t2` enthält also nur die Funktionen von der Denotation von `t1` in die Denotation von `t2`, die einige Eigenschaften besitzen, die die Berechenbarkeit der (mathematischen) Funktionen sicherstellen. Im Folgenden zeigen wir, dass zwei Eigenschaften notwendig sind: Die Monotonie und die Stetigkeit.

- *Monotonie:*

  Sei eine (mathematische) Funktion $f : D1_\perp \to D2_\perp$ mit beliebigen Mengen $D1$ und $D2$ gegeben.

  $D1_\perp$ und $D2_\perp$ sind die angehobenen Domänen, die das bottom-Element als Minimum einer (eventuell flachen) Ordnung $\sqsubseteq$ beinhalten.

  Wenn gilt

  1. $f(\perp) = d2 \in D2$ (also $d2 \neq \perp$) und
  2. $f(d1) = \perp$ für ein $d1 \in D1$ (also $d1 \neq \perp$)

  dann eignet sich f nicht als Denotation einer Funktionsdeklaration: Terminiert die Auswertung einer Funktionsanwendung auf einen Ausdruck ohne Wert, was die Gleichung (1) besagt, so muss ebenfalls die Auswertung derselben Funktion auf einen Ausdruck mit Wert terminieren, was der Gleichung (2) widerspricht.

  Die Approximationsordnung ermöglicht eine einfache Charakterisierung der (mathematischen) Funktionen, die als Denotation einer Funktionsdeklaration zulässig sind, nämlich die monotonen (mathematischen) Funktionen.

Definition (monotone (mathematische) Funktionen)
Seien $D1$ und $D2$ zwei semantische Domänen und f eine Funktion von $D1$ in $D2$.
f heißt *monoton*, wenn gilt:
Für alle $x \in D1$ und $y \in D1$, wenn $x \sqsubseteq y$, dann $f(x) \sqsubseteq f(y)$.

Die Funktion f des vorangehenden Beispiels ist nicht monoton, weil $f(d1) = \perp \sqsubseteq d2 = f(\perp)$ trotz $\perp \sqsubseteq d1$.

Die Monotoniebedingung hat weitreichende Folgen. Betrachten wir die folgende Funktionsdeklaration:

```
fun pluseins x = x + 1;
```

Ist f die Denotation von `pluseins`, so muss gelten:

$$f(\perp) \sqsubseteq f(0) \text{ und } f(\perp) \sqsubseteq f(1)$$

weil $\perp \sqsubseteq 0$ und $\perp \sqsubseteq 1$. Da aber in der flachen Domäne $\mathbb{Z} \cup \{\perp\}$, die die Bildmenge von f ist, $\perp$ das einzige Element ist, das ≤ 0 und ≤ 1 ist, muss gelten:

$$f(\perp) = \perp \text{ und } f(0) \neq f(1)$$

Das heißt, dass die Denotation von `pluseins` eine strikte Funktion sein muss — und dies unabhängig davon, auf welcher Auswertungsform die Programmiersprache beruht.

- *Stetigkeit:*

  Eine (mathematische) Funktion f von $D1$ in $D2$ heißt *stetig* genau dann, wenn f den Limeswert (d.h. die kleinste obere Schranke) einer $\sqsubseteq$–Kette in $D1$ $(d_i)_{i\in\mathbb{N}}$ auf den Limeswert (d.h. die kleinste obere Schranke) der $\sqsubseteq$–Kette in $D2$ $(f(d_i))_{i\in\mathbb{N}}$ abbildet.

  Die Stetigkeit einer Funktion bedeutet, dass ihre Werte lokal, d.h. jeder für sich statt alle zusammen, berechnet werden können. Dies entspricht zweifelsohne dem intuitiven Begriff einer Programmfunktion!

  Ist eine Funktion stetig, so ist sie auch monoton: Man betrachtete eine $\sqsubseteq$–Kette in der Ursprungsmenge einer stetigen Funktion f, die höchstens zwei unterschiedliche Elemente enthält, z.B. $x = d_0 \sqsubseteq y = d_1 = d_2 = \ldots$. Offenbar ist y die kleinste obere Schranke der $\sqsubseteq$–Kette. Wenn f stetig ist, dann gilt $f(x) \sqsubseteq f(y)$.

  Sind $D1$ und $D2$ zwei semantische Domänen, so bezeichnet

  $$[D1 \to D2]$$

  die Menge der stetigen Funktionen von $D1$ in $D2$ und

  $$[D1 \to_\perp D2]$$

  die Menge der stetigen und strikten Funktionen von $D1$ in $D2$.

(Semantische) Umgebung

Eine Umgebung bildet eine endliche Menge von Namen (oder Bezeichner, Variablen) auf Werte ab. Die Denotation einer Umgebung, ebenfalls Umgebung oder auch *semantische Umgebung* genannt, ist also keine beliebige (mathematische) Funktion, sondern eine partielle (mathematische) Funktion mit endlichem Bereich.

Es sei daran erinnert, dass der Bereich $Bereich(f)$ einer partiellen Funktion $f : A \to B$ wie folgt definiert ist:

$$Bereich(f) = \{a | \exists b \in B : (a, b) \in f\}$$

Die Menge der partiellen (mathematischen) Funktionen mit endlichem Bereich wird

$$[A \to_{fin} B]$$

bezeichnet.

Bezeichnet Var die Menge der Namen (oder Variablen) der Programmiersprache und Val die verquickende Vereinigung der Denotationen der (unendlich vielen) Typen, so ist die Denotation einer Umgebung eine partielle Funktion aus der Menge

$$[Var \to_{fin} Val]$$

Sind $env1$ und $env2$ zwei (semantische) Umgebungen, so bezeichnet $env1 + env2$ die folgende (semantische) Umgebung:

$$(env1 + env2)(x) = \begin{cases} env2(x) & \text{falls } x \in Bereich(env2) \\ env1(x) & \text{andernfalls} \end{cases}$$

Informell überschattet also die (semantische) Umgebung $env2$ die Umgebung $env1$ in $(env1 + env2)$. Das Konstrukt $(env1 + env2)$ wird verwendet, wenn eine Deklaration (durch $env2$ dargestellt) eine vorhandene Umgebung (durch $env1$ dargestellt) verändert. Die veränderte Umgebung ist dann $(env1 + env2)$.

Ist `N` ein Name und w ein (semantischer) Wert, so bezeichnet

$$[w/\texttt{N}]$$

die (semantische) Umgebung mit Bereich {`N`}, die `N` auf w abbildet. Die (semantische) Umgebung $[w/\texttt{N}]$ liefert also eine Abbildung für keinen weiteren Namen als `N`.

Die „Summe“ $(env + [w/\texttt{N}])$ einer (semantischen) Umgebungen env mit einer (semantischen) Umgebungen $[w/\texttt{N}]$ entspricht also der Veränderung einer Umgebung, die aus folgender Deklaration folgt:

```
val N = w;
```

13.3.9 Denotationelle Semantik einer rein funktionalen Programmiersprache

Es wäre mühsam und für die Programmierpraxis nicht sehr hilfreich, wenn der Programmentwickler für jede mögliche Funktionsdeklaration (wie etwa die Funktionsdeklaration `fak2`) vom Abschnitt 13.3.6) eine Denotation (wie die (mathematische) Funktion $f0$ für `fak2` — siehe Abschnitt 13.3.6) geben müsste. Statt dessen wird die Denotation jeder beliebigen Funktionsdeklaration systematisch aus dieser Funktionsdeklaration generiert. Dies wird dadurch ermöglicht, dass der Implementierung der Programmiersprache selbst eine Denotation gegeben wird. Man spricht dann von einer Semantikfunktion für die Programmiersprache.

In diesem Abschnitt wird skizziert, wie eine Semantikfunktion für die Programmiersprache SMalL definiert werden kann. SMalL wird hier betrachtet, weil einerseits SMalL eine starke Einschränkung von SML ist, andererseits über die wichtigsten Merkmale einer rein funktionalen Programmiersprache wie etwa lokale Deklarationen, Überschatten, Rekursion und Funktionen höherer Ordnung verfügt. Der Einfachheit halber wird die Fassung der Programmierprache SMalL betrachtet, die keine Ausnahmen kennt, also die Fassung, deren Programme von den Auswertern `eval2` und `eval3` (siehe Abschnitt 10.5) bearbeitet werden.

Im Folgenden wird also erläutert, wie eine Semantikfunktion definiert werden kann, die den Auswertern `eval2` (oder `eval3`) entspricht.

Die Anwendung der Semantikfunktion (die `eval2` oder `eval3` entspricht) auf eine Funktionsdeklaration (in SMalL) liefert die Denotation dieser Funktionsdeklaration. In dieser Vorgehensweise lässt sich derselbe metasprachliche Ansatz erkennen wie zur Spezifikation des Auswertungsalgorithmus (vgl. Abschnitt 3.1.3 und 3.1.5) oder zur Implementierung eines Auswerters (Abschnitt 10.9).

Bevor die Semantikfunktion für SMalL definiert wird, werden die syntaktischen und semantischen Domänen sowie die abstrakte Syntax festgelegt, auf die sich die Semantikfunktion bezieht.

Syntaktische (Wert-)Domäne

Exp	: die Menge der SMalL-Ausdrücke
Var	: die Menge der Namen (oder Variablen, Bezeichnern)
$ConsExp$	: die Menge der konstanten Ausdrücke
Def	: die Menge der SMalL-Deklarationen

Jede dieser vier Mengen ist unendlich und abzählbar. Man beachte, dass z.B. die Menge Var nicht die Menge der Namen ist, die in einem gegebenen SMalL-Programm tatsächlich vorkommen, sondern die Menge aller möglichen Namen, die in SMalL-Programme vorkommen können.

Für SMalL enthält $ConsExp$ nur die Ausdrücke wie etwa `007` und `~12`, die die ganze Zahlen darstellen. Wir erinnern daran (siehe Abschnitt 10.1.1), dass SMalL nur zwei Typen hat: Die ganzen Zahlen und die Funktionen. SMalL hat also nur einen Konstantentyp, die ganzen Zahlen.

Es gilt:

$$\begin{aligned} Var &\subset Exp \\ ConsExp &\subset Exp \\ Def &\subset Exp \end{aligned}$$

weil sowohl Namen als auch Deklarationen Ausdrücke sind (siehe Abschnitt 2.2 und 2.4).

Abstrakte Syntax

Die Definition einer Semantikfunktion für eine Programmiersprache bezieht sich auf die abstrakte Syntax dieser Programmiersprache. Es wird also auf die abstrakte Syntax von SMalL verwiesen (siehe Abschnitt 10.2), die formal beschrieben werden sollte (siehe Abschnitt 13.1). Der Einfachheit halber wird hier auf diese formale Syntaxbeschreibung verzichtet (Übung!).

Die Umwandlung eines SMalL-Programms in konkreter Syntax in ein SMalL-Programm in abstrakter Syntax sollte ebenfalls formal festgelegt werden. Der Einfachheit halber wird hier auch auf die formale Beschreibung dieser Umwandlung verzichtet. Diese Umwandlung ist Teil der Syntaxanalyse (siehe Abschnitt 13.1.3).

Eine (mathematische) Funktion $[\![\,.\,]\!]$ wird eingeführt, die einen SMalL-Ausdruck (z.B. ein SMalL-Programm) in konkreter Syntax `A` auf den entsprechenden SMalL-Ausdruck in abstrakter Syntax $[\![\,$`A`$\,]\!]$ abbildet.

Semantische (Wert-)Domäne

$ConsVal$:	die Domäne der semantischen Konstantenwerte;
	$ConsVal = \mathbb{Z}_\perp$, weil alle SMalL-Konstanten vom Typ `int` sind;
$FnVal$:	die Domäne der semantischen Funktionswerte;
Val:	die Domäne aller möglichen semantischen Werte;
Env:	die Menge der (semantischen) Umgebungen.

Seien die (unendlich vielen) Funktionsdomänen D_{ij} mit $i \in \mathbb{N}$ und $j \in \mathbb{N}$ wie folgt definiert:

$$\begin{array}{lll} D_{11} & = D(z,z) & = [\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp] \\ D_{12} & = D(z,(z,z)) & = [\mathbb{Z}_\perp \to_\perp D(z,z)] \\ D_{21} & = D((z,z),z) & = [D(z,z) \to_\perp \mathbb{Z}_\perp] \\ D_{22} & = D((z,z),(z,z)) & = [D(z,z) \to_\perp D(z,z)] \\ \vdots & \vdots & \vdots \end{array}$$

Da SMalL Funktionen höherer Ordnung zulässt und auf Auswertung in applikativer Reihenfolge beruht, ist $FnVal$ wie folgt definiert (vgl. Abschnitt 13.3.8):

$$FnVal = D(z,z) \oplus D(z,(z,z)) \oplus D((z,z),z) \oplus D((z,z),(z,z)) \oplus \ldots$$

Da SMalL auf Auswertung in applikativer Reihenfolge beruht, ist Val, die Domäne aller möglichen semantischen Werte, wie folgt definiert:

$$Val = ConsVal \oplus FnVal$$

Die Mengen $ConsVal$, $FnVal$, Val und Env sind keine beliebigen Mengen, sondern semantische Domänen. Das heißt, dass jede dieser Mengen ein bottom-Element enthält und dass über jede dieser Mengen eine Ordnung $\sqsubseteq$ definiert ist, bezüglich der das bottom-Element minimal ist.

Die Einschränkungen der Programmiersprache SMalL, u.a. die Tatsache, dass in SMalL nur einstellige Funktionen deklariert werden können, haben zur Folge, dass diese semantischen Domänen leicht zu definieren sind. Kämen z.B. Vektoren in SMalL vor, so müssten die zugehörigen semantischen Domänen entsprechend der Auswertungsart von SMalL, d.h. der Auswertung in applikativer Reihenfolge, definiert werden.

Die Semantikfunktion $SemVal$

$SemVal$ ist eine curried Funktion vom Typ: $Exp \to Env \to Val$. $SemVal$ erhält also als Argument einen SMalL-Ausdruck in abstrakter Syntax und liefert eine Funktion, die auf eine (semantische) Umgebung angewandt einen semantischen Wert liefert. Ist `A` ein SMalL-Ausdruck in konkreter Syntax und env eine semantische Umgebung, so ist also

$$SemVal([\!|\,\texttt{A}\,|\!])\ env$$

auch geschrieben

$$SemVal\ [\!|\,\texttt{A}\,|\!]\ env$$

ein semantischer Wert.

$SemVal$ ist keine Programmfunktion, sondern eine (mathematische) Funktion. Auch wenn es in der Mathematik nicht üblich ist, ist es möglich und hier sinnvoll, die (mathematische) Funktion $SemVal$ als Funktion in curried Form zu definieren und ihren Typ ähnlich wie in der funktionalen Programmierung anzugeben.

Die Definition von $SemVal$ kann mit einer Fallunterscheidung wie folgt skizziert werden:

$$SemVal\ [\!|\,\texttt{C}\,|\!]\ env = c \in ConsVal = \mathbb{Z}_\perp \qquad \text{falls } \texttt{C} \in ConsExp$$

Selbstverständlich muss jedem konstanten Ausdruck (wie etwa `007`) das entsprechende Element in $ConsVal$ (wie etwa 7 für den konstanten Ausdruck `007`) zugewiesen werden.

$$SemVal\ [|\,\texttt{unop A}\,|]\ env = \begin{cases} unop'\ SemVal\ [|\,\texttt{A}\,|]\ env & \text{falls } SemVal\ [|\,\texttt{A}\,|]\ env \neq \bot \\ \bot & \text{andernfalls} \end{cases}$$

wobei $unop'$ die unäre Operation in $ConsVal$ ist, die dem Ausdruck `unop` entspricht.

$$SemVal\ [|\,\texttt{A1 binop A2}\,|]\ env = \begin{cases} SemVal\ [|\,\texttt{A1}\,|]\ env\ binop'\ SemVal\ [|\,\texttt{A2}\,|]\ env \\ \quad \text{falls } SemVal\ [|\,\texttt{A1}\,|]\ env \neq \bot \\ \quad \text{und } SemVal\ [|\,\texttt{A2}\,|]\ env \neq \bot \\ \bot \quad \text{andernfalls} \end{cases}$$

wobei $binop'$ die Binäroperation in $ConsVal$ ist, die dem Ausdruck `binop` entspricht.

$$SemVal\ [|\,\texttt{if Test then A1 else A2}\,|]\ env$$
$$= \begin{cases} SemVal\ [|\,\texttt{A1}\,|]\ env & \text{falls die Auswertung von } \texttt{Test} \text{ terminiert} \\ & \text{und } \texttt{Test} \text{ in } env \text{ erfüllt ist} \\ SemVal\ [|\,\texttt{A2}\,|]\ env & \text{falls die Auswertung von } \texttt{Test} \text{ terminiert} \\ & \text{und } \texttt{Test} \text{ in } env \text{ nicht erfüllt ist} \\ \bot & \text{falls } \texttt{Test} \text{ den semantischen Wert } \bot \text{ hat} \end{cases}$$

Selbstverständlich muss die Erfüllung oder Nichterfüllung der Bedingung `Test` in einer gegebenen Umgebung mathematisch definiert werden. Der Einfachheit halber wird hier auf diese Definition verzichtet.

$$SemVal\ [|\,\texttt{V}\,|]\ env = env(\texttt{V}) \qquad \text{falls } \texttt{V} \in Var$$

$$SemVal\ [|\,\texttt{let D in A end}\,|]\ env = SemVal\ [|\,\texttt{A}\,|]\ env'$$
$$\text{wobei } env' = env + SemEnv\ [|\,\texttt{D}\,|]\ env$$

Die Definition der Semantikfunktion $SemEnv$ ist unten skizziert. Der Operator $+$ für (semantische) Umgebungen wurde in Abschnitt 13.3.8 eingeführt. In der konkreten Syntax von SMalL entsprechen `let`–Ausdrücke Sequenzen der abstrakten Syntax, die aus einer Deklaration und einem anderen Ausdruck, der keine Deklaration ist, bestehen.

$$SemVal\ [|\,\texttt{fn N => A}\,|]\ env = f \in FnVal \text{ mit:}$$
$$\text{für jedes } w \in Val : f(w) = SemVal\ [|\,\texttt{A}\,|]\ (env + [w/\texttt{N}])$$

$$SemVal\ [|\,(\texttt{fn N => A1})\ \texttt{A2}\,|]\ env = SemVal\ [|\,\texttt{A1}\,|]\ (env + [w2/\texttt{N}])$$
$$\text{wobei } w2 = SemVal\ [|\,\texttt{A2}\,|]\ env$$

$$SemVal\ [|\,\texttt{A1 A2}\,|]\ env$$
$$= \begin{cases} f(SemVal\ [|\,\texttt{A2}\,|]\ env) & \text{falls } SemVal\ [|\,\texttt{A1}\,|]\ env = f \in FnVal \\ \bot & \text{andernfalls} \end{cases}$$

$$SemVal\ [|\,\texttt{A1 ; A2}\,|]\ env$$
$$= \begin{cases} SemVal\ [|\,\texttt{A2}\,|]\ env & \text{falls } \texttt{A1} \notin Def \\ SemVal\ [|\,\texttt{A2}\,|]\ env1 & \text{andernfalls, wobei } env1 = SemEnv\ [|\,\texttt{A1}\,|]\ env \end{cases}$$

$$SemVal\ [|\,\texttt{val N = A}\,|]\ env = SemVal\ [|\,\texttt{A}\,|]\ env$$

Aus einer korrekten Definition einer Semantikfunktion $SemVal$ sollen Eigenschaften folgen wie etwa:

$$SemVal\ [|\,\texttt{let N = A1 in A2 end}\,|]\ env = SemVal\ [|\,\texttt{A2}\,|]\ (env + [w/\texttt{N}])$$

wobei $w = SemVal\ [|\,\texttt{A1}\,|]\ env$.

Die Semantikfunktion *SemEnv*

Die Funktion *SemEnv* ist eine (mathematische) Funktion in curried Form vom Typ $Def \rightarrow Env \rightarrow Env$. Sie bildet also eine Deklaration auf eine Funktion ab, die eine (semantische) Umgebung auf eine (semantische) Umgebung abbildet. Es sei daran erinnert, dass SMalL nur Sequenzen von Deklarationen, also keine `and`–verknüpften Deklarationen ermöglicht.

Ist `D` eine SMalL-Deklaration in konkreter Syntax und *env* eine (semantische) Umgebung, so definiert *SemEnv* [| `D` |] *env* lediglich die Bindungen, die wegen der Deklaration `D` die Umgebung *env* verändern. Die (mathematische) Umgebung, die sich aus einer (mathematischen) Umgebung *env* und einer Deklaration `D` ergibt, ist also (env+*SemEnv* [| `D` |] *env*).

Die Definition der Funktion *SemEnv* kann wie folgt skizziert werden:

SemEnv [| `D1 D2` |] $env = (env1 + env2)$

wobei $env2 =$ *SemEnv* [| `D2` |] $(env + env1)$ und $env1 =$ *SemEnv* [| `D1` |] *env*

SemEnv [| `D1 ; D2` |] *env* wird wie im vorangehenden Fall definiert.

SemEnv [| `local D1 in D2 end` |] $env = env2$

wobei $env2 =$ *SemEnv* [| `D2` |] $(env + env1)$ und $env1 =$ *SemEnv* [| `D1` |] *env*

In der konkreten Syntax von SMalL entsprechen `local`–Ausdrücke Ausdrücken der abstrakten Syntax, die Sequenzen von Deklarationen sind.

SemEnv [| `val N = A` |] $env = (env + [w/\texttt{N}])$

wobei $w =$ *SemVal* [| `A` |] *env*

Der letzte Fall ist die Definition der Umgebung, die sich aus der Deklaration einer (möglicherweise) rekursiven Funktion ergibt, also eine `val rec`–Deklaration. Die folgende Spezifikation gibt die (etwaige) Rekursivität der `val rec`–Funktionsdeklaration wieder:

SemEnv [| `val rec N = A` |] $env = env'$

wobei env' durch die folgende (rekursive!) Gleichung definiert ist:

$env' = (env + [w/\texttt{N}])$ mit $w =$ *SemVal* [| `A` |] env'

Man erkennt an der vorangehenden Gleichung in der semantischen Umgebung dieselbe Art zyklischer Verweise wie in der operationalen Umgebung (siehe Abschnitt 4.2.8 und 10.3.1).

Die vorangehende Spezifikation von *SemEnv* [| `val rec N = A` |] wirft zwei Fragen auf:

1. Existiert immer für jede (semantische) Umgebung *env* eine (semantische) Umgebung env', die der vorangehenden Spezifikation entspricht?
2. Falls ja, kann diese Umgebung berechnet werden?

Kann die erste Frage positiv beantwortet werden, dann ist die vorangehende Spezifikation ein Fehlschlag, weil sie eben das Ziel verfehlt, die sich aus einer `val rec`–Deklaration ergebende Umgebung (formal) zu definieren.

Eine positive Antwort auf die zweite Frage soll sicherstellen, dass ein Algorithmus zur Generierung der Umgebung env' aus [| `val rec N = A` |] und *env* bekannt ist. Ein reiner Existenzbeweis für env', der keinen Algorithmus zur Generierung von env' aus [| `val rec N = A` |] und *env* liefert, wäre sicherlich als formale Spezifikation einer Programmiersprache völlig unzufriedenstellend.

Bemerkung über die vorangehenden Definitionen der Semantikfunktionen $SemVal$ und $SemEnv$

Die vorangehenden Definitionen der Semantikfunktionen $SemVal$ und $SemEnv$ sind nicht vollständig sondern wurden nur skizziert. Vollständige Definitionen würden u.a. eine präzisere Behandlung der bottom-Elemente benötigen und eine sorgfältige Unterscheidung zwischen inkorrekten Ausdrücken, die statisch erkannt werden, und inkorrekten Ausdrücken, die erst dynamisch als solche erkannt werden, erfordern.

13.3.10 Fixpunktsemantik rekursiver Funktionen

Die Verwendung rekursiver Gleichungen zur Definition ist prinzipiell problematisch. Rekursive Gleichungen werden oft erst dann verstanden, wenn deren Auswertung betrachtet wird.

Wenn rekursive Funktionsdeklarationen passende operationale Spezifikationen darstellen, sind sie keineswegs zufriedenstellende deklarative, d.h. vom Auswertungsalgorithmus unabhängige, Definitionen. Es lohnt sich also zu untersuchen, was die Denotation, d.h. die mathematisch formalisierte Bedeutung, einer beliebigen rekursiven Funktionsdeklaration eigentlich ist.

Der Einfachheit halber wird im Folgenden keine wechselseitige rekursive Funktionsdeklarationen (siehe Abschnitt 4.2.8) betrachtet. Diese Annahme ist in den Werken üblich, die die Semantik der Rekursion behandeln, auch wenn sie selten erwähnt wird. Der Auschluss von wechselseitigen rekursiven Funktionsdeklarationen schränkt die Allgemeinheit des im Folgenden eingeführten Ansatzes aber nicht ein.

Von der Rekursion zu einer Definition am Beispiel der Fakultätsfunktion

Betrachten wir die folgende Implementierung in SML der Fakultätsfunktion:

```
- val rec fak = fn x => if x = 0
                        then 1
                        else x * fak (x - 1);
val fak = fn : int -> int
```

Die folgende Programmfunktion höherer Ordnung `fak'` ist nichtrekursiv, liefert jedoch die „Struktur“ der rekursiven Programmfunktion `fak`:

```
- val fak' = fn g => (fn x => if x = 0
                              then 1
                              else x * g (x - 1));
val fak' = fn : (int -> int) -> int -> int
```

Unter Verwendung der nichtrekursiven Programmfunktion höherer Ordnung `fak'` lässt sich die vorangehende rekursive Programmfunktion `fak` wie folgt als `h` neu definieren:

```
 - val rec h = (fn x => fak' h x);
val h = fn : int -> int

- h 4;
val it = 24 : int
```

Die Programmfunktion `h` weist dasselbe operationale Verhalten wie die Programmfunktion `fak` auf, obwohl `h` und `fak` syntaktisch unterschiedlich sind. Folglich müssen die Denotationen von `h` und `fak` dieselbe (mathematische) Funktion aus $[\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp]$ (siehe Abschnitt 13.3.8) sein.

Die Programmfunktion höherer Ordnung `fak'` kann zur Deklaration von nichtrekursiven Programmfunktionen verwendet werden, die die (rekursive) Programmfunktion `fak` approximieren (im Sinne der Approximationsordnung).

Als Hilfsmittel deklarieren wir zunächst eine Programmfunktion namens `undefiniert` vom Typ `int -> int`, die nicht terminiert, wenn sie auf irgendeine ganze Zahl angewandt wird:

```
- val rec undefiniert : int -> int = fn x => undefiniert x;
val undefiniert = fn : int -> int
```

Die Denotation der Programmfunktion `undefiniert` ist offensichtlich die folgende (mathematische) Funktion aus $[\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp]$:

$$\begin{array}{lccc} b: & \mathbb{Z}_\perp & \to & \mathbb{Z}_\perp \\ & n & \mapsto & \perp \end{array}$$

Die flache Domäne $\mathbb{Z} \cup \{\perp\}$ ist die Denotation des Typs `int`.

Man beachte, dass die (mathematische) Funktion b das bottom-Element der semantischen Domäne $[\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp]$ der stetigen und strikten (mathematischen) Funktionen der flachen Domäne $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$ (siehe Abschnitt 13.3.8) ist.

Seien die folgenden (unendlich vielen!) nichtrekursiven Programmfunktionen `faki` ($\texttt{i} \in \mathbb{N}$) wie folgt (gedanklich!) deklariert:

```
- val fak0 = undefiniert;
val fak0 = fn : int -> int

- val fak1 = fak' fak0;
val fak1 = fn : int -> int

- val fak2 = fak' fak1;
val fak2 = fn : int -> int

- val fak3 = fak' fak2;
val fak3 = fn : int -> int

- val fak4 = fak' fak3;
val fak4 = fn : int -> int

- val fak5 = fak' fak4;
val fak5 = fn : int -> int

...
```

Offenbar hätten die folgenden nichtrekursiven Deklarationen dieselben Programmfunktionen wie die vorangehenden Deklarationen definiert:

- Deklaration von fak1:

```
val fak1 = fn x => if x = 0 then 1 else x * fak0 (x - 1);
```
oder
```
val fak1 = fak' fak0;
```

- Deklaration von fak2:
```
val fak2 = fn x => if x = 0 then 1 else x * fak1 (x - 1);
```
oder
```
val fak2 =fak' (fak' fak0);
```

- Deklaration von fak3:
```
val fak3 = fn x => if x = 0 then 1 else x * fak2 (x - 1);
```
oder
```
val fak3 = fak' (fak' (fak' fak0));
```

- usw.

Durch einige Tests und durch einfache Beweise kann man sich leicht davon überzeugen (Übung!), dass das Folgende gilt:
(*) Für alle $\texttt{i} \in \mathbb{N} \setminus \{0\}$:

1. für alle $\texttt{j} \in \mathbb{Z}$ mit $0 \leq \texttt{j} \leq \texttt{i} - 1$ terminiert die Auswertung von `faki j` und liefert den Wert `j`!;
2. für alle $\texttt{j} \in \mathbb{Z}$ mit $\texttt{j} < 0$ oder $\texttt{j} \geq \texttt{i}$ terminiert die Auswertung von `faki j` nicht.

Unter Verwendung der (nichtrekursiven) Programmfunktionen $(\texttt{faki})_{\texttt{i}\in\mathbb{N}}$ kann die rekursive Programmfunktion `fak` wie folgt induktiv neu definiert werden:

Basisfall:	für $\texttt{i} < 0$	$\texttt{fak i} = \texttt{fak0 i}$
Induktionsfall:	für $\texttt{i} \geq 0$	$\texttt{fak i} = \texttt{fak}_{(\texttt{i}+1)}\ \texttt{i}$

oder auch wie folgt:

Basisfall:	für $\texttt{i} < 0$	$\texttt{fak i} = \texttt{fak0 i}$
Induktionsfall:	für $\texttt{i} \geq 0$	$\texttt{fak i} = (\texttt{fak}'^{(\texttt{i}+1)})\ \texttt{fak0 i} = (\texttt{fak}'^{(\texttt{i}+1)})\ \perp$

Dabei bezeichnet $(\texttt{fak}'^{(i)}\texttt{g})$ i Anwendungen der Programmfunktion höherer Ordnung `fak'` auf einen Ausdruck `g`, d.h.

$$(\texttt{fak}'^{(i)}g) = \underbrace{\texttt{fak}'(\texttt{fak}' \ldots (\texttt{fak}'}_{i \text{ mal}}\ \texttt{g}) \ldots))$$

Aus mathematischer Sicht ist die vorangehende induktive Neudefinition der Programmfunktion `fak` einwandfrei. Als Funktionsdeklaration ist sie aber unbrauchbar, weil SML wie jede andere Programmiersprache keine induktive Funktionsdefinition ermöglicht. Anstelle von induktiven Definitionen bieten Programmiersprachen eben rekursive Deklarationen an.

So unbrauchbar die vorangehende induktive Deklaration als Funktionsdeklaration auch sein mag, sie liefert die Denotation der rekursiven Programmfunktion `fak`:

Sei f_i die Denotation der nichtrekursiven Programmfunktion `faki`. Diese Denotation ist unproblematisch zu ermitteln, weil `faki` nicht rekursiv ist. Aus (*) folgt (3. folgt daraus, dass SML auf der Auswertung in applikativer Reihenfolge beruht):
(**) Für alle $i \in \mathbb{N} \setminus \{0\}$ gilt:

1. für alle $j \in \mathbb{Z}$ mit $0 \leq j \leq i-1$ gilt: $f_i(j) = j!$;
2. für alle $j \in \mathbb{Z}$ mit $j < 0$ oder $j \geq i$ gilt: $f_i(j) = \bot$;
3. $f_i(\bot) = \bot$.

Es gilt ferner:

$$\bot = f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq f_{n+1} \sqsubseteq \ldots$$

Die Folge $(f_i)_{i\in\mathbb{N}}$ ist also eine $\sqsubseteq$–Kette in $D = [\mathbb{Z}_\bot \to_\bot \mathbb{Z}_\bot]$. ($D$ ist die (semantische) Domäne der strikten und stetigen (mathematischen) Funktionen von der flachen Domäne $\mathbb{Z}_\bot$ in $\mathbb{Z}_\bot$ ist.)

Im nächsten Abschnitt wird gezeigt, dass jede solche $\sqsubseteq$–Kette von (mathematischen) Funktionen aus einer Menge $D = [D1 \to_\bot D2]$ oder $D = [D1 \to D2]$ eine kleinste obere Schranke $s \in D$ hat. Diese kleinste obere Schranke s ist also eine strikte bzw. nicht notwendigerweise strikte (mathematische) Funktion aus D, je nach dem, ob $D = [D1 \to_\bot D2]$ oder $D = [D1 \to D2]$.

Da s eine obere Schranke der $\sqsubseteq$–Kette $(f_i)_{i\in\mathbb{N}}$ ist, folgt aus (**):

Für alle $i \in \mathbb{N}$ gilt:

1. für alle $j \in \mathbb{Z}$ mit $0 \leq j \leq i-1$ gilt: $f_i(j) = j! \sqsubseteq s(j)$;
2. für alle $j \in \mathbb{Z}$ mit $j < 0$ oder $j \geq i$ gilt: $f_i(j) = \bot \sqsubseteq s(j)$;
3. $s(\bot) = \bot$ (weil sonst s keine kleinste obere Schranke der Kette $(f_i)_{i\in\mathbb{N}}$ wäre).

Folglich:

1. Für alle $j \in \mathbb{Z}$ mit $0 \leq j$ gilt: $s(j) = j!$;
2. Für alle $j \in \mathbb{Z}$ mit $j < 0$ gilt: $s(j) = \bot$ (weil sonst s keine kleinste obere Schranke der Kette $(f_i)_{i\in\mathbb{N}}$ wäre);
3. $s(\bot) = \bot$.

Man beachte, dass die kleinste obere Schranke s der Kette $(f_i)_{i\in\mathbb{N}}$ die gewünschte Denotation von `fak` ist. Im nächsten Abschnitt wird gezeigt, dass die Denotation jeder rekursiven Funktionsdeklaration sich in ähnlicher Weise definieren lässt.

Systematische Herleitung einer Funktionsdefinition aus einer beliebigen rekursiven Funktionsdeklaration — Der Fixpunktsatz

Die Vorgehensweise zur Herleitung der Denotation der rekursiven Programmfunktion `fak` aus ihrer Deklaration, die im vorangehenden Abschnitt erläutert wurde, lässt sich problemlos auf jede beliebige rekursive Funktionsdeklaration übertragen.

Sei zunächst daran erinnert, dass eine rekursive Funktionsdeklaration einer Deklaration der Gestalt

```
val rec funk = expr
```

entspricht, wobei `expr` ein Ausdruck ist. Ähnlich wie die Programmfunktion höherer Ordnung `fak'` aus der rekursiven Deklaration der Programmfunktion `fak` gewonnen werden kann, kann eine Programmfunktion höherer Ordnung `funk'` aus einer beliebigen rekursiven Deklaration einer Programmfunktion `funk` hergeleitet werden:

```
val funk' = fn funk => expr
```

Dabei wird angenommen, dass der Name `funk'` im Ausdruck `expr` nicht vorkommt.
Das Überschatten (siehe Abschnitt 3.5.1) hat zur Folge, dass im definierenden Ausdruck

```
fn funk => expr
```

der Deklaration der Programmfunktion höherer Ordnung `funk'` der Name `funk` ein formaler Parameter ist.
Wie im Fall der Programmfunktion `fak` kann eine (beliebige) Programmfunktion `funk` unter Verwendung der Programmfunktion höherer Ordnung neu definiert werden:

(#) `val rec funk = funk' funk`

ähnlich wie im Fall der Prgrammfunktion `fak` kann die Programmfunktion `funk'` verwendet werden, um die folgende Folge von nichtrekursiven Programmfunktionen $(\texttt{funk}_i)_{i\in\mathbb{N}}$ zu definieren, wobei `undefiniert` die im vorangehenden Abschnitt eingeführten Programmfunktion ist, die auf keinem Wert definiert ist:

```
funk0 = undefiniert
funk1 = funk'(funk0) = funk'(undefiniert)
funk2 = funk'(funk1) = funk'(funk'(undefiniert))
    ⋮
funki = funk'(funki-1) = funk'(funk'...(funk' undefiniert)...))
                                 \_____/
                                 i mal
```

Aus der rekursiven Funktionsdeklaration (#) folgt, dass die Denotation von `funk` und die Denotation von `funk'(funk)` gleich sein sollen:

Denotation von `funk` = Denotation von `funk'(funk)`

Ist also F die Denotation der (nichtrekursiven) Programmfunktion `funk'` und f die Denotation der Programmfunktion `funk`, so soll gelten:

$$f = F(f)$$

Anders ausgedrückt heißt dies, dass die Denotation f der (rekursiven) Programmfunktion `funk` eine Lösung der rekursiven Gleichung $g = F(g)$ sein soll.
Eine Lösung dieser Gleichung wird *Fixpunkt von F* genannt. Ist ein Fixpunkt von F für die Approximationsordnung kleiner als jeder andere Fixpunkt von F, so spricht man von einem *kleinsten Fixpunkt*.
Wie bereits in Abschnitt 13.3.8 erwähnt stellen sich zwei Fragen:

1. Hat die rekursive Gleichung $g = F(g)$ immer mindestens eine Lösung?
2. Hat die Gleichung $g = F(g)$ mehrere Lösungen, welche dieser Lösungen soll als Denotation der Funktionsdeklaration `funk` ausgewählt werden?

Der folgende Fixpunktsatz von Kleene liefert Antworten auf beide Fragen. Er zeigt, dass die Denotation f der (rekursiven) Programmfunktion `funk` der kleinste Fixpunkt der rekursiven Gleichung $g = F(g)$ ist.

Bevor der Satz gegeben und bewiesen wird, wird zunächst eine Notation eingeführt.

Ist $D1$ die Denotation der Ursprungsmenge der Programmfunktion `funk` und $D2$ die Denotation der Bildmenge von `funk`, so ist die (mathematische) Funktion f ein Element von $D = [D1 \to D2]$ und die (mathematische) Funktion F ein Element von $[D \to D]$.

Man definiert Potenzen von F wie folgt:

$$\begin{aligned} F^0 &= \bot_D, && \text{d.h. die konstante (mathematische) Funktion, die jedes Element} \\ & && \text{von } D1 \text{ auf das bottom-Element von } D2 \text{ abbildet.} \\ F^1 &= F(F^0) \\ \vdots &\ \vdots \\ F^{i+1} &= F(F^i) \end{aligned}$$

Eine Funktion F^i ist also ein Element von $[D1 \to D2]$.

Fixpunktsatz (Kleene)
Sei D eine Domäne. Sei $F \in [D \to D]$, d.h. eine stetige Funktion von D in D.

1. Die rekursive Gleichung $g = F(g)$ mit einer Unbekannten $g \in D$ hat immer mindestens eine Lösung.
2. Die kleinste obere Schranke der Folge $(F^i)_{i \in \mathbb{N}}$ ist eine Lösung der rekursiven Gleichung $g = F(g)$, die für die Approximationsordnung $\sqsubseteq$ über D kleiner ist als jede andere Lösung dieser rekursiven Gleichung.

Beweis:

a. Die Folge $(F^i)_{i\in\mathbb{N}}$ ist eine $\sqsubseteq$–Kette in $D = [D1 \to D2]$, wobei $\sqsubseteq$ die Approximationsordnung über $D = [D1 \to D2]$ ist — siehe Abschnitt 13.3.8.

Nach Definition von $F^0 = \bot_D$ gilt: $F^0 \sqsubseteq F^1$.

Sei $i \in \mathbb{N}$, $d \in D1$. $F^{i+1}(d) = F(F^i(d))$. Da F stetig, ist F monoton. Folglich ist $F^i(d) \sqsubseteq F(F^i(d)) = F^{i+1}(d)$. Da i und d beliebig gewählt wurden, ist $(F^i)_{i\in\mathbb{N}}$ eine $\sqsubseteq$–Kette.

b. Da D nach Annahme eine Domäne ist, besitzt die $\sqsubseteq$–Kette $(F^i)_{i\in\mathbb{N}}$ eine kleinste obere Schranke in D. Sei s diese kleinste obere Schranke. Wir zeigen nun, dass s ein Fixpunkt von F, d.h. eine Lösung der rekursiven Gleichung $g = F(g)$ ist:

Da F nach Annahme stetig ist, ist $F(s)$ (nach Definition der Stetigkeit) die kleinste obere Schranke der Folge $(F(F^i))_{i\in\mathbb{N}}$.

Die Folge $(F(F^i))_{i\in\mathbb{N}}$ ist aber identisch mit der Folge $(F^i)_{i\in\mathbb{N}\setminus\{0\}}$.

Die kleinste obere Schranke s der Folge $(F^i)_{i\in\mathbb{N}}$ ist ebenfalls die kleinste obere Schranke der Folge $(F^i)_{i\in\mathbb{N}\setminus\{0\}}$, d.h. $s = F(s)$.

[qed]

c. Aus b. folgt, dass die rekursive Gleichung $g = F(g)$ immer mindestens eine Lösung besitzt (Punkt 1 des Fixpunktsatzes).

d. Aus b. folgt auch, dass die kleinste obere Schranke s der Folge $(F^i)_{i\in\mathbb{N}}$ eine Lösung der Gleichung $g = F(g)$ ist. Wir zeigen, dass s für die Approximationsordnung $\sqsubseteq$ über D kleiner als jede andere Lösung von $g = F(g)$ ist.

Sei t eine Lösung von $g = F(g)$, d.h. $t = F(t)$ und $t \in D$.

1. Wir zeigen durch vollständige Induktion über i, dass für jedes $i \in \mathbb{N}$ gilt:

$$F^i \sqsubseteq t$$

Basisfall: $F^0 = \bot_D \sqsubseteq t$.
Nach Definition einer Domäne gilt $\bot_D \sqsubseteq d$ für jedes $d \in D$. Nach Annahme ist F monoton. Es folgt also: $F^0 = \bot_D \sqsubseteq t$.
Induktionsfall:
Induktionsannahme: Für ein $i \in \mathbb{N}$, $F^i \sqsubseteq t$.
Wir zeigen nun, dass $F^{i+1} \sqsubseteq t$.
Nach Induktionsannahme und da F stetig, also folglich auch monoton ist, gilt:

$$F(F^i(t)) \sqsubseteq F(t)$$

Nach Annahme gilt $t = F(t)$, also:

$$F^{i+1} = F(F^i(t)) \sqsubseteq t.$$

Der Induktionsfall ist also bewiesen.

2. Da nach Annahme s die kleinste obere Schranke der Kette $(F^i)_{i\in\mathbb{N}}$, ist s kleiner als t für die Approximationsordnung, d.h. $s \sqsubseteq t$.

qed.

Der Fixpunktsatz besagt also, dass die Denotation einer rekursiven Funktionsdeklaration (als Gleichung betrachtet) ihr kleinster Fixpunkt ist.

Praktische Relevanz des Fixpunktsatzes

Der Fixpunktsatz ist nicht nur zur (mathematischen) Formalisierung der Semantik der Rekursion vom Belang. Sein Beweis beruht auf einer Berechnungstechnik, die in vielen Bereichen der praktischen Informatik Anwendung findet.

Ein Verfahren, dass Lösungen berechnet, bis keine neue hinzukommen, kann als *Fixpunktberechnung* formalisiert werden. Solche Fixpunktberechnungen können endlich sein, wenn es endlich viele Lösungen gibt, oder unendlich sein, wenn es unendlich viele Lösungen gibt.

Auch wenn es unendlich viele Lösungen gibt, kann eine Fixpunktberechnung nützlich sein. Unter gewissen Voraussetzungen liefert eine Fixpunktberechnung jede einzelne Lösung nach endlicher Zeit, so dass sie zur Auswahl einer (nach gewissen Kriterien) passenden Lösung verwendet werden kann.

Beispiele solcher Fixpunktberechnungen sind u.a. bei Deduktionssystemen, deduktiven Datenbanksystemen und bei der Ähnlichkeitsuche in Datenbanken zu finden.[4]

Literaturhinweise

Die Definition einer denotationellen Semantik für Programmiersprachen, die nicht notwendigerweise rein funktional sind, sowie anderer verwandter Ansätze zur Beschreibung der Semantik von Programmiersprachen sind im folgenden Buch ausführlich behandelt. In diesem Buch werden nicht nur semantische Domänen wie im Abschnitt 13.3.8 betrachtet, sondern auch andere:

D. A. Schmidt:
Denotational Semantics — A Methodology for Programm Development.
Allyn and Bacon, 1986.

Die Fixpunkttheorie, d.h. die Mathematik, die der Semantik der Rekursion zugrunde liegt, ist im folgenden Buch behandelt:

L. Paulson:
Logic and Computation.
Cambridge Tracts in Theoretical Computer Science (2),
Cambridge University Press, 1987.

Eine Einführung in die denotationelle Semantik aus theoretischer Sicht ist im folgendem Artikel zu finden:

P. D. Moses:
Denotational Semantics.
in: Handbook of Theoretical Computer Science, Volume B (Formal Models and Semantics), Elsevier and MIT Press, 1990.

Eine vollständige formale Spezifikation von SML unter Verwendung eines Ansatzes namens „natürliche Semantik“, der dem Ansatz „denotationelle Semantik“ nahe verwandt, jedoch etwas weniger abstrakt ist, d.h. auch Bezug auf operationale Aspekte nimmt, ist im folgenden Werk spezifiziert:

R. Harper, R. Milner, and M. Tofte:
The Semantics of Standard ML (Version 1).
Report ECS-LFCS-87-36, Computer Science Department, Edingburgh University, 1998.

[4] siehe die Hauptstudiumvorlesungen zu diesen Themen