Einführung in SML und CML

Übungsunterlagen
für Programmiermethoden Praktikum I
Informatikpraktikum I
Softwareparadigmen

Verfaßt von:
Andreas Bollin
TR-GEN-940304-GE-003-BOLLIN

1997
IST - Softwaretechnologie
Technische Universität Graz

Vorwort

Die Übungen "Programmiermethoden Praktikum I" (PMP1), "Informatikpraktikum I" (IMP1) und die Vorlesung über "Softwareparadigmen" (SWP) sind der Einführung und den grundlegenden Strukturen der Programmierung gewidmet. Eine Programmiersprache ist meist einem bestimmten Modell zugeordnet und betont bestimmte methodische Aspekte. Man spricht dann von funktionalen, imperativen oder relationalen Sprachen, bzw. von strukturierten und objektorientierten Sprachen.

Es scheint nun nicht sinnvoll, in einer einsemestrigen Vorlesung (SWP) oder einer Übung (IMP1, PMP1) für jeden Aspekt eine für diesen Aspekt besonders spezialisierte Sprache zu verwenden, weil die Hörer dann gezwungen wären, sich eine verwirrende Fülle verschiedener Notationen anzueignen. Der eigentliche Gegenstand der Vorlesung SWP und den Übungen PMP1 und IMP1 sind semantische Strukturen und nicht notationelle syntaktische Konstruktionen. Insbesondere für die Vorlesung sollte daher eine möglichst allgemeine Sprache gewählt werden, deren Notation möglichst einfach ist.

Für dieses Vorhaben bietet sich heute besonders ML, genauer gesagt SML (Standard ML), an. Viele der Grundstrukturen, die in der Vorlesung besprochen werden, lassen sich in SML darstellen. ML besitzt die volle Allgemeinheit einer funktionalen Sprache, hat aber auch grundlegende imperative Komponenten und neuerdings auch Konstrukte für parallele Abläufe (CML). ML besitzt das allgemeinste statische Typensystem (das Hindley-Milner Type System), das heute bekannt ist. ML bietet auch ein modernes Modulkonzept zur Strukturierung großer Systeme. Seit dem Sommersemester 1994 wird SML in der Vorlesung SWP, seit dem Wintersemester 1997/98 auch in den Übungen PMP1 und IMP1 verwendet. Das bewährte Scriptum in SWP wird nun auch für PMP1 und IMP1 verwendet und enthält ein Kapitel mehr als für PMP1 und IMP1 notwendig.

Achtung! Dieses SKriptum beinhaltet eine Einführung in CML, der parallelen Erweiterung von SML. CML wird nur(!) in der Vorlesung und in den Übungen aus SWP benoetigt.

Für die praktische Arbeit mit SML stehen eine Reihe von Implementierungen zur Verfügung. Alle diese Implementierungen sind "public domain software", stehen also frei zur Verfügung. Für die Übungen wird Standard ML of New Jersey verwendet. Zur Zeit (Stand September 1997) gibt es Implementierungen von SML unter Windows 95/NT, Linux, FreeBSD, OS/2 und Unix (AIX, Solaris, SunOS). SML läuft auch unter Windows 3.11 mit der 32-bit Erweiterung (ist aber aus Performancegründen nicht zu empfehlen). Implementierungen von CML gibt es im Augenblick nur für Unix und Linux. Sämtliche Distributionen sind am Institutsserver **ftp.ist.tu-graz.ac.at** oder auf den Servern der Entwicklergruppen zu finden. Nähere Informationen dazu entnehmen Sie bitte unserer Homepage

http://www.ist.tu-graz.ac.at

Das Skriptum soll den Gebrauch der Sprache und die Benützung des Systems für die Übungen erleichtern. Dabei werden, soweit als möglich, zu den englischen Fachbegriffen die deutschen Übersetzungen mit angegeben. Im Skriptum wird versucht, die wichtigsten Befehle und bedeutendsten Methoden von Standard ML of New Jersey vorzustellen. Es folgt dabei hauptsächlich den Büchern von L.C. Paulsen [L.C91] und J.D. Ullmann [Ull94], die für zusätzliche Lektüre auch zu empfehlen sind. Eine ausgezeichnete Einführung in die funktionale Programmierung ist auch in [Wik90] zu finden. Zusätzliche Literaturhinweise zu SML ([RM90], [RM91], [Har93]), funktionaler Programmierung ([Wik90], [RB88] und [Jon87]) und parallele Programmierung ([And91]) sind auch im Anhang zu finden. Sollten Sie Wünsche oder Anregungen in Bezug auf die Unterlagen haben, so schicken Sie bitte eine email an bollin@ist.tu-graz.ac.at.

Graz, September 1997, Andreas Bollin

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Aufbau der Übungsunterlagen

- Einführung in die funktionale Welt
- Benutzung des Systems
- Kleiner Exkurs durch die Sprache
  - Name, Funktionen, Typen
  - Listen
  - Funktionen höherer Ordnung
  - Module
- Übungsbeispiele

1.2 Einführung in die Welt der funktionalen Programmiersprachen

Funktionale Programme arbeiten mit values (Werten), nicht mit states (Zuständen). Ihre Werkzeuge sind expressions (Ausdrücke) und nicht commands (Kommandos). Trotzdem gibt es Methoden um Zuweisungen, Arrays und Schleifen zu verwirklichen. Zunächst einige wichtige Begriffe im Zusammenhang mit funktionalen Sprachen:

- Das funktionale Modell

  Während das konventionelle Maschinenmodell von schrittweise ablaufenden Folgen von Aktionen (Operationen) ausgeht, greift das funktionale Modell auf die Definition des mathematischen Funktionsbegriffs zurück. Funktionen sind hier Abbildungen von einem abstrakten Definitionsbereich auf einen abstrakten Wertebereich. Der Programmierer geht hierbei von gewissen Grundfunktionen aus und kann aus ihnen neue

Funktionen aufbauen. Somit repräsentiert das gesamte Programm eine Funktion vom Definitionsbereich der Eingangsdaten in den Wertebereich der Ausgangsdaten.

- Funktionen

  Sie sind, wie schon erwähnt, hauptsächlich wiederum mit Hilfe von Funktionen definiert, wobei es für Funktionen nur wenige und wohldefinierte Einschränkungen gibt. Argumente können von jedem beliebigen Typ sein, und auch das Ergebnis kann beliebigen Typs sein. Das beinhaltet auch Funktionen als Argumente und als Ergebnisse.

- Funktionen höherer Ordnung

  Funktionen fungieren selbst wie berechenbare Werte. Eine Funktion höherer Ordnung ist eine Funktion, die Funktionen als Argument übernimmt und möglicherweise wieder Funktionen als Ergebnis liefert. Eine typische Funktion höherer Ordnung ist die map-Funktion, die wir später genauer untersuchen wollen.

- Unendliche Datenstrukturen (Lazy-Lists)

  Sie werden mittels 'lazy evaluation' implementiert. Das heißt, daß kein Wert dieser (theoretisch unendlich langen) Liste berechnet wird, ehe er nicht gebraucht wird, um das entgültige Resultat zu erhalten. Unendliche Listen werden somit nie vollständig realisiert, vielmehr existiert ein Prozeß um sukzessive Elemente zu erzeugen.

- Ein- und Ausgabe

  Es gibt zwei Möglichkeiten Ein- und Ausgabe in eine funktionale Sprache einzubauen. Entweder behandelt man einen Input oder Output als eine unendlich lange Liste, oder man fügt der funktionalen Sprache einige imperativen Elemente hinzu. Dazu mehr aber ebenfalls später.

SML ist keine reine funktionale Programmiersprache, denn sie enthält neben reinen funktionalen Elementen auch Zuweisungen (Ergibt Anweisungen) und Kommandos für Eingabe und Ausgabe. Auch Lazy-Lists können dargestellt (simuliert) werden.

Kapitel 2

Benutzung des Systems

2.1 Arbeiten mit SML

Der Compiler von Standard ML of New Jersey ist ein interaktives System, in das man Deklarationen und Ausdrücke eingeben oder Source Code laden und anschließend ausführen kann.

- Starten von SML

  Sobald man sich in einer Unix-Shell befindet, tippt man "sml" ein, und kurze Zeit später antwortet das System mit dem Top-Level Prompt ("-"). Es gibt auch noch ein Low-Level-Prompt ("=").

- Interaktive Eingabe

  Jede Eingabe muß mit einem Strichpunkt (Semikolon) ";" abgeschlossen werden. Danach folgt die Eingabe-Taste (Enter). Ist eine Eingabe noch nicht mit einem Semikolon abgeschlossen und schon die Entertaste gedrückt, so erscheint das Low-Level-Prompt. Es mag vielleicht irritieren, hat aber keinen Einfluß auf ihre Funktion. Am Besten probieren Sie gleich aus:

  ```
  - 3;                    (* Nun die ENTER Taste ! *)
  val it  = 3 : int       (* Antwort von SML ! *)
  ```

  Es ist noch zu beachten, daß alle Eingaben in SML **case-sensitive** sind (Groß- und Kleinbuchstaben werden unterschieden).

- Unterbrechen

  Eine Eingabe oder ein laufendes Programm kann jederzeit mit CTRL-C abgebrochen werden.

- Laden eines Source Codes

  Dazu wird eine Funktion *use* benutzt, die als Argument einen String benötigt. Ein String wird von zwei Hochkommazeichen eingeschlossen. Der String wird als Filename interpretiert. Ein Beispiel dafür ist:

  ```
  - use "Filename";
  ```

- Speichern eines Images

  Hier heißt die Funktion *exportML* und nimmt wieder einen String als Argument. Das gespeicherte File beinhaltet auch die gesamte Umgebung (braucht sehr viel Speicherplatz) und kann jederzeit exekutiert werden.

- Beenden des Systems

  Nach dem Drücken von CTRL-D (end of file) befindet man sich wieder in der gewohnten Shell. Wenn Sie kein Image abgespeichert haben, ist alles was Sie bis jetzt interaktiv eingetippt haben unwiderruflich verloren.

Ein Tip: Um zu verhindern, daß massenhaft Speicher für Images verbraucht wird, sollte man ein einfaches Textfile mit einem Editor editieren und dann nach dem Starten von SML mit *use* laden. Das ist vielleicht nicht so bequem, man geht aber viel sorgsamer mit den zur Verfügung stehenden Resourcen um.

Kapitel 3

Namen,Funktionen und Typen

3.1 Allgemeines

Die meisten funktionalen Sprachen sind interaktiv (wie auch SML). Das hat den Vorteil, daß Funktionen Stück für Stück zusammengesetzt und erprobt werden können.
Im einfachsten Fall ist SML eine einfache Rechenmaschine, die die Typen Integer und Real kennt. Nach dem Prompt kann man z.B. eingeben:

```
        - 2+2;
        val it = 4 : int
oder
        - 3.2 - 2.3;
        val it = 0.9 : real
oder
        - sqrt 2.0;
        val it = 1.414213562 : real
```

Der Name `it` bezeichnet immer das letzte Ergebnis. ML antwortet mit dem Wert und mit dem Ergebnistyp. Im folgenden wollen wir uns mit

- Grundlegendem
- Numbers, Strings und Truth Values
- Paare, Tupel und Records
- Operatoren und Typen
- Rekursive Funktionen
- Deklarationen

auseinandersetzen.

3.2 Grundlegendes

ML kennt viele Klassen von Dingen: Werte, Typen, Signaturen, Strukturen und Funktoren. Wir wollen uns nun mit den ersten beiden näher beschäftigen (Values and Types).

3.2.1 Variablen und Funktionen

Eine Wertdeklaration ist sehr einfach. Man verwendet das Schlüsselwort *val*. Diese Deklaration ist mit einer **Konstantendeklaration** (static binding) vergleichbar. Im folgenden werden wir, wie die Autoren der im Anhang angegebenen Literaturhinweise, diese Konstanten auch als Variablen bezeichnen. Man sollte aber den Unterschied in der Bedeutung nicht vergessen und Verwechslungen mit Variablen in der Mathematik und anderen Programmiersprachen (Pascal, ..) vermeiden!

```
- val pi = 3.14159;
val pi = 3.14159 : real

- val r = 2.0;
val r = 2.0 : read

- pi * r * r;
val it = 12.56636 : real
```

Der Name `it` ist eine Systemvariable und nimmt immer den Wert des letzten Ausdrucks an, wobei jeder vorherige Wert von `it` verlorengeht. Um den Wert der Kreisfläche zu speichern, gibt es nun folgende Möglichkeiten:

```
        - val area = pi * r * r;
oder
        - val area = it;

        val area = 12.56636 : real
```

Funktionen werden ähnlich definiert. Hier verwendet man anstelle von *val* das Schlüsselwort *fun*.
Funktionsdeklarationen beginnen mit dem Wort *fun*, darauf folgt der Funktionsname und der Parameter gefolgt von einem Gleichheitszeichen (=) und dem sog. Körper (**Body**).

```
- fun area (r) = pi * r * r;
val area = fn : real -> real
```

Die Antwort von SML wird folgendermaßen interpretiert: Nach der mathematischen Notation ist *area* eine Funktion, welche eine Realzahl als Argument nimmt und wieder eine Realzahl als Ergebnis liefert. Es ist noch anzumerken, daß das Argument nicht unbedingt in Klammern gesetzt werden muß.
Der Aufruf der Funktion erfolgt wieder interaktiv:

```
        - area (2.0);
oder
        - area 2.0;
        val it = 12.56636 : real
```

ML unterscheidet zwei Arten von Identifier : *alphabetic names* und *symbolic names.*

- Alphabetische Identifier müssen mit einem Buchstaben beginnen, dann können beliebig viele Zahlen, Buchstaben, Underlines (_) und Quotes (') folgen. Es sollten jedoch vordefinierte Keywords vermieden werden.
- Bei symbolischen Identifiern sollte wieder auf die von ML reservierten symbolischen Namen achtgegeben werden. Prinzipiell bestehen sie aus den Zeichen:

  ! % & $ ♯ + − ∗ / : < = > ‘ ^ —

Kommentare werden mit “ (* ” und “ *) ” gekennzeichnet. Zu bemerken ist noch, daß Variablen infolge des sog. Static Binding zwar überschrieben werden können, was den Wert derselben Variable in früher definierten Funktionen aber nicht abändert. Ersichtlich ist dieser Effekt zum Beispiel beim folgenden Programm:

```
        - val pi = 0.0;                (* Neuer Wert ! *)
        val pi = 0.0 : real

        - area (1.0);
        val it = 3.14159 : real     (* !!!! *)
```

3.2.2 Funktionen ohne Namen

In ML gibt es noch eine spezielle Notation für Funktionen: Eine Funktion kann definiert werden, ohne ihr einen Namen zu geben. Die Definition startet mit dem Schlüsselwort *fn* gefolgt von ihren Parametern. Danach folgt das Symbol “=>” und der sog. **body**.

fn `arg => E;`

Anmerkung: für jene die schon vom Lambda-Kalkül gehört haben: die Notation ist ähnlich der Lambda-Notation $\lambda arg.E$.
Es ist auch **Pattern Matching** (siehe 3.5) erlaubt.

fn $\mathtt{P}_1$ `=>` $\mathtt{E}_1$ `|` ... `|` $\mathtt{P}_n$ `=>` $\mathtt{E}_n$;

Zu beachten ist aber, daß keine rekursive Definition möglich ist. Zum Schluß ein kleines Beispiel:

```
- fn x => 3 * x;
val it = fn : int -> int
```

3.2.3 Zahlen, Strings, Wahrheitswerte

Numbers (Zahlenwerte)

ML unterscheidet zwischen ganzen (Integer) und reellen (Real) Zahlen. Integer Zahlen sind zum Beispiel :

$$0,\ \sim 123,\ 01231236253$$

Erlaubte Operationen sind Addition ($+$), Subtraktion ($-$), Multiplikation ($*$), Division (*div*) und Rest (*mod*). Diese sind Infix Operatoren und gehorchen den gewohnten Rechenregeln. Zu bemerken ist, daß negative Zahlen mit der Tilde ($\sim$) gekennzeichnet werden, und nicht wie gewohnt mit einem vorangestellten Minuszeichen.
Real Zahlen werden folgendermaßen angegeben:

$$0.0,\ 2.713242321,\ \sim 1.2E12,\ 7E\sim 5$$

Als Operatoren gelten wieder Addition ($+$), Subtraktion ($-$), Multiplikation ($*$) und Achtung für die Division ($/$). Weitere Funktionen (*sqrt, sin, cos, ...*) siehe Anhang.

Strings

Strings werden von doppelten Anführungszeichen eingeschlossen ("..."). So zum Beispiel:

```
- "Was nun! Eine Ratte! Tot, fuer eine Dukate, tot !"
val it = "Was nun! Eine Ratte! Tot,
          fuer eine Dukate, tot !" : string
```

Dazu gibt es noch Steuerzeichen, welche im Anhang aufgelistet sind. Die wichtigsten Operatoren sind:

- Konkatenation (Concatenation) ($\hat{}$)
- Länge (*size*)
- Umwandlung Ordinalzahl/Charakter (*ord* oder *chr*)

Dazu folgende Beispiele:

```
- "Fair " ^ "Ophelia";
val it = "Fair Ophelia" : string

- size (it);
val it = 12 : int
```

Wahrheitswerte

Bedingte Ausdrücke in SML benehmen sich ähnlich den bedingten Anweisungen in prozeduralen Sprachen. Ein Wahrheitswert ist vom Typ **Bool**.

$$\textit{if}\ \mathrm{E}\ \textit{then}\ \mathrm{E}_1\ \textit{else}\ \mathrm{E}_2$$

Wobei "E" ein Ausdruck sein muß, der einen Wahrheitswert zurückliefert. Ist dieser *true* liefert der Ausdruck den Wert von E_1, ist er aber *false*, so liefert der Ausdruck den Wert von E_2. Der else-Teil muß immer angegeben werden. Als Beispiel sei hier die Signum Funktion genannt:

```
- fun sign (n) =
=         if n > 0 then 1
=    else if n = 0 then 0
=    else ~1;
val sign = fn : int -> int
```

Die einfachsten Tests sind:

- Größer,größer oder gleich als ($>$,$\geq$)
- Kleiner, kleiner oder gleich als ($<$,$\leq$)

Es können aber auch logische Verknüpfungen ausgeführt werden. Dafür stehen das

- Logische or (in ML heißt die Funktion *orelse*)
- Logische und (in ML *andalso*)
- Logische not (*not*)

zur Verfügung. Wichtig ist, daß die Berechnung von *orelse* und *andalso* **lazy** ist. Das bedeutet, ein Ausdruck wird nur berechnet, wenn er benötigt wird. Somit ist

$$\textit{expr}_1\ \textit{orelse}\ \textit{expr}_2$$

äquivalent dem Ausdruck

`if` $expr_1$ `then true else` $expr_2$

und

$$expr_1 \ andalso \ expr_2$$

ist äquivalent dem Ausdruck

`if` $expr_1$ `then` $expr_2$ `else false`

3.3 Paare, Tupel, Records

Aus der Mathematik wissen wir, daß eine Sammlung von Werten oft selbst wie ein Wert betrachtet werden kann. Als Beispiel dafür sei ein Vektor genannt. In ML ist es sehr einfach, derartige Strukturen wie Paare (Tripel, ...) oder Records zu implementieren.

3.3.1 Vektoren

Die Syntax für einen Vektor mit n Elementen ist sehr einfach. Man verwendet dazu die *val* Deklaration wie gewohnt und setzt in die rechte Seite die Elemente des Vektors getrennt durch Beistriche in runde Klammern. Das bedeutet ein n-Vektor hat folgende Form:

$$(x_1, x_2, x_3, \ldots , x_n)$$

Außerdem ist es wichtig zu wissen, daß die Elemente solcher Paare, Tripel,... nicht unbedingt vom gleichen Typ sein müssen. Zur Verwendung von Tupel ein kleines Beispiel:

```
- val zerovec = (0.0, 0.0);
- val a       = (1.5, 6.8);

- fun lengthvec (x,y) = sqrt ( x*x + y*y );
val lengthvec = fn : real * real -> real

- lengthvec a;
val it = 6.9635 : real
```

Das bedeutet, daß *lengthvec* eine Funktion ist, die ein Paar von reellen Zahlen als Argument nimmt, und eine reelle Zahl als Ergebnis zurückliefert.
Es ist noch wichtig zu erwähnen, daß es auch einen 0-Tupel gibt, beschrieben durch zwei runde Klammern (), **unity** ausgesprochen. Er dient als Platzhalter für Situationen, in denen keine Daten übergeben werden müssen. Der 0-Tupel ist ein eigenständiger Wert vom Typ `unit`.

3.3.2 Records

Ein Record ist wie üblich ein Tupel, dessen Elemente (fields) Namen (labels) haben. In ML ist die Syntax ähnlich wie bei den Paaren, nur daß geschwungene Klammern verwendet werden, und zusätzlich die labels angegeben werden müssen:

$$\{ \text{label}_1\text{=field}_1, \text{label}_2\text{=field}_2, ..., \text{label}_n\text{=field}_n \}$$

Um nun bestimmte Elemente anzusprechen, verwendet man die Funktion ♯ (Kanalgitter !) gefolgt von dem Namen des labels. Auch Pattern Matching (siehe 3.5) ist möglich.

```
- val richardIII =
   { name      = "Richard III",
     born      = 1452,
     crowned   = 1483,
     died      = 1485,
     quote     = "Plots have I laid..."};

 - #quote richardIII;
 val it = "Plots have I laid..." : string

 - type king = { name      : string,
                 born      : int,
                 crowned   : int,
                 died      : int,
                 quote     : string
               };

 - fun lifetime ( k : king ) = #died k - #born k;
 val lifetime = fn : king -> int

 - lifetime richardIII;
 val it = 33 : int
```

Die Typenvereinbarung haben wir zwar noch nicht näher besprochen, im folgenden Kapitel werden wir aber noch näher darauf eingehen.

3.4 Operatoren und Typen

3.4.1 Infixoperatoren

Ein Infixoperator ist eine Funktion, die zwischen zwei Argumenten geschrieben wird. Wir kennen diese Schreibweise von der Mathematik her. Ohne Infixoperator müßten wir

$$2 + 2 = 4 \text{ als } = (+(2, 2), 4) \text{ schreiben.}$$

ML bietet uns nun die Möglichkeit, Funktionen in der Infixschreibweise zu benutzen. Dazu gibt man an, welche Funktion in der Infix Notation verwendet werden soll (man verwendet hier die Funktion *infix* oder *infixr*), und definiert daraufhin die neue Funktion. An einem Beispiel ist dies am einfachsten zu verstehen:

```
- infix 4 xor;
- fun p xor q = (p orelse q) andalso not (p andalso q);
val xor = fn : (bool * bool) -> bool
```

Es ist noch anzumerken, daß mit *infix* deklarierte Operatoren linksassoziativ sind, mit *infixr* deklarierte Operatoren sind rechtsassoziativ. Vorrangregeln können dadurch angegeben werden, daß man zwischen *infix* und Funktionsnamen eine Zahl zwischen 0 und 9 angibt, wobei ein Operator mit Präzedenz 6 Vorrang hat vor Operatoren mit der Präzedenz kleiner als sechs. Die Präzedenz muß aber nicht angegeben werden, defaultmäßig ist sie Null. Und so sieht dann der ganze Ausdruck aus:

infix Präzedenz `Funktionsname`

Zum Vergleich: In ML hat die Addition die Präzedenz 6, die Multiplikation die Präzedenz 7, und der Exponent die Präzedenz 8.

3.4.2 Typenvereinbarung

In ML ist es möglich, ähnlich wie in anderen prozeduralen Sprachen, Typenvereinbarungen zu treffen. Ersichtlich am Beispiel mit den Records (type king), verwendet man als Schlüsselwort das Wort *type*. Jene Typen können beliebig weiterverwendet werden, sie bringen mehr Übersichtlichkeit in ihre Funktionen. Weitere Beispiele wären:

```
- type vec = real * real;
- type collection = (real * real) * string;
```

3.4.3 Polymorphismen

Definieren wir zunächst folgende Funktion (*first*), die von einem Paar von Werten (oder Funktionen) das erste Element zurückliefert:

```
- fun first (x,y) = x;
val first = fn : 'a * 'b -> 'a
```

Wir bemerken, daß die Funktion ein Paar von Typen 'a und 'b nimmt, und als Ergebnis einen Wert vom Typ 'a liefert. Diese Typen bezeichnet man als sog. **Typenvariablen**. Sie können für jeden beliebigen Typ (der definiert werden kann) stehen. Eine Funktion, die nun für verschiedene Typen definiert ist, nennt man **polymorphic** beziehungsweise den

Typ, den sie nimmt, **polytype**. SML überprüft bei der Funktionsdefinition selbstständig alle Typen, und es ist üblich, daß immer der allgemeinste Type (eventuell der polytyp) genommen wird. Der einzige Fall, wo man acht geben muß, ist bei der Verwendung von "overloaded" Operatoren (wie zum Beispiel $+, -, *$). Hier muß man sich sehr wohl auf einen bestimmten Typen (int, real) festlegen.

3.5 Rekursive Funktionen

3.5.1 Call by Need, Call by Value

Es sei hier nur erwähnt, daß man bei der Übergabe der Argumente zwischen zwei wichtigen Arten zu unterscheiden hat: *call-by-value* oder `strict-evaluation` und *call-by-need* oder `lazy-evaluation`.

Standard ML of New Jersey verwendet `strict-evaluation`. Auf den genauen Unterschied zwischen den beiden Arten, Verwendung und Bedeutung von `lazy-evaluation` werde ich im 6. Kapitel näher eingehen.

3.5.2 Rekursives Beispiel: ggt

Da Rekursionen in funktionalen Sprachen sehr bedeutend sind, werden wir anhand eines Beispiels, dem größten gemeinsamen Teiler, zeigen, wie rekursive Funktionen in SML definiert werden können. Wir erinnern uns an die Definition des Euklidischen Algorithmus:

```
- fun gcd ( m , n ) =
=     if m = 0 then n
=     else gcd ( n mod m, m);
val gcd = fn : int * int -> int
```

Rekursive Lösungen reduzieren ein Problem in mehrere kleine Unterprobleme, die dann leicher zu lösen sind.
Es gibt aber noch eine zweite Methode, um die Funktion anzuschreiben. Und diese Methode der Fallunterscheidung ist sogar um einiges übersichtlicher:

```
fun gcd ( 0 , n ) = n
|   gcd ( m , n ) = gcd ( n mod m , m );
```

Der senkrechte Strich (|) ist mit einem "Oder" vergleichbar. Die Methode ist inbesondere im Zusammenhang mit **Pattern Matching** sehr nützlich Es ist aber zu beachten, daß bei derartigen Definitionen die Anzahl, der Typ und die Reihenfolge der jeweiligen Argumente übereinstimmen müssen.

3.6 Deklarationen

Denken wir einmal an folgendes Beispiel:

```
- fun fraction ( n,d ) =
= (n div gcd ( n,d ), d div gcd( n,d ));
```

Zur Abkürzung, Übersichtlichkeit, Modularisierung und auch um zu verhindern, daß Funktionen oder Werte bei erneutem Aufruf wieder berechnet werden, hat man in ML nun folgende Möglichkeiten zur Verfügung: **let expressions** und **local expressions**.
Abhängig vom Compiler kann es auch sein, daß let expressions zu effizienteren Programmen führen. Dies ist der Fall wenn der Compiler gemeinsame Unterausdrücke nicht erkennt und daher den gemeinsamen Unterausdruck zweimal auswerten würde. Gute Compiler erkennen jedoch gemeinsame Unterausdrücke und werten diese nur einmal aus.

3.6.1 Let Expression

Eine let expression hat folgende Form:

let `D` *in* `E` *end*;

Während der Ausführung wird zuerst der Ausdruck `D` berechnet und das Ergebnis einem Namen zugewiesen. Dann wird `E` berechnet und an jeder Stelle der Name `D` durch seinen Wert ersetzt.
`D` kann auch ein zusammengesetzter Ausdruck der Form

$$D_1;D_2;...;D_n$$

sein.
Es ist noch wichtig anzumerken, daß die Deklaration von `D` nur innerhalb des *let*-Ausdrucks sichbar ist. Somit ergibt sich für unser obiges Beispiel:

```
- fun fraction ( n,d ) =
=     let val com = gcd ( n,d )
=     in  ( n div com, d div com ) end;
```

3.6.2 Local Expression

Local-Expressions ähneln den Let-Expressions und haben folgende Form:

local $\texttt{D}_1$ *in* $\texttt{D}_2$ *end;*

Diese Form verhält sich wie eine Liste von Deklarationen, wobei aber $\texttt{D}_1$ nur innerhalb von $\texttt{D}_2$ sichtbar ist.

3.6.3 Simultane Deklaration

Eine simultane Deklaration definiert Namen oder Funktionen auf einmal und nicht hintereinander. Dadurch ist es zum Beispiel möglich verschränkt rekursive Funktionen (*mutually recursive functions*) zu programmieren.

Variablen

Eine Variablendeklaration hat dann folgende Form:

$$\textit{val}\ \texttt{Id}_1\ \texttt{=}\ \texttt{E}_1\ \textit{and}\ \ldots\ \textit{and}\ \texttt{Id}_n\ \texttt{=}\ \texttt{E}_n\ ;$$

Zuerst werden die Ausdrücke $\texttt{E}_1, \ldots, \texttt{E}_n$ berechnet und dann erst die Identifier deklariert. Sehr anschaulich ist folgendes Beispiel:

```
- val one = "Bong";
- val three = " Bong Bong Bong";

- val one = three
= and three = one;
val one = "Bong Bong Bong" : string
val three = "Bong" : string
```

Funktionen

Viele Funktionen sind *mutual recursive*. Das bedeutet, daß sie einander gegenseitig rekursiv definieren können. Man könnte nun leicht auf Probleme stoßen, wenn man überlegt, welche Funktion man als erste angeben sollte. In SML hat nun eine "gleichzeitige" Funktionsdefinition folgende Form:

$$\begin{aligned}&\textit{fun}\ \texttt{name}_1\ \texttt{arg}_1\ \texttt{=}\ \texttt{Expr}_1\\ &\textit{and}\ \texttt{name}_2\ \texttt{arg}_2\ \texttt{=}\ \texttt{Expr}_2\\ &\qquad\ldots\ \texttt{;}\end{aligned}$$

Es können beliebig viele Funktionen gleichzeitig deklariert werden, wobei zu beachten ist, daß nach der Definition einer Funktion kein Strichpunkt gesetzt werden darf. Erst am Ende der ganzen Deklaration ist ein Strichpunkt zu setzen. Dazu ein kleines, wenn auch ineffizientes Beispiel. Das Erkennen einer geraden oder ungeraden Zahl:

```
- fun even 0 = true
= |   even n = odd (n-1)
= and  odd 0 = false
= |    odd n = even(n-1);
fun even : int -> bool
fun odd  : int -> bool
```

Kapitel 4

Listen, eine Einführung

4.1 Allgemeines

Eine Liste ist eine endliche Folge von Elementen. Typische Beispiele von Listen sind die leere Liste [], [1,2,4] und [“Werner”,”Andreas”,”Brigitte”]. Wie wir sehen, ist eine Liste in ML dadurch gekennzeichnet, daß sie von eckigen Klammern begrenzt wird. Die Elemente einer Liste müssen vom gleichen Typ sein, der sich jedoch aus jedem beliebigen anderen Typ zusammensetzen kann. Dazu gehören auch Listen und Records wie zum Beispiel [(1,”one”),(2,”two”),(3,”three”)] oder gar Funktionen. Im folgenden wollen wir uns mit dem Aufbau von Listen und deren Anwendungen anhand von einfachen Beispielen auseinandersetzen.

4.2 Aufbau einer Liste

Der Aufbau einer Liste ist sehr einfach und besteht im wesentlichen aus nur zwei Dingen: Der Konstanten *nil*, ein Synonym für die **leere Liste** [] und dem Infixoperator *::* (2 Doppelpunkte hintereinander), ausgesprochen *cons* für **construct**. Mit diesem Operator wird ein Element an den Beginn einer Liste angeführt:

head :: tail

Hier ist *head* ein Element mit dem selben Typ, wie die Elemente in der Liste, und *tail* die Liste, zu der das Element hinzugefügt werden soll.
Um zum Beispiel eine aufsteigende Folge von ganzen Zahlen in einer Liste zu erzeugen, schreiben wir ein kurzes Programm:

```
- fun upto (m,n) =
=   if m > n then []
=            else m :: upto (m+1,n);
val upto = fn : int * int -> int list
```

```
- upto (2,5);
val it = [2,3,4,5] : int list
```

Wir sehen, daß diese kurze Funktion ein Paar von Integerwerten als Argument nimmt und eine Integerliste zurückliefert. Das geschieht dadurch, daß jeweils das erste Element an den Rest der verbleibenden (und noch zu konstruierenden) Liste angehängt wird.

4.3 Arbeiten mit Listen

So einfach wie der Aufbau der Listen, so ist auch das Arbeiten mit ihnen. Im folgenden werde ich die wichtigsten Begriffe und Operatoren im Zusammenhang mit Listen erklären. Abschließend sei ein kleines Beispiel angeführt.

4.3.1 Zugriff auf Elemente

Nachdem wir gesehen haben, wie eine Liste aufgebaut wird, wollen wir natürlich wissen, wie wir auf jedes einzelne Element der Liste zugreifen können.
Es gibt nun zuerst die Möglichkeit, jedes Element einer Liste explizit zu nennen. Dazu werden einfach die Listenelemente zwischen 2 eckige Klammern gesetzt und voneinander durch Beistriche getrennt. Gut ersichtlich an einem kleinen Beispiel:

```
- fun prod_of_3 [i,j,k] : int = i * j * k;
val prod_of_3 = fn : int list -> int

*** Warning: Match nonexhaustive
             i :: j :: k :: nil => ...
```

Die Angabe des Typs Integer hinter der Klammer ist notwendig, damit ML weiß um welche Funktion es sich bei der Multiplikation wirklich handelt (real oder integer). Probieren Sie ruhig einmal aus, was passiert, wenn Sie die Typenangabe weglassen.
Zu beachten ist auch die Warnung am Ende. ML erkennt, daß in der Funktionsdefinition für die leere Liste oder Listen mit einer Elementanzahl ungleich drei keine Definition vorgesehen ist. (Ist es klar, daß keine anderen Funktionsaufrufe möglich sind, so kann diese Warnung ignoriert werden.)

Was aber, wenn wir nicht wissen, wieviele Elemente es in einer Liste gibt. Hier steht uns eine einfache, sehr verständliche und gut einsetzbare Schreibweise zur Verfügung:

```
- fun prod []       = 1
=   |  prod (n::ns)  = n * (prod ns);
val prod = fn : int list -> int
```

Wie wir sehen, benutzen wir den *cons* - Operator dazu, die Liste in ein Head-Element und einer Restliste (Tail) aufzuspalten. Somit kann einfach auf das erste Element der Liste zugegriffen werden.
Da ein mögliches Resultat eine 1 (Integer) sein kann, ist hier die Angabe des Typs unnötig. Auch werden alle möglichen Fallunterscheidungen getroffen, und somit entfällt die Warnung des Systems.

Nun noch zwei Funktionen, die wir später im Sortierbeispiel brauchen werden. Das ist zunächst eine Funktion, die uns die ersten i Elemente einer Liste liefert (*take*), und eine Funktion, die eine Liste liefert, bei der die ersten i Elemente entfernt wurden (*drop*).

```
- fun take ( i, [] )       = []
=   |  take ( i, x::xs )    = if i > 0 then x :: take (i-1,xs)
                                       else [];

- fun drop ( i, [] )       = []
=   |  drop ( i, x::xs )    = if i > 0 then drop (i-1,xs)
                                          else x::xs;
```

4.3.2 Operatoren und Funktionen

Nun werden die wichtigsten Operatoren und Funktionen, die im Grundpaket von SML für Listen enthalten sind vorgestellt:

- Um das erste Element einer Liste anzusprechen, verwendet man die Funktion *hd* (für head). Es ist zu beachten, daß diese Funktion nicht bei der leeren Liste anwendbar ist.
- Der Rest der Liste wird mit *tl* zurückgeliefert. Auch hier ist diese Funktion nur auf nicht leere Listen anwendbar.
- Der Infixoperator @ (append) hängt eine Liste an eine andere an. Er ist folgendermaßen definiert:

$$[x_1,...,x_n] \ @ \ [y_1,...,y_m] = [x_1,...,x_n,y_1,...,y_m]$$

- Die Funktion *rev* (für revers) ändert die Reihenfolge einer beliebigen Liste, d.h es wird die reverse Liste zurückgeliefert.
- Die Funktion *null* liefert true, falls eine Liste leer ist, sonst false.

4.4 Ein Beispiel: Mergesort

Ich gehe davon aus, daß jeder den Algorithmus kennt. Außerdem ist die Definition des Algorithmus durch die Definition in ML sehr gut ersichtlich.
Zunächst brauchen wir eine Funktion, die 2 sortierte Listen zu einer verschmilzt, indem sie jeweils das kleinere Element der beiden Listen nimmt und an die erste Stelle der beiden Listen stellt. Dieser Vorgang wiederholt sich dann rekursiv.

```
- fun merge ( [], ys )         = ys : real list
=  |  merge ( xs, [] )         = xs
=  |  merge ( x::xs,y::ys)     =
=       if x <= y then x :: merge ( xs, y::ys )
=                 else y :: merge ( x::xs, ys );
```

Nun gibt es zwei Methoden, um eine Liste mit Mergesort zu sortieren. Die **top-down** oder die **bottom up Methode**. Bei der ersteren wird die Liste immer in zwei (ungefähr) gleich große Teile zerlegt und diese rekursiv weitersortiert. Bei der zweiten geht man von n Listen der Länge 1 aus. Hier sei das **top-down Mergesort** angeführt:

```
- fun mergesort []    = []
=  |  mergesort [x]   = [x]
=  |  mergesort xs    =
=       let val k = length xs div 2
=       in
=          merge ( mergesort(take(k,xs)),
=                  mergesort(drop(k,xs)))
=       end;
```

Diese Funktion braucht zum Sortieren von 10000 Zahlen ungefähr sechs Sekunden auf unserem Institutsrechner, wobei die Rekursionstiefe bereits sehr groß ist. Auf einem schnellen 486 PC schafft es uML immerhin 450 Zahlen in 6 Sekunden zu sortieren.

Kapitel 5

Funktionen höherer Ordnung

Die große Ausdrucksfähigkeit von funktionalen Programmiersprachen liegt unter anderem in der Möglichkeit, Funktionen als Argumente zu übergeben, oder Funktionen als Ergebnis von Funktionen zu liefern. Derartige Funktionen nennt man **Funktionen höherer Ordnung**. Sie machen es möglich, eine sehr allgemeine Funktion zu schreiben, die dann in einer Vielzahl von Anwendungen gebraucht werden kann. In den folgenden Kapiteln werden nun die Grundzüge von Funktionen höherer Ordnung und einige Beispiele näher vorgestellt. Eine gute Erklärung ist in [Wik90], Kapitel 15 zu finden.

5.1 Grundlagen

5.1.1 Partielle Funktionen

In ML nimmt eine Funktion ohne Ausnahme immer nur ein Argument. Wenn mehrere Argumente zu übergeben sind, dann müssen sie als Paare, Tripel usw. übergeben werden. Betrachten wir nun folgende Funktion:

```
- fun plus (x,y):int = x + y;
val plus = fn : int * int -> int
```

Die Funktion *plus* übernimmt zwei Argumente, wir wissen aber, daß sie in Wirklichkeit als Paar übergeben werden. Man kann jedoch eine Funktion definieren, die jeweils ein Argument nach dem anderen übernimmt. Dazu läßt man die runden Klammern und die Kommazeichen weg. Unsere Funktion sieht dann folgendermaßen aus:

```
- fun add x y : int = x + y;
val add = fn : int -> int -> int
```

Eine derartige Funktion wird auch oft als **curried function** oder **partially applicable function**

bezeichnet (benannt nach dem Logiker H.B. Curry, erfunden aber vom Logiker M. Schönfinkel). In unserem Fall ist die Funktion *add* die "curried function" der Funktion *plus*. Sie ist eine Funktion, die, wenn sie ein Argument übernimmt, eine genauer spezifizierte Funktion zurückliefert. Im speziellen bedeutet das, daß *add* eine Integerzahl als Argument nimmt und eine Funktion von Integer auf Integer zurückliefert. Das bedeutet, man könnte den Typ von *add* auch folgendermaßen anschreiben: `int -> ( int -> int )`.

Diese Art von Funktion können wir nun auch anders verwenden als gewohnt:

```
- (add 10) 5;
val it = 15 : int

- val add10 = add 10;
val add10 = fn : int -> int

- add10 5;
val it = 15 : int
```

Wie wir sehen, wird der entgültige Wert des Ausdrucks erst berechnet, wenn das letzte Argument übergeben worden ist.
Auch die Umwandlung einer Funktion von zwei Argumenten in die entsprechende "curried" Version der Funktion und umgekehrt, ist in ML ausdrückbar. Die Funktion *curry* ist dann wie folgt definiert:

```
- fun curry f = fn x => fn y => f(x,y);
val curry = fn :
    ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Die Transformation einer "curried function" (mit zwei Argumenten) in eine "uncurried function" besorgt folgende Funktion:

```
- fun uncurry f = fn (x,y) => f x y;
val uncurry = fn :
    ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

Damit können wir zum Beispiel die Funktion *take* aus dem vorigen Kapitel in eine "curried function" verwandeln:

```
- val take_c = curry take;
val take_c = int -> 'a list -> 'a list

- take_c 2;
```

```
val it = fn : 'a list -> 'a list

- it [1,2,3,4,5];
val it = [1,2]
```

5.1.2 Funktionen als Argumente

Wir haben nun eine Funktion kennengelernt, die wiederum eine Funktion als Ergebnis zurückliefert. In vielen Anwendungsfällen ist es aber auch von Vorteil, Funktionen als Argumente übergeben zu können. Zur Demonstration wollen wir eine einfache Funktion (*square*) betrachten, die dann als Argument einer zweiten Funktion (*twice*) übergeben wird.

```
- fun square x : int = x * x;
val square = fn : int -> int

- fun twice f x = f(f x);
val twice = fn : ('a -> 'a) -> 'a - 'a

- twice square 5;
val it = 625
```

Und das bewirkt die *twice*-Funktion:

twice square **5** –> *square(square* **5***)* –> *square(***25***)* –> **625**

Die Funktion *twice* ist selbst eine **curried function** und man kann ihr somit auch eine Funktion ohne Argument zuweisen, was natürlich eine genauer spezifizierte Funktion zurückliefert. So zum Beispiel:

```
- val power4 = twice square;
val power4 = fn : int -> int;
```

Als Schlüsselwort muß hier *val* verwendet werden, da ja keine neuen Parameter dazukommen. Einer Funktion (auf der rechten Seite) wird nur ein neuer Name gegeben (den auf der linken Seite des Gleichheitszeichens).
Im folgenden wollen wir uns mit einer für Listen sehr bedeutenden Funktion auseinandersetzen:

5.1.3 Map, eine wichtige Funktion

Listen werden häufig in funktionalen Programmen verwendet. Viele Funktionen, die mit Listen arbeiten, sind Funktionen höherer Ordnung; die vermutlich wichtigste unter ihnen ist die *map*-Funktion. Sie nimmt eine Funktion und eine Liste als Argumente und wendet die Funktion auf alle Elemente der Liste an. Um zum Beispiel alle Elemente eine Liste zu quadrieren, genügt folgender Aufruf:

```
- map square [1,3,5];
val it = [1,9,25] : int list
```

Die *map*-Funktion ist in SML schon definiert, aber es sollte keine Schwierigkeiten machen, sie nocheinmal zu definieren:

```
- fun map f nil = nil
>   |  map f (x::xs) = f x :: map f xs;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

5.1.4 Functional Composition (Einsetzungsoperator)

Functional composition wird in SML durch ein kleines "o" (auch Compose Operator) ausgedrückt. Die *compose*-Funktion nimmt zwei Funktionen als Operanden und liefert eine neue Funktion als Argument zurück. Der Operator ist durch folgende Gleichung definiert:

$$(\mathrm{f\ o\ g})\mathrm{x} = \mathrm{f}\ (\mathrm{g\ x})$$

Mit Hilfe des Operators können wir die *twice*-Funktion nochmals definieren. Auch ist der Operator nützlich, um das letzte Element einer Liste (Funktion *last*, siehe unten) zu bekommen. Es sind *hd* und *rev* bereits bekannte Funktionen. *hd* liefert das erste Element einer Liste zurück, und *rev* dreht die Reihenfolge einer Liste um.

```
- fun twice f = f o f;
val twice = fn : ('a -> 'a) -> 'a -> 'a

- val last = hd o rev;
val last = fn : 'a list -> 'a
```

5.2 Anwendungsbeispiel

In ML ist es sehr einfach, mit Hilfe von Funktionen höherer Ordnung mathematische Ausdrücke zu formulieren. Ich möchte nun ein kleines Beispiel erarbeiten, um zuerst die Sigma Funktion und mit ihrer Hilfe das bestimmte Integral einer beliebigen Funktion zu berechnen.

Für die Summation von einer Folge von Zahlen haben wir eine spezielle Notation eingeführt:

$$\sum_{i=n}^{m} x_i \;=\; x_n \;+\; x_{n+1} \;+ \ldots +\; x_m$$

Um eine ähnliche Notation auch in SML zu erreichen, muß nicht viel getan werden. Wir benötigen nur zwei weitere einfache Funktionen. Die eine generiert eine aufsteigende Folge von Zahlen ($--$, infix), und die zweite Funktion bildet die Summe über eine Liste von Integer (*sum'i*) oder Realwerten (*sum'r*).

```
- infix --;
- fun lower -- upper =
> if lower > upper then []
             else lower :: ( (lower+1) -- upper )

- fun sum'i  nil  =  0
> |   sum'i  x    = hd x + sum'i (tl x);

- fun sum'r  nil  = 0.0
> | sum'r  x      = hd x + sum'r (tl x);
```

Nun können wir die Sigma Funktion definieren. Dabei übergeben wir der Funktion drei Argumente: eine Funktion, über die die Summe gebildet werden soll (*term*) und die untere (*lower*) und obere (*upper*) Grenze.

```
- fun sigma'i term lower upper =
>   sum'i ( map term (lower -- upper));

- fun sigma'r term lower upper =
>   sum'r ( map term (lower -- upper));
```

Es wurden 2 Funktionen definiert, eine für Integer, die andere für Realwerte. Der Definition ist jedoch völlig gleich. Die Funktion $--$ erzeugt eine Liste aufsteigender Integerwerte. Die map-Funktion sorgt nun dafür, daß der Term auf jedes Element der Liste angewendet wird. Zum Schluß wird dann noch über die Liste die Summe gebildet.

Nun ist es einfach, den Ausdruck $\sum_{i=1}^{100} x^2$ zu berechnen:

```
- sigma'i (fn x => x*x) 1 100;
val it = 338350
```

Die Summation über eine unendliche Anzahl von Termen kann natürlich nur annähernd berechnet werden. Um zum Beispiel die Basis e des natürlichen Logarithmus zu berechnen bedienen wir uns seiner Definition: $\sum_{i=0}^{\infty} \frac{1}{i!}$. Da wir wissen, daß die Summe konvergiert, können wir e annähernd berechnen.

```
- fun invert x = 1.0 / x;

- fun fact 0  = 1
> |   fact x  = x * fact (x-1);

- sigma'r ( invert o real o fact ) 0 12
val it = 2.718281
```

Wie wir sehen, ein wirklich schönes Beispiel Funktionen höherer Ordnung.

Ähnlich kann man auch beim bestimmten Integral vorgehen. Für das Integral

$$\int_a^b f(x)\ dx$$

gilt ja folgende folgende Näherungsformel:

$$h * (\tfrac{f(a)+f(b)}{2} + \textstyle\sum_{i=1}^{n-1} f(a + i * h)) \text{ mit } h \ = \ \tfrac{b-a}{n}$$

In SML definieren wir nun die Funktion *integrate* und versuchen gleich ein Beispiel:

```
- fun integrate f a b n =
>     let val h = (b-a) / real n
>     in h * ((f a + f b ) / 2.0 +
>        sigma'r (fn i => f(a + real i * h )) 1 (n-1))
>     end;

- integrate (fn x => x * x * x) 0.0 2.0 10
val it = 4.04 : real
```

Kapitel 6

Lazy Evaluation

In den vorhergehenden Kapiteln sind Ausdrücke wie *strict evaluation, lazy evaluation, call by need* usw. verwendet worden. Ich möchte nun versuchen, deren Bedeutung näher zu erklären. Anschließend implementieren wir eine unendlich lange Liste und zeigen deren Anwendung anhand eines einfachen Beispiels. Siehe auch [L.C91], Kapitel 5 und [Wik90], Kapitel 6.3.

6.1 Allgemeines

Beschäftigen wir uns ganz kurz mit dem Berechnen von Ausdrücken. Wenn ein Ausdruck *f(E)* berechnet wird, dann muß das Argument *E* dem sogenannten *body* der Funktion *f* übergeben werden. Man unterscheidet **"lazy languages"** und **"strict languages"**, je nachdem ob das Argument unausgewertet oder bereits ausgewertet übergeben wird. "lazy languages" sind allgemeiner aber schwieriger effizient zu implementieren. "strict languages" sind weniger allgemein aber einfacher effizient zu implementieren. ML gehört zur Klasse der "strict languages". Die Parameterweitergabe im letzteren Fall ist auch unter dem Namen "call by value" bekannt. Die bei "lazy languages" verwendete Methode der Parameterweitergabe ist ähnlich dem "call by name"; es sind bei funktionalen Sprachen auch noch Bezeichnungen wie "call by need", "lazy evaluation" und "normal order reduction" üblich. Die genauen und subtilen Unterschiede in der Bedeutung dieser Konzepte werden im folgenden und vor allem auch in der Vorlesung im Detail ausgeführt.

Grundsätzlich gib es folgenden Unterschied:

- Bei "call by value" wird der Operand ausgerechnet, bevor der "body" der Funktion berechnet wird.
- Bei der "normal order reduction" aber wird der Wert des Operanden nur berechnet, wenn er bei der Berechnung des "bodies" der Funktion benötigt wird.

6.1.1 Call by value

Um den Unterschied zwischen den beiden Regeln studieren zu können, betrachten wir folgende zwei Funktionen. Die eine (*sqr*) berechnet das Quadrat der übergebenen Integerzahl, die zweite Funktion (*zero*) nimmt eine Integerzahl als Argument, ignoriert aber deren Wert und liefert immer Null als Ergebnis.

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

- fun zero(x : int) = 0;
val zero = fn : int -> int
```

ML berechnet den Ausdruck *sqr(sqr(sqr(2)))* in den folgenden Schritten:

```
sqr(sqr(sqr(2))) => sqr(sqr(2 * 2))
                 => sqr(sqr(4))
                 => sqr(4 * 4)
                 => sqr(16)
                 => 16 * 16
                 => 256
```

Nun schauen wir aber, wie der Ausdruck *zero(sqr(sqr(sqr(2))))* berechnet wird.

```
zero(sqr(sqr(sqr(2)))) => zero(sqr(sqr(2 * 2)))
                       => zero(sqr(sqr(4)))
                       .
                       .
                       .
                       => zero(256)
                       => 0
```

Wir erkennen sofort den Nachteil der strikten Berechnung. Es wäre doch viel ökonomischer, den Ausdruck gleich auf Null zu setzen. Es muß aber erwähnt werden, daß "call by value" keineswegs immer ineffizienter ist als "call by name".
Es gibt auch einen entscheidenden semantischen Unterschied, der sich im Resultat und nicht nur in der Effizienz zeigt. Wenn wir der Funktion *zero* ein Argument übergeben dessen Auswertung nicht terminiert, würde "call by value" nicht terminieren, während "call by name" das richtige Resultat 0 liefern würde. In diesem Sinne ist "call by name" allgemeiner als "call by value".

6.1.2 Lazy Evaluation (Call by name, Call by need)

Call by name und Call by need sind sich zwar recht ähnlich, es gibt aber, wie wir gleich sehen werden, einen entscheidenden Nachteil bei Call by name.

Call by name

Um Ausdrücke wie den obigen intelligenter zu berechnen, gehen wir nach folgender Berechnungsvorschrift vor:

Um den Wert von f(E) zu berechnen, setzen wir E sofort in den "body" der Funktion f ein. Dann berechnen wir den Wert des verbleibenden Ausdrucks.

Diese Vorschrift reduziert den Ausdruck *zero(sqr(sqr(sqr(2))))* in einem Schritt zu Null. Sie hat aber einen entscheidenden Nachteil: diese Berechnungsvorschrift funktioniert sehr schlecht, wenn die äußere Funktion *strict* ist (d.h. alle Argumente gebraucht werden und daher schon bei der Übergabe berechnet werden können). Strikte Funktionen sind alle Operatoren wie Addition, Multiplikation und so weiter. "Call by name" hat nun zur Folge, daß z.B. der Ausdruck *sqr(sqr(sqr(2)))* "reduziert" wird zu sqr(sqr(2)) $*$ sqr(sqr(2)), da sqr(x) = x $*$ x ist. In unserem Fall wäre dann:

```
sqr(sqr(sqr(2))) => sqr(sqr(2)) * sqr(sqr(2))
                 => (sqr(2) * sqr(2)) * sqr(sqr(2))
                 => ((2 * 2) * sqr(2)) * sqr(sqr(2))
                 .
                 .
                 .
```

Wie wir sehen, wird das Ergebnis erst nach einer sehr langen Folge von Schritten geliefert. Diese Berechnungsvorschrift kann nicht unsere gewünschte sein.

Call by need

Diese Vorschrift ist semantisch gleich der Berechnungsregel bei Call by name, es wird jedoch sichergestellt, daß jedes Argument nur einmal berechnet werden muß. Beim Auftreten eines Ausdrucks wird dieser in einer Pointerstruktur vermerkt und somit sichergestellt, daß, wird der Ausdruck jemals berechnet, das Ergebnis von den anderen Ausdrücken verwendet werden kann. In Worten schwierig zu erklären, aber anhand unseres Beispiels (siehe unten) sehr gut ersichtlich.

Die Pointerstruktur ist ein gerichteter Graph, der aus Funktionen und Argumenten besteht. Wird ein Teil des Graphen berechnet, so wird dieser Teil durch dessen Ergebnis ersetzt. Noch eine kurze Anmerkung zur folgenden Schreibweise: um sich eine Graphendarstellung zu ersparen, zeigt der Ausdruck [x = e] an, daß alle Vorkommnisse von x sich den Wert E teilen:

```
sqr(sqr(sqr(2))) => x * x [x = sqr(sqr(2))]
                 => x * x [x = y * y] [y = sqr(2)]
                 => x * x [x = y * y] [y = 2 * 2 ]
```

```
=> x * x [x = 4 * 4]
=> 16 * 16
=> 256
```

Wir sehen, daß bei dem Beispiel keinerlei Probleme mehr auftreten. Auch wird der Ausdruck *zero(sqr(sqr(sqr(2))))* unmittelbar auf Null gesetzt.

Die Effizienz ist ein entscheidendes Problem. Es ist möglich, mit Lazy Evaluation viel Platz zu sparen, oder auch zu verschwenden. Weiters besagt diese Regel, daß der Ausdruck *zero(E)* Null ist, egal ob E terminiert oder nicht.

Lazy Evaluation ist zu diesem Zeitpunkt sehr wichtig in der Forschung.

6.2 Lazy Evaluation in SML

In ML werden alle Operanden berechnet, ganz egal ob sie gebraucht werden oder nicht. Die Ausnahmen sind boolsche (*andalso, orelse*) und bedingte Ausdrücke (*if-then-else*). Wir haben dies bereits im Kapitel 3.2 kennengelernt. Die Funktionen *andalso* und *orelse* haben die Eigenschaft, daß ihre Operanden nur berechnet werden, wenn sie logisch gebrauch werden. Ähnlich ist es auch bei den bedingten Ausdrücken. Hier wird entweder der *then* Zweig oder der *else* Zweig berechnet, aber nie beide Zweige. Diese Berechnungsvorschrift ist ein Spezialfall der allgemeinen Strategie, *lazy evaluation* genannt.

6.3 Lazy Lists

Die Elemente einer "Lazy List" werden erst dann berechnet, wenn deren Werte vom Rest des Programms benötigt werden. Deshalb dürfen "Lazy Lists" auch unendlich sein.

In diesem Kapitel wird beschrieben, wie man in ML eine "Lazy List" implementieren kann, indem man den *tail* der Liste durch eine Funktion darstellt, und somit deren Berechnung verzögert.

6.3.1 Sequences

In ML nennt man unendliche Listen **sequences**. Wie eine Liste können sie entweder leer sein, oder aus einem *head* und einem *tail* bestehen. Die leere Liste ist `Nil` und die nicht leere Liste hat die Form `Cons(x,xf)`. Es ist `x` der *head* der Liste und `xf` eine Funktion um den *tail* zu berechnen.

Wir führen nun einen neuen Datentypen ein. Die Erklärung der Notation ist sehr umfangreich, und wird in der Vorlesung näher besprochen. Eine gute Einführung ist auch in [L.C91], p.104 − 113 zu finden.

```
- datatype 'a seq = Nil
>                | Cons of 'a * (unit -> 'a seq);
```

Um damit arbeiten zu können, definieren wir uns zunächst drei Funktionen. Die erste (*head*) liefert uns das erste Element der Liste, die zweite (*tail*) entfernt das erste Element und liefert uns den *tail* der Liste. Die dritte Funktion kombiniert ein *head* Element und einen *tail* zu einer längeren Liste:

```
- fun head  (Cons(x, _ )) = x;
- fun tail  (Cons(x, xf)) = xf ();
- fun consq (x, xq)       = Cons(x, fn () => xq);
```

Die *tail* Funktion wird auf den Universalwert "unity" () angewendet, um die Berechnung der Liste zu erzwingen. (Es wird dadurch ein Paar zurückgeliefert, daß aus einem Head-Element und einer Funktion für den Tail besteht.)

Es ist sehr wichtig zu wissen, daß *Cons(x,E)* nicht "lazy" berechnet wird. Um eine verzögerte Berechnung zu erreichen, muß man *Cons(x, fn()=>E)* schreiben. Der Gebrauch von *fn* sorgt für eine Berechnung nach der Vorschrift call-by-name.

Sehen wir uns nun ein kleines Beispiel an:

```
- fun from k = Cons(k, fn()=> from (k+1));
val from = fn : int -> int seq

- from 1;
val it = Cons(1,fn) : int seq

- tail it;
val it = Cons(2,fn) : int seq

- tail it;
val it = Cons(3,fn) : int seq

- head it;
val it = 3 : int
```

Um nun die ersten n Elemente einer unendlichen Liste zu bekommen genügt folgende Funktion:

```
- fun takeq (0, xq)         = []
> |   takeq (n, Nil)        = []
> |   takeq (n, Cons(x,xf)) = x :: takeq (n-1, xf());

- takeq (5, from 30);
val it = [30,31,32,33,34] : int list
```

Auch Funktionen über unendliche Listen sind einfach zu definieren, insoferne sie eine endlich Anzahl von Inputs brauchen. So zum Beispiel die Funktion *squares* die die Elemente einer Liste quadriert:

```
- fun squares Nil : int seq = Nil
> |   squares (Cons(x,xf))  = Cons(x*x, fn()=>squares (xf()));
val squares = fn : int seq -> int seq

- squares (from 1);
val it = Cons(1,fn) : int seq

- takeq (5, it);
val it = [1,4,9,16,25] : int list
```

6.3.2 Sieve of Eratosthenes

Eine gute Methode, um Primzahlen zu finden, ist der Algorithmus von Eratosthenes [L.C91], p.171:

- Man beginnt mit der Folge [2,3,4,5,6,..].
- Man nimmt 2 als Primzahl und löscht alle Vielfachen von 2. Das liefert eine Folge [3,5,7,9,11,..].
- Nun nimmt man 3 als Primzahl und löscht dessen Vielfache. Man erhält als Liste [5,7,11,13,17,..].
- Man nimmt 5 als Primzahl ...

Wir brauchen somit eine Funktion, die die Vielfache einer Sequence herauslöscht. Diese Funktion (*sift*) benutzt eine Funktion *filterq*, um Elemente aus einer Sequence herauszufiltern. *filterq* ist eine "curried-Function", die eine Funktion (*pred*) und eine Liste als Argument übernimmt.

```
- fun filterq pred Nil           = Nil
> |   filterq pred (Cons(x,xf))  =
```

```
>             if pred x then Cons(x,fn()=>filterq pred (xf()))
>                       else filterq pred (xf());

- filterq (fn n => n mod 10 = 7) (from 50);
val it = Cons(57,fn) : int seq

- takeq (5, it);
val it = [57,67,77,87,97] : int list
```

Nun können wir die Funktion *sift* definieren.

```
- fun sift p = filterq( fn n => n mod p <> 0);
val sift = fn : int -> int seq -> int seq
```

Mit ihrer Hilfe bilden wir nun die Funktion *sieve*, die nach dem Algorithmus von Eratosthenes, laufend die Funktion *sift* auf die unendliche Liste anwendet und somit eine Liste aufbaut, in der alle Vielfachen der Primzahlen gelöscht worden sind.
Im Anschluß daran wird die Funktion gleich ausprobiert und das Ergebnis in einer unendlich langen Liste, die nur aus Primzahlen besteht (*primlist*), abgespeichert. Deren ersten 10 Elemente sehen wir mit der Funktion *takeq* an.

```
- fun sieve (Cons(p,nf)) = Cons(p,fn()=>sieve(sift p (nf())));
val sieve = fn : int seq -> int seq

- val primlist = sieve (from 2);
val primlist = Cons(2, fn) : int seq

- takeq (10, primlist);
val it = [2,3,5,7,11,13,17,19,23,29]
```

Kapitel 7

Modulare Programmierung

Modulare Programmierung ist ein Versuch, komplexere Software menschlich bewältigbar zu halten. Es wird dabei das Programm hierarchisch in kleinere Teile zerlegt, und diese werden dann durch möglichst einfache Interfaces miteinander verbunden. In Standard ML kann ein Programm oder System durch **Module** organisiert werden. So ein Modul besteht entweder aus einer Struktur (**structure**) oder einem sogenannten **functor**.
Eine Struktur ist eine Sammlung von Deklarationen (Typen, Werte und andere Strukturen). Eine **Signatur** besteht aus der Typeninformation über jedes Item, das in der Struktur definiert wurde. Es werden die Werte, die Typen, die Substrukturen und deren Signaturen aufgelistet. Ein **functor** ist ein "mapping" von einer Struktur auf eine andere Struktur. Der "body" definiert die Struktur mittels formaler Parameter, welche in der Signatur definiert worden sind. Dadurch wird getrennte Compilierung ermöglicht. Dieses Kapitel soll nur den Überblick über Module vermitteln, nähere Informationen zu diesem Thema finden Sie in den Büchern, die im Literaturverzeichnis angegeben sind. Die Details werden auch in der Vorlesung genauer besprochen.

7.1 Structures

Für die weitere Betrachtung sehen wir uns einmal das Beispiel einer FIFO-Queue an. Folgende Operationen sollten ermöglicht werden.

- *empty* - die leere Queue,
- *enqueue(q,x)* - hinzufügen eines Elements zu einer Queue,
- *null(q)* - testen, ob die Queue leer ist,
- *hd(q)* - das erste Element der Queue,
- *deq(q)* - entfernen eines Elements von der Queue,
- *Qerror* - Fehlermeldung, wenn auf leere Queue zugegriffen werden soll (hd, dequeue).

Wenn wir die Queue als eine Liste auffassen, dann könnte unsere Struktur folgendermaßen aussehen:

```
- structure Queue =
>   struct
>   type 'a T = 'a list;
>   exception E;
>   val empty       = [];
>   fun enq (q,x)   = q @ [x];
>   fun null (x::q) = false
>     | null _      = true;
>   fun hd (x::q)   = x
>     | hd []       = raise E;
>   fun deq (x::q)  = q
>     | deq []      = raise E;
> end;
```

Der Aufruf kann folgendermaßen erfolgen:

```
- Queue.deq ["Now","I","know","all","about","structures"];
val it = ["I","know","all","about","structures"] : string list
```

Bei vielen verschachtelten Strukturen kann ein Aufruf ziemlich lange werden. Nun ist es einfacher, sich durch eine *open* Deklaration, eine Struktur zu öffnen. Das bedeutet, der Aufruf *Queue.deq ["We","are","happy"]* vereinfacht sich zu *deq ["We","are","happy"]*, wenn man vorher die Struktur mit **open** *Queue* geöffnet hat.

7.2 Signatures

Im Sinne der Abstraktion ist es möglich, für jede Struktur eine Signatur anzugeben. In unserem Fall hat die Signatur für die Struktur *Queue* folgendes Aussehen:

```
- signature Queue =
>   sig
>     type 'a T
>     exception E
>     val empty  : 'a list
>     val enq    : 'a list * 'a -> 'a list
>     val null   : 'a list -> bool
>     val hd     : 'a list -> 'a
>     val deq    : 'a list -> 'a list
>   end;
```

Wenn wir eine Struktur für einen Baum (*tree*) entwickeln wollen, dann können wir folgendermaßen vorgehen. Zuerst definieren wir uns die Signatur, um den Inhalt der Struktur zu dokumentieren.

```
- signature TREE =
>   sig
>     datatype 'a T = Lf | Br of 'a * 'a T * 'a T
>     val count   : 'a T -> int
>     val deph    : 'a T -> int
>     val reflect : 'a T -> 'a T
>   end;
```

Nachdem nun die Struktur definiert ist (mit der ihr auferlegten Signatur *TREE*), überprüft ML, ob Signatur und Struktur zusammenpassen.

```
- structure Tree : TREE =
>   struct
>     datatype 'a T = Lf | Br of 'a * 'a T * 'a T;
>     fun count Lf              = 0
>       | count (Br(v,t1,t2))   = 1 + count t1 + count t2;
>     fun depth Lf              = 0
>       | depth (Br(v,t1,t2))   = 1 + maxl[depth t1, depth t2];
>     fun reflect Lf            = Lf
>       | reflect (Br(c,t1,t2)) = Br(v, reflect t2, reflectt1);
>   end;
```

7.3 Functors

Functors bilden Strukturen aus Strukturen. Sie können generische Algorithmen ausdrücken und erlauben es, daß bestimmte Teile des Programms unabhängig voneinander geschrieben und compiliert werden können.

Für nähere Informationen siehe [L.C91],Kapitel 7.

Kapitel 8

Imperatives Programmieren

Wie bereits erwähnt, ist Standard ML keine reine funktionale Programmiersprache. Sie besitzt auch einige imperative Eigenschaften wie Referenztypen, Arrays und Kommandos für Input und Output.

8.1 Kontrollstrukturen

ML unterscheidet Kommandos (Commands) nicht von Ausdrücken (Expressions). Ein Command wird als Ausdruck angesehen, der einen Zustand ändert, wenn er ausgeführt wird. Die meisten Commands haben den Typ Unit und liefern Unity () zurück.

Ähnlich wie Pascal, besitzt ML folgende grundlegenden Kontrollstrukturen:

- *if then else* Ausdrücke,
- die *while do* Schleife,
- und die *case* Anweisung.

Wie bereits im dritten Kapitel beschrieben, hat eine **if expression** folgende Form:

$$\textit{if}\ \mathrm{E}\ \textit{then}\ \mathrm{E}_1\ \textit{else}\ \mathrm{E}_2;$$

Sie wird auch als *conditional command* bezeichnet. Der Ausdruck E_1 wird nur dann berechnet, wenn der Ausdruck E wahr ist, ansonsten wird E_2 berechnet. Es ist dabei auch erlaubt, daß bei der Berechnung des Ausdrucks E der Zustand des Systems verändert wird.

Eine **case expression** hat folgende Form:

$$\textit{case}\ \mathrm{E}\ \textit{of}\ \mathrm{P}_1 \Rightarrow \mathrm{E}_1 \mid ... \mid \mathrm{P}_n = \mathrm{E}_n;$$

Ein typische Beispiel wäre:

```
- fun diff q r = case r - q of
>   0 => "zero"
> | 1 => "one"
> | 2 => "two"
> | n => if n < 10 then "lot" else "lots and lots"
```

Beim "pattern matching" ist es auch möglich, mit einem "_" (underline character) den allgemeinen Fall zu behandeln:

```
- fun diff q r = case r - q of
>   0 => "zero"
> | 1 => "one"
> | 2 => "two"
> | _ => "lots and lots"
```

Eine **Serie** von Commands kann in folgender Form angegeben werden:

$$(\mathrm{E}_1;\mathrm{E}_2;...;\mathrm{E}_n);$$

Dabei werden die Ausdrücke von links nach rechts berechnet, und jeder Ausdruck kann den Zustand des Systems und somit den nächsten Ausdruck beeinflussen. Der einzige Fall, bei dem man die Klammern weglassen kann, ist der der *let expression*:

$$let\ \mathrm{D}\ in\ \mathrm{E}_1;\mathrm{E}_2;...;\mathrm{E}_n\ end;$$

Für Wiederholungen kann die **while expression** verwendet werden. Sie hat wie in Pascal die Form:

$$while\ \mathrm{E}_1\ do\ \mathrm{E}_2;$$

Wird der Ausdruck E_1 *false*, dann wird dieses Command beendet, ist der Ausdruck aber *true*, dann wird E_2 berechnet und danach das Command nochmals ausgeführt.

8.2 Referenz Typen

Referenzen in ML können als Speicheradressen angesehen werden. Sie sind vergleichbar mit Variablen in Pascal oder ähnlichen Sprachen und dienen als Pointer oder verkettete Datenstrukturen.

Eine ML Referenz bezeichnet eine Adresse im Speicher, und diese Adresse beinhaltet einen Wert oder eine neue Adresse.

Der Konstruktor *ref* erzeugt eine Referenz. Die Funktion *ref* liefert jedesmal, wenn sie angewendet wird, eine neue Adresse (einer freien Speicherstelle) zurück.

Die Funktion *!* (Rufzeichen), angewandt auf eine Referenz, liefert deren Inhalt zurück. Dieser Vorgang wird als **dereferencing** bezeichnet.

Die Zuweisung

$$E_1 := E_2$$

berechnet E_1 (muß ein Referenztyp sein) und E_2. Dabei ersetzt sie den Wert in E_1 mit dem Wert von E_2. Syntaktisch gesehen ist *:=* eine Funktion und $E_1 := E_2$ ein Ausdruck, obwohl sich der Zustand des Systems ändert. Der zurückgelieferte Wert ist stets (), also Unit. Nun aber einige einfache Beispiele:

```
- val p = ref 5
> and q = ref 2;

val p = ref 5 : ref int
val q = ref 2 : ref int

- (!p,!q);
val it = (5,2) : int * int

- p := !p + 2;
val it = () : unit

- (!p,!q);
val it = (7,2) : int * int
```

Referenzen können auch zu Datenstrukturen gehören, also Elemente von Tupel, Listen (...) sein. Auch Referenzen auf Referenzen sind erlaubt.

Sehr interessant sind zyklische Datenstrukturen. Überlegen wir uns einmal folgendes: Definieren wir *cp* derart, daß es eine Referenz auf eine Funktion (*cfact*) ist, die von einem Integerwert immer den darauffolgenden Wert zurückliefert.

```
- val cp = ref ( fn k => k+1);
val cp = ref fn : (int -> int) ref

- fun cfact n = if n = 0 then 1 else n * !cp(n-1);
val cfact = fn : int -> int

- cfact 8;
val it = 64 : int
```

Nun, jedesmal wenn *cfact* aufgerufen wird, nimmt es den momentanen Wert von *cp* und berechnet das Ergebnis. Das bedeutet: *cfact(8) = 8 x 8 = 64* . Nun soll *cp cfact* enthalten:

```
- cp := cfact;
val it = () : unit

- cfact 8;
val it = 40320
```

Jetzt enthält *cfact* über *cp* eine Referenz auf sich selbst. Die Funktion wird rekursiv und kann die Faktorielle berechnen. Viele funktionale Sprachen verwenden diese Methode, um rekursive Funktionen zu implementieren. Mit Hilfe von Referenzen ist es nun möglich, viele Datenstrukturen zu implementieren. So zum Beispiel Lazy Lists, Ring Buffers, Functional Arrays, (...). Alle jene Methoden vorzustellen, würde aber den Umfang dieses Skriptums sprengen. Für alle Interessierten sei hier auf "ML, for the Working Programmer, Chapter 8 [L.C91] verwiesen.

8.3 Input/Output in ML

Grundsätzlich gibt es zwei verschiedene Arten Input/Output zu behandeln.

- eine rein funktionale Behandlung von Input/Output ist nur möglich wenn sowohl der Input als auch der Output als unendliche Folgen betrachtet werden, d.h. als "lazy" ansieht.
- in einer strikten Sprache muß Input/Output imperativ behandelt werden, es muß also der Begriff des Zustandes eingeführt werden. Es ist auch praktischer Input/Output so aufzufassen.

8.3.1 Die Struktur von IO

In Standard ML sind nun Ein- und Ausgabe imperativ. Die Kommunikation erfolgt über sogenannte **streams**. Der *input stream* ist mit einem Datenerzeuger (Tastatur, File,...) und der *output stream* mit einem Datenkonsumierer (Drucker, File,...) verbunden.

Im folgenden möchte ich kurz auf die einzelnen Typen und Funktionen eingehen.

Input Streams haben den Typ *instream*, und Output Streams besitzen den Typ *outstream*. Beide Typen dürfen nicht verwechselt werden, denn sie haben unterschiedliche Bedeutung. Die Funktionen *open_in(s)* und *open_out(s)* erzeugen einen Stream, der mit einem File oder Device *s* verbunden ist. *close_in(is)* und *close_out(os)* schließen den Stream. Ein Inputstream kann auch durch das Device geschlossen werden. Die Funktion *input(is,n)* entfernt

n Character vom Stream *is* und liefert einen String zurück. *lookahead(is)* liefert denselben Wert wie *input(is,1)*, entfernt den Character aber nicht vom Stream. Es wird gewartet, bis ein Character zur Verfügung steht oder der Stream geschlossen wird. *end_of_stream(is)* ist *true*, wenn *is* leer oder geschlossen ist. Um einen Character *s* in einen Stream zu schreiben, genügt der Aufruf *output(os,s)*. *std_in* und *std_out* sind die Standard Input und Output Streams. Bei einer interaktiven Sitzung ist dies die Tastatur. Die Input/Output Primitiva haben folgende Signatur:

```
- signature IO = sig
>   type instream and outstream
>   exception Io of string
>   val sdt_in          : instream
>   val open_in         : string -> instram
>   val close_in        : instream -> unit
>   val input           : instream * int -> string
>   val lookahead       : instream -> string
>   val end_of_stream   : instream -> bool
>   val std_out         : outstream
>   val open_out        : string -> outstream
>   val close_out       : outstream -> unit
>   val output          : outstream * string -> unit
> end;
```

Es gibt noch zwei weitere Funktionen: *print(x)* und *makestring()*. Der Aufruf von *print(x)* schreibt den Wert x auf den Bildschirm und *makestring(x)* konvertiert x in einen String. Beide Funktionen sind *overloaded*, sie funktionieren mit den Typen *int, real, bool*.

8.3.2 Ein kleines Beispiel

Um eine Zeile einzulesen und in einen String zu verwandeln, genügt folgende Funktion:

```
- open IO;
- fun input_line is =
>   let fun getstring s =
>         case input(is,1) of
>              ""    => s               (* input has closed *)
>            | "\n"  => s ^ "\n"        (* end of line *)
>            | c     => getstring(s^c)
>   in getstring "" end;
```

Zuerst wird die Struktur *IO* geöffnet und dann vom Inputstream Zeichen für Zeichen eingelesen. Um eine Eingabe von der Tastatur zu erreichen, genügt der Aufruf *input_line std_in*,

da *std_in* der Standard Inputstream, also das Keyboard, ist. Will man eine Eingabe vom File, muß das File mit *open_in "Filename"* als Inputstream definiert werden, und dieser Stream kann dann bei *input_line* übergeben werden. Ein Beispiel wäre:

val file = open_in "harry"; input_line file;

Kapitel 9

Einführung in CML

Einige der wichtigsten Charakteristika einer parallelen Programmiersprache sind sicherlich die **Synchronisations-** und **Kommunikationsprimitiva**. Es gibt buchstäblich Hunderte von *concurrent programming languages*, die ich hier nicht im einzelnen aufzählen möchte. Im folgenden möchte ich auch nicht näher auf die Paradigmen des *concurrent programming* eingehen, sondern auf die Vorlesung "Betriebssysteme 1" verweisen. Ein empfehlenswertes Buch zu diesem Thema ist auch [9]. Ein Buch über CML gibt es noch nicht, diese "Einführung in CML" genügt jedoch für die Lösung der in den Übungen gestellten Programmieraufgaben. Da ich auch versuche, die Grundlagen und Ideen von CML zu erläutern, empfiehlt es sich, dieses Kapitel zuerst durchzulesen und dann erst mit kleinen Übungsbeispielen zu beginnen.

9.1 Die Entwicklung von CML

CML wurde gänzlich in SML/NJ geschrieben. Dabei profitiert CML bei seiner Ausführung und Performance eindeutig vom Multi-Pass-Compiler von SML/NJ. Dessen Frontend ist ziemlich normal (scanning, parsing, type-checking,..) das Backend aber ist interessant. Es benutzt eine Repräsentation, genannt **continuation passing style (CPS)**.
CPS ist eine spezielle Form des λ-Kalküls, die eine eindeutige Repräsentation für alle Kontrollbefehle (Verzweigungen, loops, function calls und returns) hat. Somit hat der Compiler die Möglichkeit, Kontrollbefehle so schnell wie möglich zu machen.

Bei der Entwicklung der "concurrent extension" von SML/NJ wollte man folgenden Punkten genügen:

- Dynamische Prozeßerzeugung (thread creation).
- Synchrone Kommunikation über channels.
- Dynamische Erzeugung von channels.

- Unterstützung der "generalized selective communication".

Damit der Prozessor nicht monopolisiert wird, verwendet CML *preemptive thread scheduling*. Dazu wird der Unix Intervall Timer (SIGALRM) in Zusammenarbeit mit dem Signal Mechanismus, der von SML/NJ zur Verfügung gestellt wird, verwendet. Weiters bietet CML die automatische Erkennung von threads und channels, auf die nicht mehr zurückgegriffen werden kann. Sie werden "gekillt" und der Speicherplatz durch automatische "Garbage Collection" wieder freigegeben.

Ein praktisches Anwendungsbeispiel von CML ist **eXene**, ein multi-threaded Grafikinterface, das ungefähr den Level von XLib besitzt.

9.2 Selektive Kommunikation

Im wesentlichen wurde bei CML versucht, CSP's "selective communication" nachzubauen. Für jene, die von CSP noch nichts gehört haben sollten, möchte ich im folgenden (wirklich nur sehr kurz) auf CSP eingehen.

1978 hat C.A.R. Hoare eine Notation für eine Menge von sequentiellen Prozessen, die parallel laufen und mittels *synchronous message passing* kommunizieren, eingeführt und diese **CSP** (*Communicating Sequential Prozesses*) genannt. Diese hat einen Inputoperator (`P?x`), einen Output Operator (`P!x`) und `cobegin` Ausdrücke zur Prozeßerzeugung. Wichtig dabei ist, daß es zum Blockieren der Prozesse kommen kann, bis die Kommunikation erfolgreich ausgeführt worden ist. An folgendem Prozess-Zeitdiagramm (9.1) ist das Blockieren sehr gut ersichtlich.

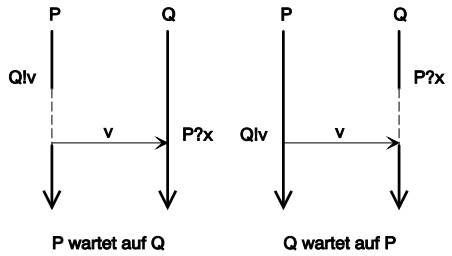

Abbildung 9.1: Rendezvous

Im ersten Fall ist `P` gezwungen auf `Q` zu warten. Die Outputoperation `Q!P` wird erst durchgeführt, wenn `Q` bereit ist, die Meldung `v` zu empfangen. Im zweiten Fall ist `Q` gezwungen zu warten, bis `P` seine Meldung an `Q` schickt.
Eine der Hauptideen bei CSP war der Begriff der selektiven Kommunikation (**selective communication**), welche als Erweiterung von Dijkstra's "**guarded command**" (1975)

gedacht war. Ein Ausdruck S, vorangestellt ein *Guard* (= boolscher Ausdruck B), wird genau dann ausgewertet, wenn der guard den Wert "true" liefert und das Statement C bereit ist, ausgeführt zu werden. Ein *guarded communication statement* hat somit folgende Form:

```
B; C -> S
```

Guarded communication statements können in if- und do-Ausdrücken auftreten und haben dabei folgende Bedeutung:

- **if-statement**
  Bei einem oder mehreren Guards die "true" liefern, wird einer zufällig ausgewählt. Nur wenn alle Guards "false" liefern, terminiert der Ausdruck, es sei denn ein *communication statement* ist blockiert. Dann wird gewartet, bis ein Ausdruck ausgeführt werden kann.

- **do-statement**
  Die Funktion ist ähnlich dem if-statement, nur werden die Operationen solange ausgeführt, bis alle Guards "false" liefern.

Sind bei den *communication statements* In- und Output Operationen erlaubt, dann spricht man auch von "**generalized (symmetric) selective communication**".

Als Beispiel sei nun noch ein Programm in CSP angeführt, das Werte von einem `Source` in eine `Destination` kopiert und dabei einen kleinen Puffer der Größe 10 verwendet:

```
Copy :: var buffer [1:10] : char
        var front := 1, rear := 1, count := 0
        do
            count < 10; Source?buffer[rear] ->
                        count := count + 1;
                        rear := (rear mod 10) + 1
        []
            count > 0 ; Dest!buffer[front] ->
                        count := count - 1;
                        front:= (front mod 10) +1
        od
```

In diesem Programm werden zwei *guarded communication statements* eingesetzt. Das erste Statement kann ausgeführt werden, wenn im `buffer` Platz ist und `Source` ein Zeichen ausgeben kann. Das zweite Statement wird ausgeführt, wenn sich im `buffer` (mindestens) ein Zeichen befindet und `Dest` ein Zeichen annehmen kann. Das do-statement wird nie beendet, da nicht beide Guards zur selben Zeit "false" sein können (zumindest ein boolscher Ausdruck ist immer erfüllt).

9.3 Grundlegende Operationen

Ein CML Programm besteht aus einer Sammlung von **threads**, die "synchronous message passing" zur Synchronisation und Kommunikation verwenden. Dabei ist unter einem thread ein ML-Prozeß, also die sequentielle Abarbeitung von ML Ausdrücken, zu verstehen. Grundlegende Operationen sind :

```
val spawn   : (unit -> unit) -> thread_id

val channel : unit -> '1a chan

val accept  : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit
```

Dabei erzeugt `spawn` einen neuen thread, `channel` einen (weakly-polymorphic) channel und `send` und `accept` sind für die synchrone Kommunikation verantwortlich.
Eine Kommunikation zwischen zwei threads zustande zu bringen, ist denkbar einfach. Man definiert mit der Funktion `channel` einen Kommunikationskanal, den die zwei threads zur Kommunikation benützen können. Führt nun ein thread eine `send` oder `accept` Operation aus, dann ist dieser solange blockiert, bis der zweite thread die komplementäre Operation durchführt. Am besten sehen wir uns dazu ein kleines Beispiel an:

```
fun simple_comm () =
  let
      val ch = channel ()
      val pr = CIO.print
  in
      pr "HI-0 \n";
      spawn (fn () => (pr "HI-1 \n"; accept ch; pr "BYE-1 \n"));
      spawn (fn () => (pr "HI-2 \n"; send (ch,17); pr "BYE-2 \n"));
      pr "BYE-0 \n"
  end;
```

In der Funktion `simple_comm` erzeugt ein root-thread einen Channel `ch` und zwei child-threads. Jeder thread liefert am Beginn und am Ende seiner Abarbeitung eine Meldung. Die Ausgabe erfolgt durch die Funktion `CIO.print`, wobei die CIO Struktur die Input- und Outputoperationen von CML enthält. (Wie man dieses Beispiel zum Laufen bringt, erfahren Sie dann im Kapitel 9.5.)

Die oben erwähnten Funktionen haben nur indirekt mit CSP zu tun. Wie ich bereits erwähnt habe, hat man bei CML aber versucht, gerade CSP's selektive Kommunikation zu implementieren, und das wollen wir uns näher ansehen.

Bei genauerer Betrachtung stellen wir beim traditionellen select Mechanismus folgende Eigenschaften fest:

1. Wahlweise Input oder Output Operationen.
2. Eine Aktion, die mit einer Operation verbunden ist.
3. Nichtdeterministische Auswahl.
4. Synchronisation.

Um das zu erreichen, ist man bei CML folgendermaßen vorgegangen: Man definiert einen neuen Typ, **event** genannt, der die synchrone Operation repräsentiert, startet mit **base events** um die Kommunikationsoperationen zu repräsentieren, verbindet mittels **Kombinatoren** *events* und Aktionen und bietet eine nichtdeterministische Auswahl an. Im wesentlichen wird so CSP's *selective communication* nachgebaut und mit der Eigenschaft, daß nun auch Input und Output Operationen als "select statement" auftreten dürfen, erweitert.

Die oben erwähnten Funktionen sind:

```
val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val choose   : 'a event list -> 'a event
val wrap     : ('a event * ('a -> 'b)) -> 'b event

val sync     : 'a event -> 'a
```

Diese Funktionen lassen sich auch den Punkten 1-4 zuordnen:

1. `receive`,`transmit`.
   Diese bilden die *base-events*, und repräsentieren die channel-Input und Output Operationen.
2. `wrap`.
   Dieser Combinator bindet eine Aktion (die hier durch eine Funktion repräsentiert wird) an einen "event" Wert.
3. `choose`.
   Dieser Combinator ermöglicht die nichtdeterministische Auswahl der "event" Werte. (Analog zum *generalized selective communication* Mechanismus.)
4. `sync`.
   Diese Operation erzwingt die Synchronisation auf einen "event" Wert.

Wichtig dabei ist, daß bei der Erzeugung eines events noch nichts "passiert". Erst bei Anwendung der `sync` Funktion auf einen event, kommt es zur tatsächlichen Abarbeitung des events.

Als einfaches Beispiel sei hier die Implementation der grundlegenden Operationen (`send` und `accept`) gezeigt.

```
val accept = sync o receive
val send   = sync o transmit
```

Ein etwas aufwendigeres Beispiel ist folgendes:

```
fun accum sum = sync (
    choose [
            wrap (receive addCh, fn x => accum (sum+x)),
            wrap (receive subCh, fn x => accum (sum-x)),
            wrap (transmit (readCh,sum), fn () => accum sum)
           ]);
```

Dieser thread implementiert einen Akkumulator, der den über die beiden Eingabekanäle `addCh` und `subCh` erhaltenen Wert entweder zur aktuellen Summe addiert oder davon subtrahiert. Über den Channel `readCh` kann der aktuelle Wert des Akkumulators abgefragt werden.

Als quasi Gegenstück zur `wrap`-Funktion existiert noch der `guard` Combinator.

```
val guard : (unit -> 'a event) -> 'a event
```

Hier wird ebenfalls eine Aktion an einen "event" gehängt. Der Unterschied zu `wrap` ist aber, daß hier die Aktion vor den Synchronisationspunkt gehängt wird und nicht erst danach. Sehr schön ist das an folgendem Beispiel zu sehen:

```
fun timeout t = guard (fn () => waitUntil (add_time (t, current_time ())))
val timeout = fn : time -> unit event

sync (choose [ wrap (timeout (TIME{sec=5, usec=0}), fn () => raise TimeOut),
               input instream
             ])
```

Dieses Programm verlangt innerhalb von fünf Sekunden eine Eingabe über die Tastatur, sonst wird eine Fehlermeldung (`TimeOut`) erzeugt. Dabei liefert die Funktion `current_time` die aktuelle Uhrzeit und `add_time` addiert zwei Uhrzeiten miteinander.

9.4 Arbeiten mit CML

Nach Erläuterung der grundlegenden Funktionen und Funktionsweisen von CML möchte ich noch kurz auf das Arbeiten mit CML eingehen.

9.4.1 Starten von CML

CML ist vollständig in SML geschrieben. Die Arbeitsumgebung ist dieselbe wie bei SML/NJ und kann durch Aufruf von 'cml' gestartet werden. Wie bei SML antwortet das System mit dem Top-Level Prompt.

```
>cml
Concurrent ML -- version 0.9.8 -- February 1, 1993
Standard ML of New Jersey, Version 0.93, February 15, 1993

                          CML - Version (c) ab,1994
                             Library Version 0.2
val it = () : unit
-
- open CML;
```

Mit dem Kommando "cml" werden SML und verschiedene CML Bibliotheksmodule (Strukturen) geladen, die durch ein `open` geöffnet werden können. Ansonsten müssen qualifizierte Namen für die Komponenten verwendet werden (siehe auch Kapitel 7, Modulare Programmierung). `CML` stellt den Kern von Concurrent ML dar und enthält Definitionen für threads, channels und events. Um also mit CML bequem arbeiten zu können, empfiehlt es sich, gleich nach dem Starten von CML die Struktur CML mit dem Befehl `open CML` zu öffnen. Weitere top-level Strukturen sind:

- `RunCML`. Enthält Funktionen zum Ausführen und Beenden von CML-Programmen.
- `CIO`. Diese Struktur besitzt Funktionen zur Unterstützung von multi-threaded Input und Output.

9.5 Starten von CML Programmen

Um nun ein CML-Programm auszuführen, ist die Funktion `doit` aus der Struktur `RunCML` erforderlich:

```
- RunCML.doit;
val it = fn : (unit -> unit) * int option -> unit
```

CML ist zur Zeit so implementiert, daß ein Programm nur in einer single-thread Umgebung gestartet werden kann, und weitere threads nur innerhalb eines Programmes erzeugt werden können. Das erste Argument von `doit` ist somit der root-thread (eine Funktion mit der Signatur `unit -> unit`). Das zweite Argument gibt die Größe der scheduling Zeitscheiben in Millisekunden an. Man kann mit `SOME msec` die Zeitdauer angeben, oder mit `NONE` das preemptive Scheduling ausschalten. (Dies ist aber nicht unbedingt zu empfehlen, da einige Systemfunktionen nur mit preemptive Scheduling funktionieren!)

Unsere Funktion `simple_comm` aus Kapitel 9.3 kann also folgendermaßen gestartet werden:

```
- simple_comm;
val it = fn : unit -> unit
- RunCML.doit (simple_comm, SOME 20);
HI-0
HI-1
HI-2
BYE-0
BYE-1
BYE-2
val it = () : unit
-
```

Bei der Abarbeitung wird zuerst das `print` des root-threads ausgeführt, und dann werden die neuen threads erzeugt. Die weiteren Ausgaben erfolgen, indem durch das preemptive scheduling abwechselnd immer einer der drei threads ausgeführt wird. Das dabei erhaltene Ergebnis stellt nur eines von mehreren Möglichkeiten dar, da die Auswahl von Prozesse in CML nicht deterministisch ist.

9.6 Anwendungsbeispiele

9.6.1 Streams

Eine Anwendung von Prozessen und channels ist die Erzeugung eines *streams*. Ein stream ist eine Folge von Werten eines bestimmten Typs. Streams können somit als unendliche Listen betrachtet werden. Die Funktion `makeNatStream` stellt die Implementierung für einen Stream der natürlichen Zahlen vor.

```
fun makeNatStream () = let
   val ch = channel()
   fun f i = (send(ch, i); f(i+1))
   in
     spawn (fn () => f 0);
```

```
    ch
  end;

val makeNatStream = fn : unit -> int chan
```

Dies wird erreicht, indem die Funktion `f` als thread und der eigentliche stream als channel repräsentiert wird. Das nachfolgende Beispiel zeigt eine Anwendung von Streams.

9.6.2 Sieve of Eratosthenes

Die Methode, Primzahlen aus einer Folge von natürlichen Zahlen zu berechnen, ist bereits in Kapitel 6.3.2 vorgestellt worden. Hier eine interessante Implementation in CML:

```
fun cfilter (pre,inCh,outCh) =
  let
    val i = accept inCh;
  in
    if pre(i) then send(outCh,i) else ();
    cfilter (pre,inCh,outCh)
  end;

local
  fun sievep(inCh,res_ch) =
    let
      val outCh = channel();
      val p = accept inCh
      fun notmult p i = (i mod p) <> 0;
    in
      send(res_ch,p);
      spawn(fn () => cfilter(notmult p,inCh,outCh));
      sievep(outCh,res_ch)
    end;

  fun Nat2(outCh) =
    let
      fun count(i) = (send(outCh,i);
                      count(i+1))
    in
      count(2)
    end
in
```

```
  fun sieve(res_ch) =
    let
      val outCh = channel()
    in
      spawn(fn () => Nat2(outCh));
      spawn(fn () => sievep(outCh,res_ch))
    end;

  fun primes n =
    let
      val res_ch: int chan = channel();
      fun loop 0 = ()
        | loop i = (CIO.print(makestring(accept res_ch)^"\n");
                    loop(i-1))
    in
      sieve(res_ch);loop n
    end
end;
```

Die Funktion `cfilter` ist eine Filterfunktion; Werte werden vom Input (`inCh`) eingelesen und jene Werte, für die das Prädikat `pre` zutrifft, werden als Resultat an den `outCh` gesendet. Die Funktion `sievep` empfängt als Argument einen Input-stream (`inCh`), der als erstes Element die nächste Primzahl enthält. Diese wird als Ergebnis auf den `res_ch` gesandt. Der verbleibende Stream wird gefiltert, indem ein neuer thread erzeugt wird, der die Vielfachen der neuen Primzahl streicht. Das Ergebnis des Filters wird in einem neuen Kanal empfangen, der den Input für den nächsten Funktionsaufruf von `sievep` darstellt.
Der Stream mit den natürlichen Zahlen wird durch die Funktion `Nat2` gebildet. Der erste Aufruf erzeugt die Zahl 2, die Funktion `sievep` generiert dann rekursiv die restlichen Zahlen. In `sieve` wird ein thread für die Erzeugung der natürlichen Zahlen und ein thread für die Ausführung der Funktion `sievep` generiert. Die Funktion `primes` liest n Primzahlen von res_ch ein und gibt diese am Bildschirm aus.
Insgesamt werden während der Ausführung der Funktion `primes` zwei threads in der Funktion `sieve` erzeugt und für jede Primzahl ein thread zur Entfernung derer Vielfachen. Die Berechnung und Anzeige der ersten 500 Primzahlen dauert ca. 6 sec (bzw. Rechenzeit 310 msec).

```
- RunCML.doit(fn () => (primes 3),SOME 20);
2
3
5
val it = () : unit
```

Die Funktion `primes_of` ermöglicht eine interaktive Eingabe der Anzahl der zu berechnenden Primzahlen. Es wird dazu mit der Funktion `CIO.input` ein String vom Standardinput eingelesen und in eine Integerzahl konvertiert. Diese Zahl wird als Argument an die Funktion `primes` übergeben.

```
fun primes_of () =
  let
    fun get_number () =
      let
        fun getstring s =
            case CIO.input(CIO.std_in, 1) of
              " "  => s
            | "\n" => s
            | c    => getstring(s ^ c)
      in
        (CIO.print("\n Wieviele Primzahlen sollen berechnet werden ? ");
         getstring "")
      end
  in
    CIO.print("\n CML- Beispiel: Das Sieb des Eratosthenes\n");
    primes (StringCvt.atoi(get_number ()))
  end;

RunCML.doit (primes_of, SOME 20);
```

Die Funktion `atoi` aus der Struktur `StringCvt` wandelt einen String in einen Integerwert um. Bei einem Fehler wird eine Exception `Convert` erzeugt. Die Struktur StringCvt ist in der Library Extension 0.2 von SML/NJ definiert.
Nun können wir das Programm ausprobieren und uns die ersten drei Primzahlen angeben lassen. Nach dem Aufruf von `RunCML.doit` erhalten wir folgendes Resultat:

```
- RunCML.doit (primes_of, SOME 20);

 CML- Beispiel: Das Sieb des Eratosthenes

 Wieviele Primzahlen sollen berechnet werden ? 3
2
3
5
val it = () : unit
-
```

Anhang A

Tabellen

A.1 Vordefinierte Identifier

A.1.1 Wahrheitswerte

type bool	Signatur	
true, false	bool	
not	bool $->$ bool	Negation
=	"a $*$ "a $->$ bool	Gleichheit
<>	"a $*$ "a $->$ bool	Ungleichheit

A.1.2 Numbers

type int oder **real**	Signatur	
+, −, $*$	int $*$ int $->$ int **or** real $*$ real $->$ real	Arithm. Operationen
$\sim$	int $->$ int **or** real $->$ real	Minuszeichen
abs	int $->$ int **or** real $->$ real	Absolutwert
div, mod	int $*$ int $->$ int	Division, Rest
$<$, $>$, $\geq$, $\leq$	int $*$ int $->$ bool **or** real $*$ real $->$ bool	Relationen
real	int $->$ real	Umwandlung Integer zu Real
/	real $*$ real $->$ real	Division
floor	real $->$ int	Umwandlung Real zu Integer
sqrt	real $->$ real	Quadratwurzel
sin, cos, arctan	real $->$ real	Trigonom. Funktionen
exp, ln	real $->$ real	Exponent und nat. Log.

exception	
Neg, Sum, Diff, Prod, Quot	Fehler bei $\sim$, +, −, $*$, /
Abs, Div, Mod, Floor	Fehler bei abs, div, mod und floor
Sqrt, Exp, Ln	Fehler bei sqrt, exp, ln

A.1.3 Strings

type string	Signatur	
^	string * string -> string	Verbindung von zwei Strings
size	string -> int	Länge des Strings
chr	int -> string	Erzeugt String aus ASCII-Code
ord	string -> int	Liefert ASCII-Code des Strings
explode	string -> string list	Erzeugt aus String eine Char-Liste
implode	string list -> string	Wandelt Char-Liste in einen String um

exception	
Ord, Chr	Fehler bei der String Konversion

A.1.4 Listen

type 'a list	Signatur	
nil	'a list	Die leere Liste
::	'a * 'a list -> 'a list	Konstruktion einer Liste
@	'a list * 'a list -> 'a list	Verbindung von Listen
rev	'a list -> 'a list	Reverse der List
map	('a -> 'b) -> 'a list -> 'b list	Map-Funktion
o	('b -> 'c) * ('a -> 'b) -> ('a -> 'c)	Compose-Operator

A.1.5 Leeres Tupel

type unit	Typ des leeren Tupels

A.1.6 Referenzen

type 'a ref	Signatur	
ref	'_a -> _a ref	Erzeugung einer Referenz
!	'a ref -> 'a	Dereferenzierung
:=	'a ref * 'a -> unit	Zuweisung zu einer Referenz

A.1.7 Input/Output Streams

type instream	Signatur	
std_in	instream	Für Input vom Device
open_in	string -> instream	File für Input öffnen
close_in	instream -> unit	Input Stream schließen
input	instream * int -> string	Zeichen vom Stream lesen
lookahead	instream -> string	Prüfen auf verbleibende Zeichen
end_of_stream	instream -> bool	Test, ob Eingabe beendet

type outstream	Signatur	
std_out	outstream	Für Output zum Device
open_out	string $->$ outstream	Öffnet File für Input
close_out	outstream $->$ unit	Schließt den Stream
output	outstream $*$ string $->$ unit	Zeichen senden

exception		
Io	string	Fehler während I/O

A.1.8 Exceptions

type exn	
exception Match	Fehler beim Pattern Matching
exception Bind	Fehler in *val* Deklaration
exception Interrupt	Interrupt Taste wurde gedrückt

A.2 Infixoperatoren

A.2.1 Precedences

Precedence	Operator	Precedence	Operator
7	/	4	=
7	div	4	<>
7	mod	4	<
7	*	4	>
6	+	4	$\geq$
6	-	3	$\leq$
6	^	3	:=
5	::	3	o
5	@		

Bem.: Der Infixoperator "::" ist rechtsassociativ, alle anderen sind linksassoziativ.

A.3 Strings

Durch **escape sequences**, welche mit einem sog. **backslash** (\) beginnen, können spezielle Zeichen in einen String eingefügt werden.

Steuerzeichen	
\n	Bewirkt einen Zeilenumbruch
\t	Bewirkt einen Tabulatorsprung
\"	Fügt ein Anführungszeichen ein
\\	Fügt einen backslash ein

Anhang B

Standard ML Syntax Diagramme

Folgende Abbildungen stammen aus dem Buch "ML for the Working Programmer" von L.C. Paulson [L.C91].

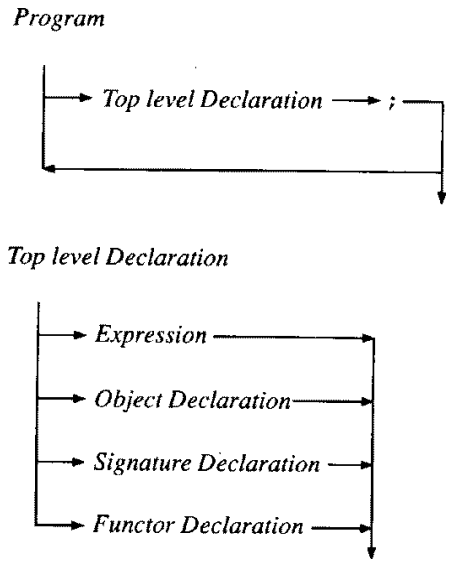

Abbildung B.1: Programs and Modules I

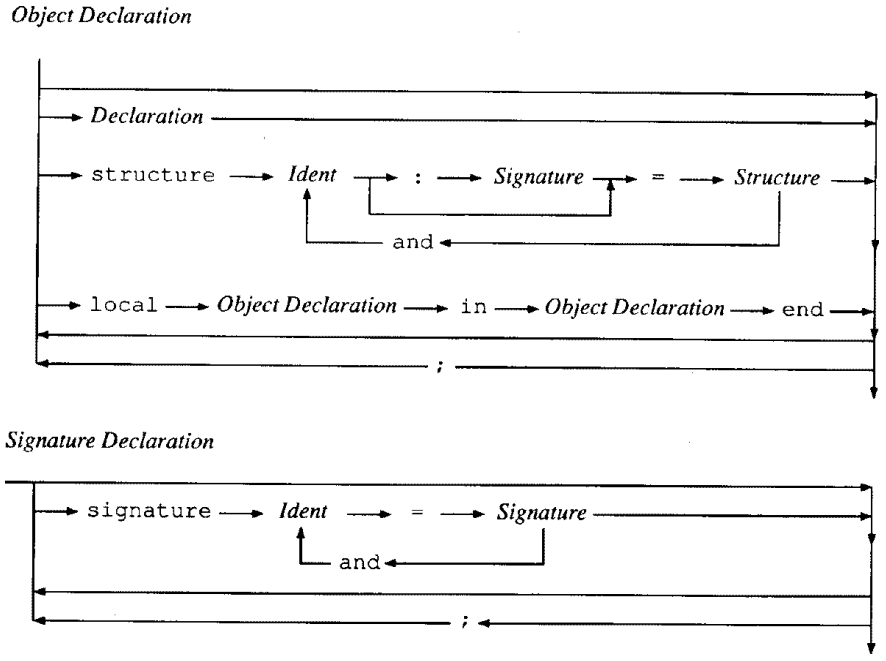

Abbildung B.2: Programs and Modules II

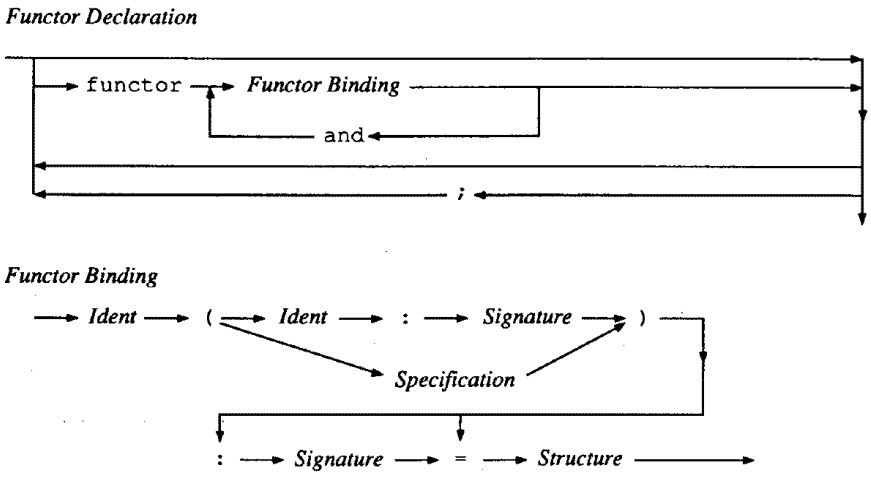

Abbildung B.3: Programs and Modules III

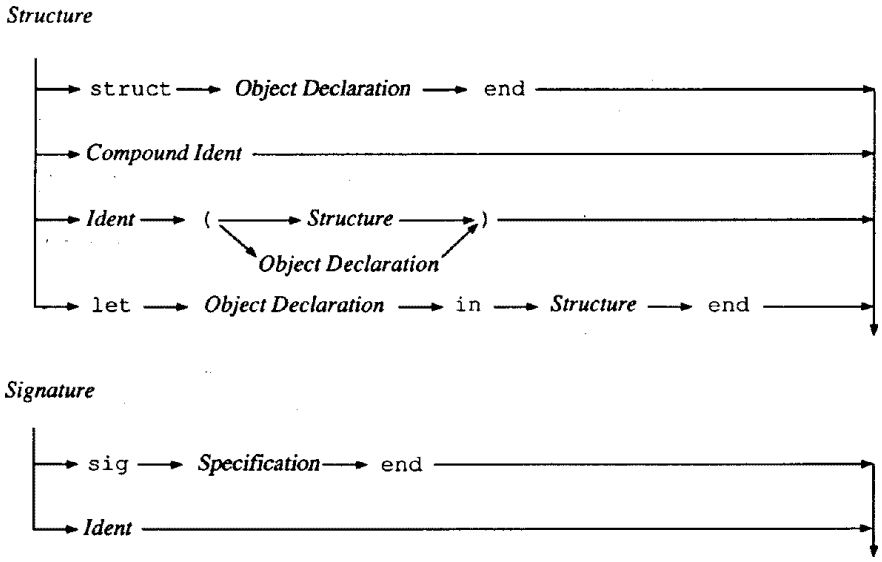

Abbildung B.4: Programs and Modules IV

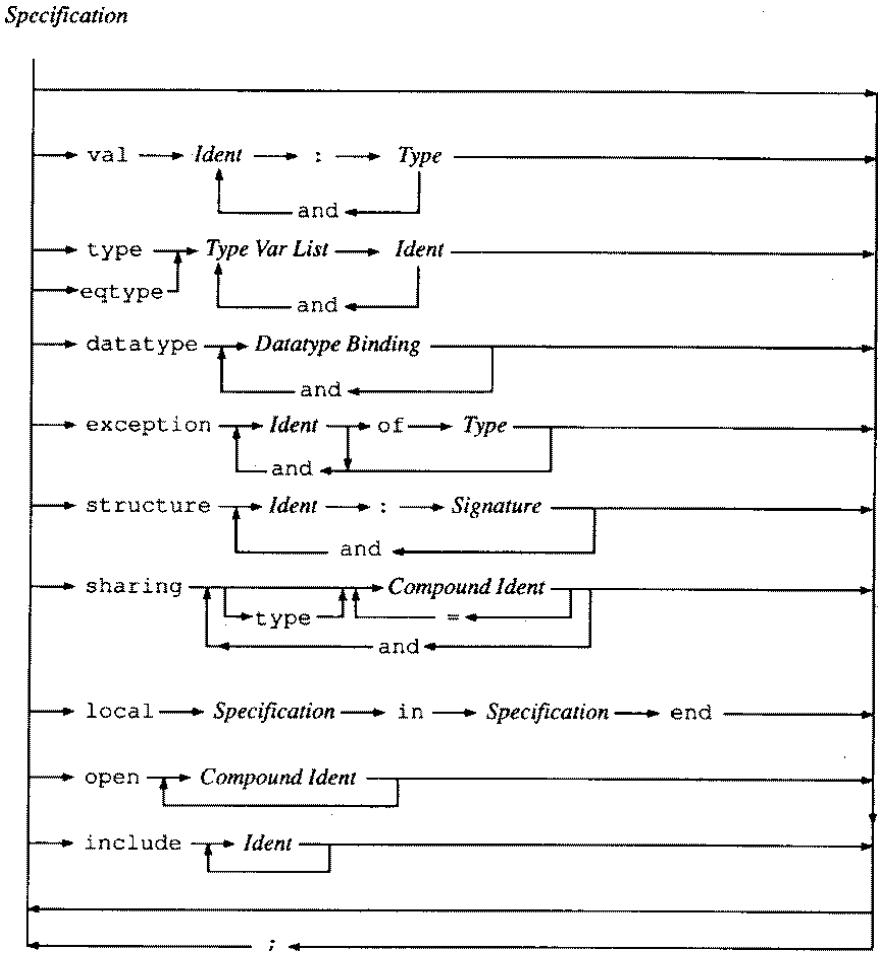

Abbildung B.5: Programs and Modules V

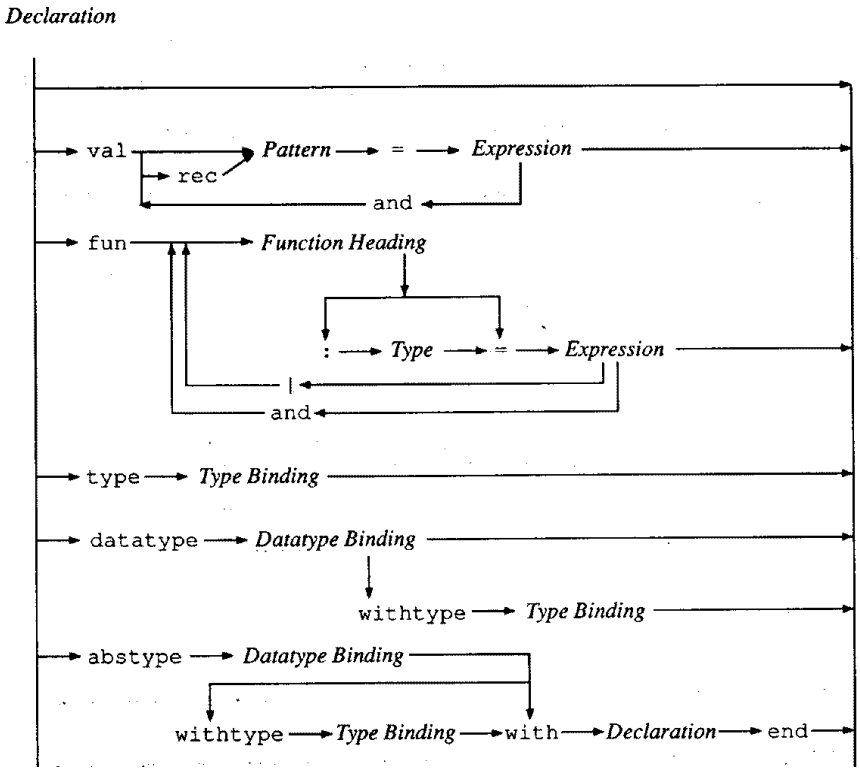

Abbildung B.6: Declarations I

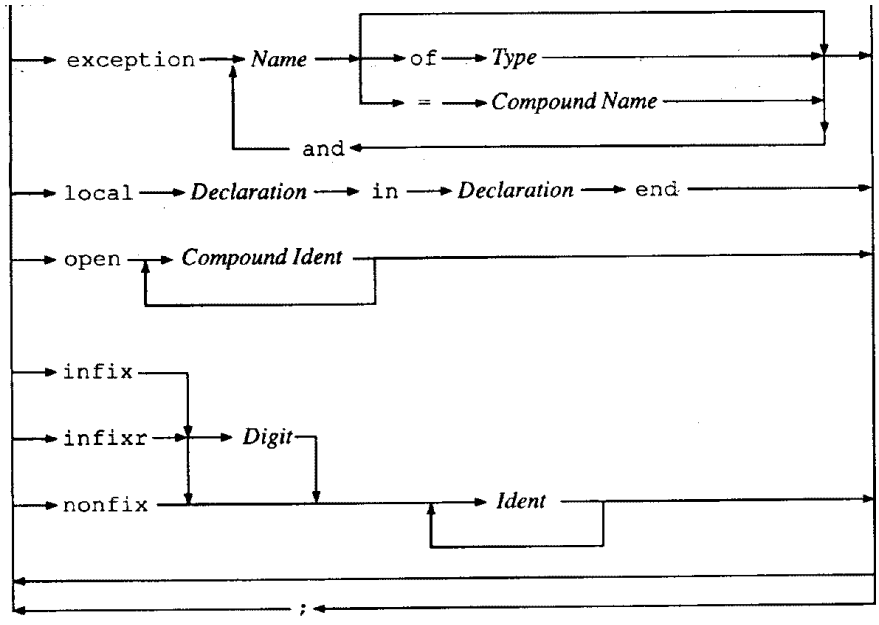

Abbildung B.7: Declarations I (cont.)

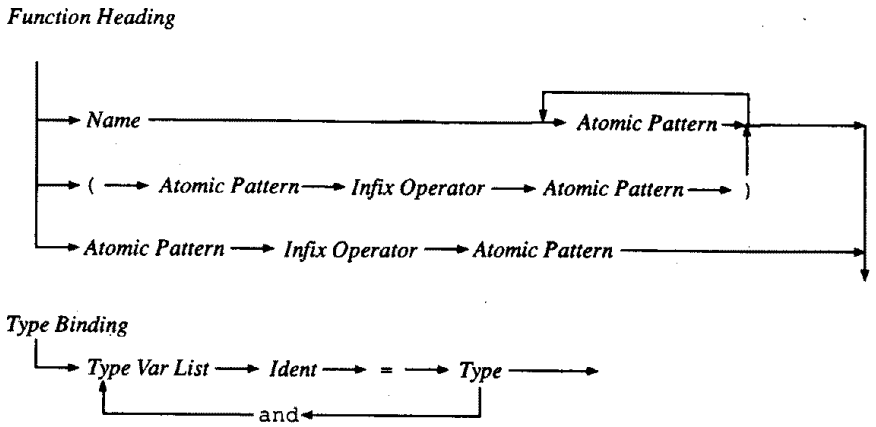

Abbildung B.8: Declarations II

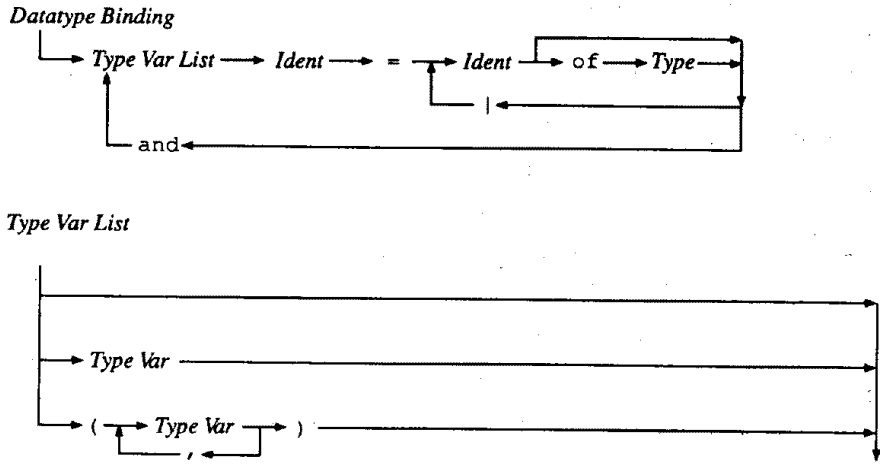

Abbildung B.9: Declarations III

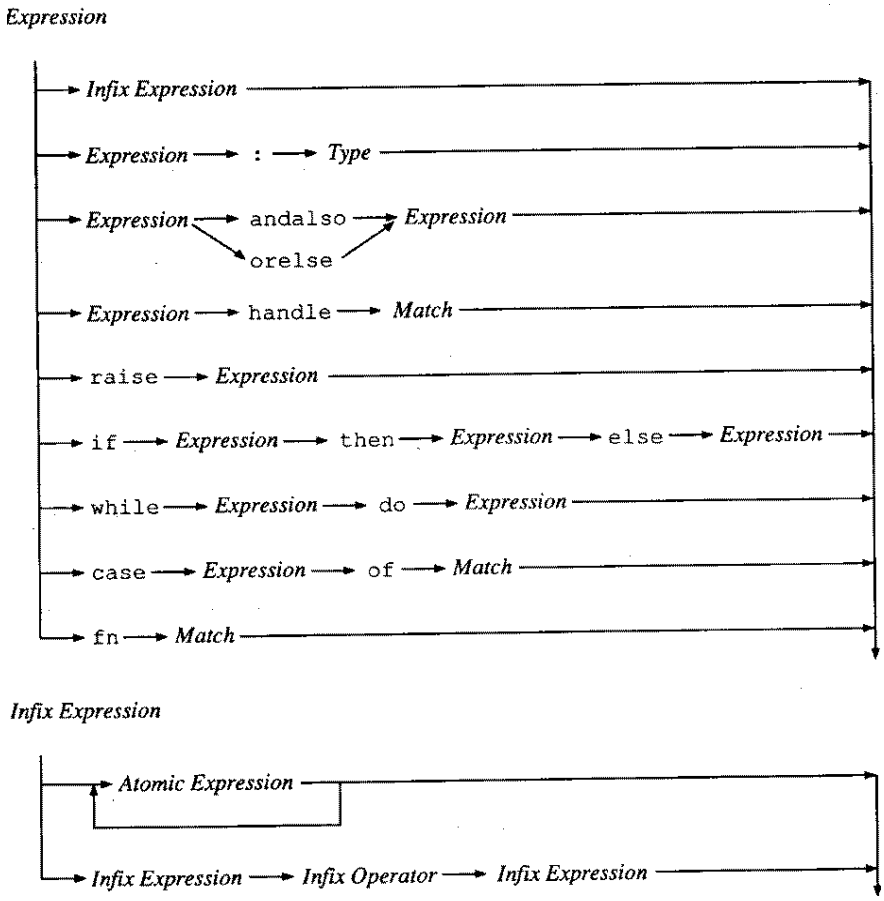

Abbildung B.10: Expressions I

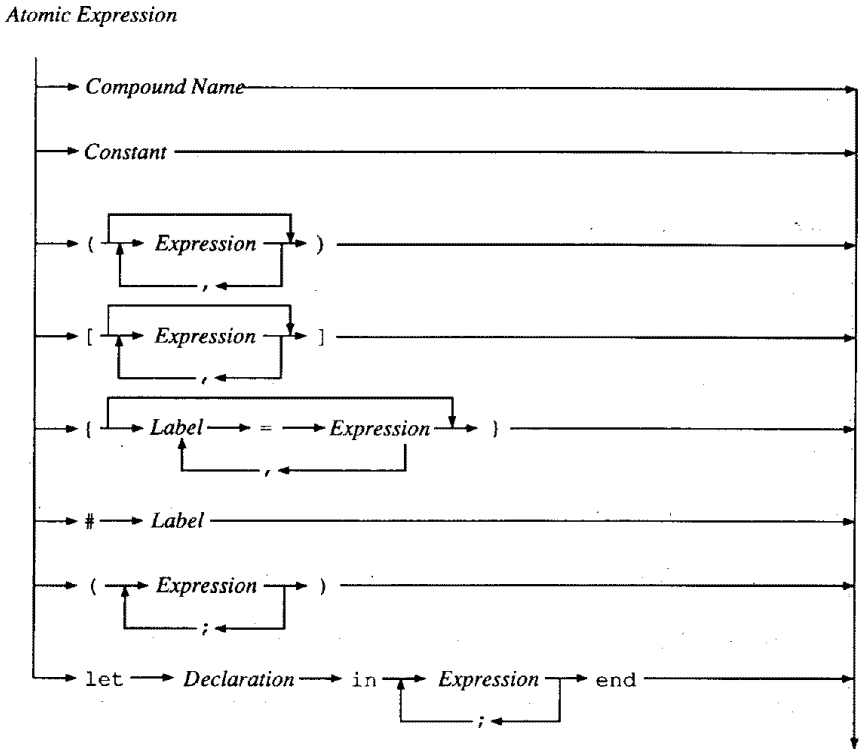

Abbildung B.11: Expressions II

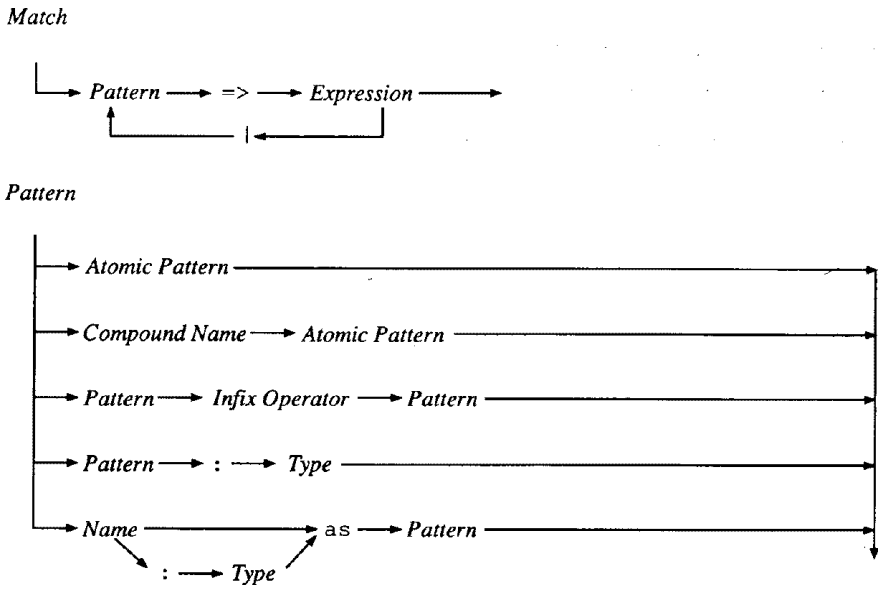

Abbildung B.12: Matches and Patterns I

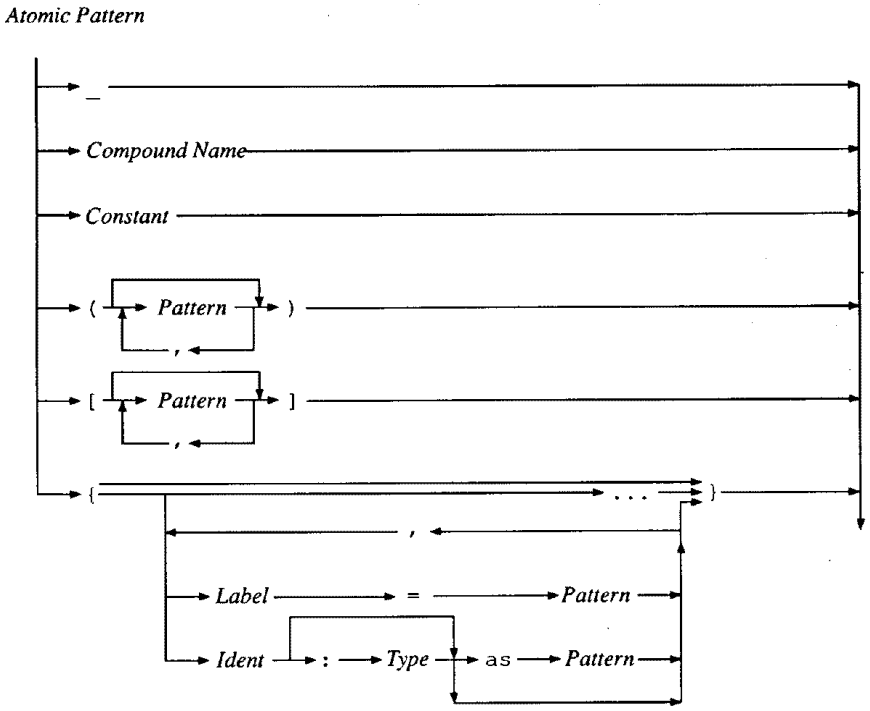

Abbildung B.13: Matches and Patterns II

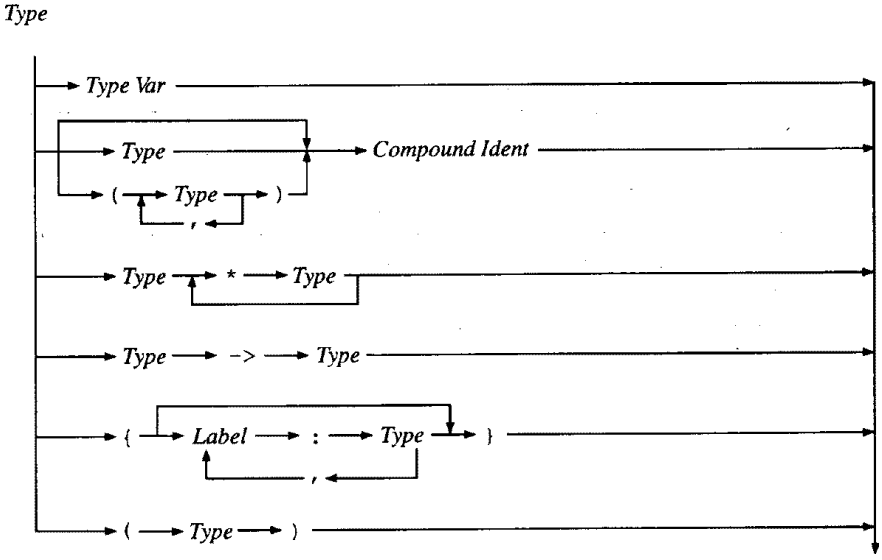

Abbildung B.14: Types

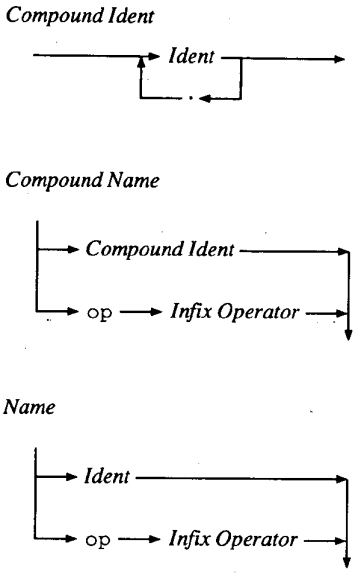

Abbildung B.15: Lexical Matters I

Infix Operator

any *Ident* that has been declared to be infix

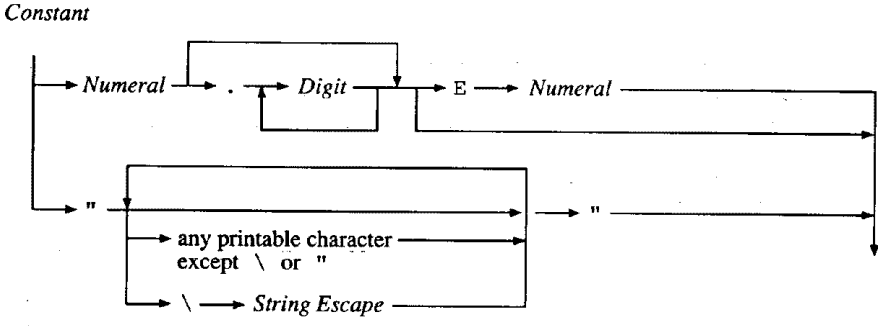

Abbildung B.16: Lexical Matters II

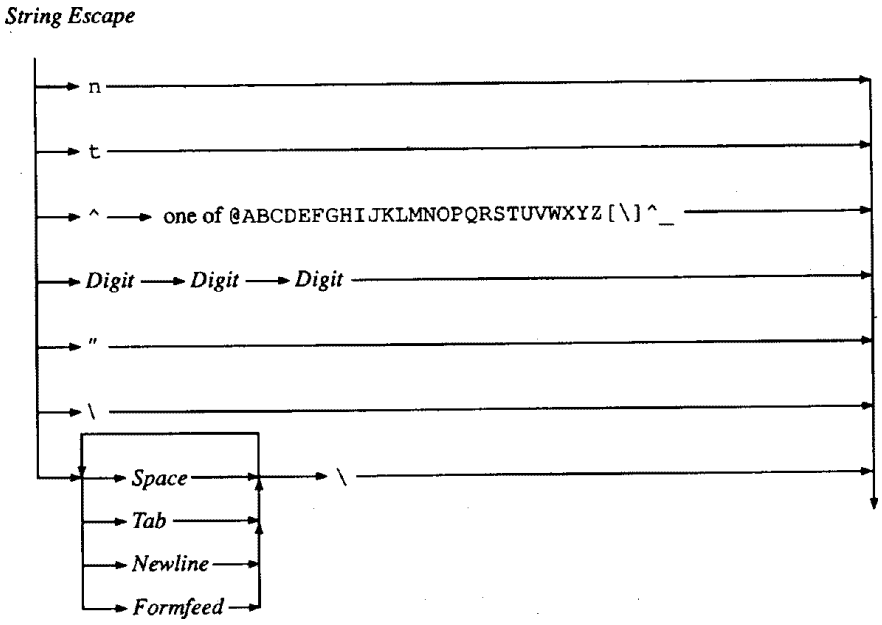

Abbildung B.17: Lexical Matters III

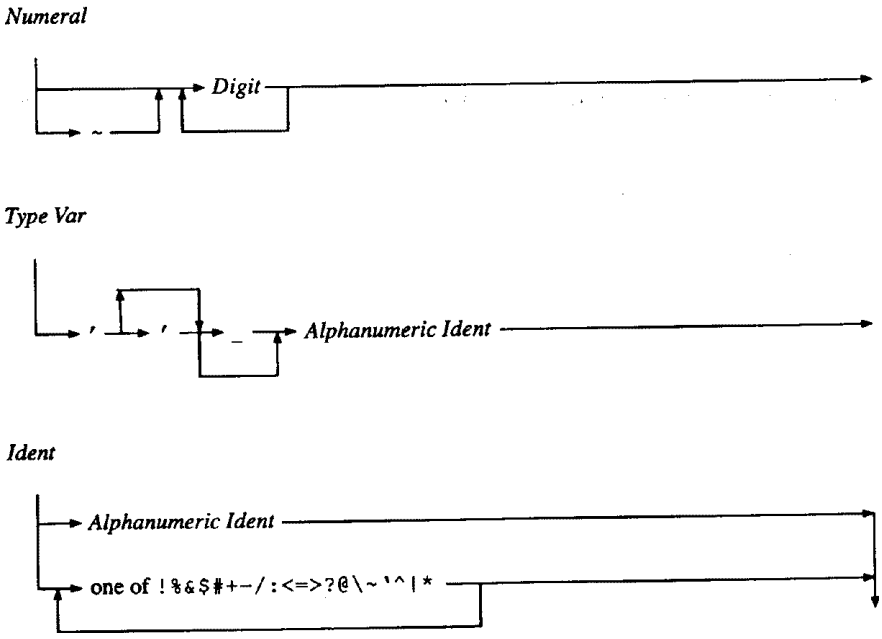

Abbildung B.18: Lexical Matters IV

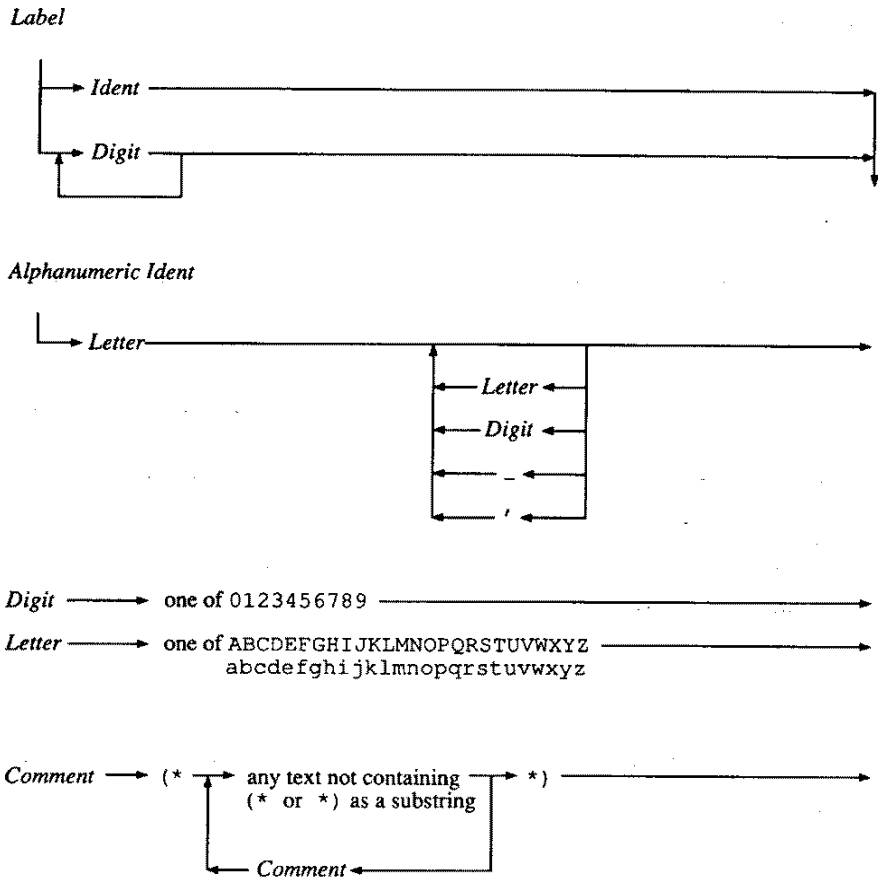

Abbildung B.19: Lexical Matters V

Index

Literaturverzeichnis

[And91] Gregory R. Andrews. *Concurrent Programming, Principles and Practice.* The Benjamin/Cummings Puplishing Company Inc., 1991.

[Har93] Rachel Harrison. *Abstract Data Types in Standard ML.* John Wiley and Sons Ltd., 1993.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[L.C91] L.C.Paulson. *ML for the Working Progammer.* Cambridge University Press, 1991.

[RB88] Philip Wadler Richard Bird. *Introduction to Functional Programming.* Prentice Hall, 1988.

[RM90] Robert Harper Robin Milner, Mads Tofte. *The Definition of Standard ML.* Robin Milner, 1990.

[RM91] Mads Tofte Robert Milner. *Commentary on Standard ML.* Massachusetts Institute of Technology, 1991.

[Ull94] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 1994.

[Wik90] Åke Wikström. *Funktional Programming Using Standard ML.* Prentice Hall, 1990.