14 Caml Light als Beispiel für einen ML-Dialekt

1978 wurde die Sprache ML [MTH90] von R. Milner vorgeschlagen. Die ursprüngliche Motivation war, eine Metasprache für das interaktive Beweissystem LCF zu entwickeln *(Meta Language)*. Als Vorbild diente dabei die Sprache ISWIM (für "If You see What I Mean") von P. Landin [Lan66] aus dem Jahre 1965. Seit 1984 wird die Sprache Caml als ML-Dialekt entwickelt. Die erste Version erschien 1987. Caml Light ist eine kleine extrem portable Implementierung von Caml.
Wir geben in diesem Kapitel eine Kurzeinführung in Caml Light [Ler97]. Als Vorlage dienten hierzu [Mau95] und [Ché95].

14.1 Eigenschaften

Caml zeichnet sich durch folgende Charakteristika aus:

- starkes polymorphes Typsystem
- benutzerdefinierte algebraische Datentypen
- Pattern-Matching
- höhere Ordnung
- Currying
- strikte Auswertung
- veränderbare Variablen (Zuweisung)
- Syntax für Schleifen
- Streams als verzögert ausgewertete Datenstruktur

Gegenüber SML hat Caml ein wesentlich vereinfachtes Modulsystem.

14.2 Funktionsdefinitionen

Funktionen können in Caml über ein `let`-Statement definiert werden. Ausdrücke werden dabei in Caml immer mit einem doppelten Semikolon beendet.

```
let successor = function x -> x+1;;
```

Das Schlüsselwort `function` entspricht dabei der Bildung einer Funktionsabstraktion, entsprechend des `lambda` in Lisp. In einer Funktionsdefinition können die formalen Parameter auch direkt auf der linken Seite des `let`-Konstruktes angegeben werden, so daß eine explizite Bildung mit `function` nicht notwendig ist. Die obige Definition wird somit äquivalent zu:

```
let successor x = x+1;;
```

Wie man sieht gibt es eingebaute arithmetische Funktionen in Infixschreibweise. Funktionen können ungesättigt auf Argumente angewendet werden, so daß die obige Definition auch wie folgt möglich ist:

```
let addition x y = x+1;;
let successor = addition 1;;
```

Ein `let`-Statement kann auch lokal für nur einen Ausdruck erfolgen.

```
let f y=1+y in f 5;;
```

Rekursive Funktionen sind mit `let rec` zu deklarieren. Verschränkte Rekursionen sind allgemein nicht zulässig.

14.3 Eingebaute Typen

Wie jede gängige Sprache stellt auch Caml eine Reihe von Basistypen zur Verfügung. Wir geben sie im folgenden kurz an:

Ganze Zahlen Caml hat die ganzen Zahlen (Typ `int`) mit einem Bereich von $-2^{30} \ldots 2^{30} - 1$ eingebaut. Unter den eingebauten Funktionen für ganze Zahlen sind die Grundrechenarten in Infixschreibweise: `+, -, *, /, mod`

Fließkommazahlen Ebenso sind Fließkommazahlen eingebaut. Für diese sind jedoch andere Funktionen für die Grundrechenarten zu benutzen, da die obigen nur für ganze Zahlen typgerecht angewendet werden dürfen: `+., -., *., /., mod.`.

Bool'sche Werte Für den Typ der bool'schen Werte gibt es zwei Konstanten: `true` und `false`. Funktionen, die einen bool'schen Wert als Ergebnis liefern heißen mitunter auch *Prädikat*. Es gibt natürlich entsprechend für die beiden Zahlendatentypen Vergleichsoperatoren: `<, <=, >, >=` und entsprechend: `<., <=., >., >=.`.
Die Gleichheit ist in Caml überladen und steht für alle Datentypen außer Funktionen zur Verfügung. Ausdrücke die auf Gleichheit getestet werden, müssen allerdings vom gleichen Typ sein.
Das Konditional für bool'sche Werte kann über das syntaktische Konstrukt `if` e_{test} `then` e_1 `else` e_2 gebildet werden. Die Auswertung erfolgt dann in naheliegender Weise.
Die logischen Junktoren werden mit `or` bzw. `&` bezeichnet. Sie sind in ihrer Auswertung nicht-strikt, d.h. das zweite Argument wird nur ausgewertet, wenn der Wert des Gesamtausdruckes nicht aus dem Wert des ersten Arguments allein hervorgeht.

Tupel Datenobjekte können mehrere Unterobjekte in einem Tupel (Paar, Tripel etc.) zusammenfassen. Hierzu wird in Caml einfach das Komma als Tupelbildungskonstruktor verwendet. Es können um Tupel dann noch zusätzlich Klammern gesetzt werden, was aber nicht unbedingt notwendig ist. So lassen sich z.B. folgende Tupelausdrücke bilden:

```
1,2;;
```

```
1,8,54;;
```

```
(1+2,1>4);;
```

Die einzelnen Elemente eines Tupels können per Pattern-Matching selektiert werden.

```
let premier (x,y) = x
and seconde (x,y) = y;;
```

Strings In üblicher Weise stellt Caml auch Strings und Buchstabenzeichen zur Verfügung. Die Stringkonkatenation wird dabei durch die Infixfunktion `^` bezeichnet. Zeichen stehen in einfachen Quotes: `'a'`, `'b'`.
Strings sind keine Listen von Zeichen, wie z.B. in Haskell!

14.4 Algebraische Typen

Listen Wie fast alle funktionale Sprachen stellt auch Caml eingebaute Listen zur Verfügung. Die Listenkonstruktoren sind dabei `[]` für leere Listen und `::` für den üblichen `Cons`-Konstruktor zum Hinzufügen eines Elements zu einer bestehenden Liste. Listenelemente müssen dabei alle vom selben Typ sein. Desweiteren gibt es eine abkürzende Syntax für die Bildung von Listen: `[1;2;3;3]` entspricht `1::2::3::3::[]`.
Es lassen sich so z.B. Listen von Funktionen definieren:

```
let adds =
  let add x y = x+y
  in [add 1; add 2; add 3];;
```

Die Listenkonkatenation ist als die Infixfunktion `@` definiert. Sie entspricht der Funktion `append` in Haskell.

Listenextraktion über Pattern-Matching Eine Fallunterscheidung nach leerer und nicht-leerer Liste und eine Extraktion des Kopfelements und der Restliste kann in Caml über Pattern-Matching realisiert werden. Die zwei Standardlistenfunktionen `length` und `map` haben in Caml dann folgende Gestalt:

```
let rec length = function []   -> 0
                        | _::l -> 1 + length(l);;

let rec map f =
    function []   -> []
           | x::xs-> (f x)::(map f xs);;
```

Benutzerdefinierte algebraische Typen Neben dem vordefinierten algebraischen Datentypen der Listen, können vom Benutzer beliebige algebraische rekursive Datentypen definiert werden. Dabei gibt es die Summenbildung, in der verschiedene Konstruktoren für einen gemeinsamen Datentypen zusammengefaßt werden und die Produktbildung, in der ein Konstruktor definiert wird, der mehrere Argumente hat und quasi das Produkt der Argumenttypen bildet.

Produkttypen Die Produktbildung erlaubt es, mehrere Typen zu einer Struktur, die ein Datenelement beschreiben soll, zusammenzufassen. Ein einfaches Beispiel:

```
type person =
  {Name:string; Age;int; Job:string; City:string};;
```

Damit sind *label* definiert, mit denen auf einzelne Elemente eines Produktes zugegriffen werden kann. Die Reihenfolge der Produktelemente im Gegensatz zu Tupeln ist damit irrelevant. Ein Element des obigen Typs kann dadurch wie folgt definiert werden:

```
let hs = {Job ="TV-Moderator"; City="Koeln";
          Name="Schmidt"     ; Age =43};;
```

Auch auf Produkttypen kann über Pattern-Matching zugegriffen werden.

```
let alter = function
      {Age=n; Name=_;Job=_;City=_} -> n;;
```

Es stehen aber auch automatisch Selektorfunktionen für die einzelnen Felder zur Verfügung:
`hs.Age;;`
Es ist dabei zu beachten, dass verschiedene Produkttypen einen gleichen Feldnamen haben können. Dann wird der Typ gewählt der als letztes definiert wurde. Wie man sieht, wird hier also die Reihenfolge der Deklarationen wichtig.
Schließlich können Produkte noch parametrisiert sein, d.h. der Typ eines Feldes noch offen gelassen werden, so daß für jede mögliche Instanz dieses offenen Typs das Produkt definiert ist.

```
type ('a,'b) pair = {Fst: 'a;Snd: 'b}
```

Summentypen Ein Summentyp stellt eine Vereinigung über mehrere Typen dar. Es ist somit z.B. möglich einen Typen zu definieren, dessen Werte entweder ein Paar zweier ganzer Zahlen oder aber eine Zeichenkette darstellen. Mit Konstruktoren wird dabei as eine Art *tag* unterschieden, um welchen der beiden Grundtypen es sich bei einem Datenobjekt handelt. Dieses läßt sich in Caml wie folgt darstellen:

```
type identification = Name of string
                    |SS of int * int::
```

Damit wurden zwei Konstruktoren definiert: `Name` und `SS`. Mit diesen Konstruktoren lassen sich Objekte des neuen Datentyps `identification` bilden:

```
let id1 = Name "Schmidt";;

let id2 = SS (345356,456465);;
```

Die Werte `id1` und `id2` haben denselben Typ.
Auch für benutzerdefinierte algebraische Datentypen steht Pattern-Matching zur Verfügung:

```
type suit = Heart
          | Diamond
          | Club
          | Spade;;

let coleur = function Diamond -> "rouge"
                    | Heart   -> "rouge"
                    | _       -> "noir";;
```

Wie von dem eingebauten algebraischen Typ der Listen bekannt, kann auch ein benutzerdefinierter algebraischer Typ rekursiv und in einigen Subtypen parametrisiert sein. Ein Beispiel:

```
type ('a,'b) arbre = Leaf of 'b
                   | Branche of ('a,'b) node
and ('a,'b) node = {Op: 'a;
                    Son1: ('a,'b) arbre;
                    Son2: ('a,'b) arbre};;
```

14.5 Destruktive Operationen

Caml erlaubt es Datentypen als veränderbar zu deklarieren, d.h. ein Objekt eines solchen Datentyps darf physikalisch durch eine destruktive Operation verändert werden. Dieses entspricht einer imperativen Programmierung, in der Speicherplätze durch Befehlssequenzen verändert werden.

Records Stellen wir uns vor, wir wollen einen Produkttypen für Personen wie im letzten Abschnitt haben, es soll jedoch möglich sein, dass sich die Daten einer Person während eines Programmdurchlaufs verändern. Hierzu sind de veränderbaren Felder eines Summentyps mit dem Schlüsselwort `mutable` zu kennzeichnen:

```
type personne =
     {Name: string; mutable Age : int;
      mutable Job: string; mutable City: string}
```

Es lassen sich weiterhin Objekte dieses Typs wie zuvor definieren:

```
let lucile =
    {Name="Lucile"; Age= 23; Job="Studente";City="Paris"};;
```

Dieses Objekt läßt sich aber nun in den mit `mutable` gekennzeichneten Feldern physikalisch verändern. Erhöht sich das Alter von Lucile, so können wir das mit folgenden Befehl erreichen:

```
lucile.Age <- lucile.Age+1;;
```

Solch ein Befehl hat im eigentlichen Sinne keinen Wert wie ein Ausdruck normalerweise. Es kommt hierbei nicht auf den Wert des Ausdrucks an, sondern auf seinen Seiteneffekt. Der obige Ausdruck selbst gibt nur den Wert `()` des zusätzlichen eingebauten Typen `unit` zurück.
Für Produkte aus nur einem Typ gibt es eine abkürzende Definition für veränderbare Daten:

```
let zaehler = ref 0;;
```

Hierbei wird implizit der folgende Typ definiert:

```
type intreference = {mutable Ref: int}
```

und ein Objekt dieses Typs definiert:

```
let zaehler = {Ref = 0};;
```

Auf den Wert eines Objekts vom Typ `ref` wird mit einem Ausrufezeichen zugegriffen:

```
!zaehler + 1;;
```

Durch solch einen Zugriff wird das Objekt nicht verändert.
Objekte des Typs `ref` werden mit dem Infix-Konstrukt `:=` verändert.

```
zaehler := !zaehler+1
```

Obiger Ausdruck ist wieder vom Typ `unit`. Es ist nicht der eigentliche Wert des Ausdrucks interessant, sondern sein Seiteneffekt.

Felder Als ein besonderer destruktiv veränderbarer Datentyp stehen Felder zur Verfügung. Sie sind parametrisiert über einen Typ. Die Syntax lehnt sich der Listensyntax an:

```
let a = [| 10; 20; 45 |];;
```

Die Feldelemente sind von 0 für das erste Element an indiziert. Über eine Punktnotation kann auf das Element eines bestimmten Indexes zugegriffen werden:

```
# a.(0);;
-: int = 10
```

Ein Zugriff auf ein Feldelement geschieht in konstanter Zeit.
Zur Modifikation eines Feldelementes steht ein eigenes syntaktisches Konstrukt zur Verfügung:

```
a.(1)<- 14;;
```

Mit dem Funktionsaufruf `make_vect k x` wird ein Feld der Länge `k` mit den initialen Elementen `x` erzeugt.

Sequenzen und Schleifen Da mit destruktiven Befehlen, die eine strenge Sequentialisierung des Programmablaufes erfordern ein klassisches Merkmal der imperativen Programmierung in Caml zur Verfügung steht, ist es auch sinnvoll de klassischen Konstrukte zur sequentiellen Abarbeitung von Befehlen zur Verfügung zu stellen:

- Sequenzoperator: wie aus gängigen imperativen Sprachen gewohnt, wird die Sequenz zweier Ausdrücke mit einem Semikolon notiert. Der Wert einer Sequenz zweier Ausdrücke ist der Wert des zweiten Ausdrucks. Der Wert des ersten Ausdrucks wird nicht berücksichtigt. (Trotzdem wird der Ausdruck ausgewertet. Caml ist strikt, und auch die Sequenz ist natürlich als eine strikte Projektionsfunktion auf das zweite Argument zu betrachten.)
- Schleifen: In Caml stehen zwei Schleifenkonstrukte zur Verfügung: `while` und `for`.

Ein Beispiel für eine einfache Befehlssequenz:

```
let x = ref 1 in
  begin
    x := !x+1;
    x := !x * !x;
    !x
  end;;
```

Die Schlüsselwörter `begin` und `end` sind dabei nur ein Synonym für eine öffnende und eine schließende Klammer.
Die Syntax und Auswertungs von Schleifenkonstrukten geht konform mit den aus der imperativen Programmierung bekannten Konstrukten:

```
let a = ref 0
and n = ref 4;;

while !n>0 do
  a:=!a + !n;
  n:=!n - 1;
 done;
!a;;
```

Für die `for`-Schleife stehen zwei Varianten zur Verfügung:

```
let a = ref 0
and n = ref 4;;

for i=1 to !n do a:=!a + i done;!a;;
```

Und eine entsprechende Variante zum Abwärtszählen eines Index:

```
let a = ref 0
and n = ref 4;;

for i = !n downto 1 do a:=!a + i done;!a;;
```

Die Schleifenkonstrukte sind dabei natürlich reiner syntaktischer Zucker, da sie sich auch jeweils durch rekursive Funktionen ausdrücken lassen. Allerdings scheint es eine gute Technik zu sein, bei Operationen mit Seiteneffekten auf die Schleifenkonstrukte und die `begin-end`-Klammerung zurückzugreifen, so dass auch syntaktisch sich imperative und rein funktionale Programmteile voneinander abheben.

14.6 Behandlung von Ausnahmen

In Caml ist eine eigene Struktur zur Behandlung von Ausnahmen oder auch Fehlermeldungen integriert. Solche Ausnahmen können geworfen werden, wenn ein unspezifizierter Fall auftritt z.B. ein Element aus einer leeren Liste extrahiert und an späteren Zeitpunkten wieder abgefangen werden soll, so dass in Falle, wenn dieser Fehler auftritt, eine bestimmte Funktion aufgerufen wird. Hierzu ist der Wert `exeption` für Ausnahmen in Caml eingebaut. Eine Ausnahme ist dabei ähnlich zu einem Summentypen aufgebaut: sie hat einen Namen entsprechend einem Konstruktor, und 0 oder ein Argumente, die genauer beschreiben können, um was für einen Fehler es sich gehandelt hat. Für die Extraktion eines Elements einer leeren Liste läßt sich somit eine Ausnahme in folgender Weise werfen:

```
let head = function
      x::_  -> x
     | []   -> raise (Failure "head: empty list");;
```

Hier wird die eingebaute Ausnahme `Failure` mit einer Zeichenkette als Argument geworfen. Es ist aber auch vom Benutzer möglich, eigene Ausnahmen zu definieren:

```
exception Found of int;;
```

Wie wir schon gesehen haben, werfen wir Ausnahmen mit der Funktion `raise`. Sie hat ein Argument, nämlich gerade die Ausnahme, und kann als Ergebnis mit jedem Typ unifiziert werden. Im obigen Beispiel ist das Ergebnis von `raise` eine Liste. Damit stellt `raise` in ihrem Typ eine außergewöhnliche Funktion dar, denn das Ergebnis ist immer nur im Kontext als typisiert zu verstehen, und hängt nicht vom Typ des Arguments ab.
Ausnahmen können natürlich nicht nur geworfen werden, sondern auch wieder abgefangen werden. Hierzu steht die Funktion `try` zur Verfügung.
`try` *expression* `with` *match-cases*
Bei der Auswertung des `try`-Konstrukts wird zunächst der Ausdruck *expression* ausgewertet. Wenn dieser zu einer Ausnahme evaluiert, die auf einen der *match-cases* passt, dann wird der darin spezifizierte Ausdruck als Ergebnis zurückgegeben, ansonsten ist das Ergebnis von `try` der Ausdruck *expression*.

```
let find_index p l =
  let rec find n =
        function []   -> raise (Failure "not found")
```

```
              | x::l -> if p(x) then raise (Found n)
                        else find (n+1) l
 in
  try find 1 l with Found n -> n;;


find_index (function n -> (n mod 2) = 0 ) [1;3;5;7;9;10];;
```

14.7 Ein- und Ausgabe

Ein- und Ausgabe wird in Caml durch Kanäle in einer imperativen Manier realisiert. Eine IO-Operation ist als ein seiteneffektbehafteter Befehl zu betrachten. Die Reihenfolge der IO-Operationen ist daher wieder wichtig. Alle IO-Operationen lesen bzw. schreiben auf IO-Kanäle. Drei solcher Kanäle sind dabei vordefiniert: `std_in, std_out, std_err`.
Für bestimmte Typen sind Lese- und Schreiboperationen auf IO-Kanälen eingebaut. Solche Typen werden als *printable* bezeichnet. Sie umfassen: `char, string, int` und `float`.

Ausgabe Zum Schreiben auf `std_out` stehen folgende Funktionen zur Verfügung:
`print_char, print_string, print_int, print_float`. Da die Ausgabe auf `std_out` gepuffert wird, gibt es eine Funktion, die die Ausgabe der Zeichen im Puffer bewirkt: `flush`. Alle bisher vorstellten IO-Operationen haben als Ergebnistyp `unit`, d.h. ihr Ergebnis ist nicht interessant sondern ihr Seiteneffekt.
Entsprechend stehen auch Funktionen zur Ausgabe von druckbaren Typen auf `std_err` zur Verfügung.

Eingabe Zum Lesen von `std_in` stehen in naheliegender Weise auch primitive Funktionen zur Verfügung:
`read_int, read_float, read_line`.

Dateien als Kanäle Um Ein- und Ausgabe auch auf Dateien durchführen zu können, müssen diese als Kanäle geöffnet und anschließend wieder geschlossen werden. Die Funktionen zum Öffnen einer Datei geben als Ergebnis einen neuen Kanal zurück, die zum Schließen haben angewendet auf einen Kanal den Typ `unit`. Die entsprechenden Funktionen heißen:
`open_in, open_out, close_in, close_out`.
Schließlich stehen Funktionen zum Lesen und Schreiben auf beliebigen Kanälen zur Verfügung:
`output_char, output_string, input_char, input_line`.
Die IO-Operationen werfen Ausnahmen, wenn sie nicht durchgeführt werden können.

14.8 Streams

Als ein besonderer Datentyp, der in eingeschränkter Form eine Programmierung wie in verzögert ausgewerteten Sprachen erlaubt, stellen Streams dar. Streams sind dabei als unendliche Listen zu verstehen, die nur auf Verlangen elementweise ausgewertet werden.
Wie wir sehen werden, sind die Streams in Caml aber auch sehr imperativ, die sie durch Seiteneffekte physikalisch verändert werden.

Syntax Streams lehnen sich syntaktisch zunächst sehr an Listen an:

- ein leerer Stream:
  `[<>];;`
- ein endlicher Stream: Die einzelnen Stream-Elemente werden durch ein `'` markiert:
  `[< '0; '1; '2>];;`
- Streamsegmente, die kein einzelnes Element darstellen, werden ohne das vorangestellte `'` markiert:
  : `[< '0; [<'1; '2>]; '3>];;`
  Damit steht auch eine Syntax für die Listenkonkatenation zur Verfügung:
  `let stream_concat s t = [<s; t>];;`
- im Zusammenhang mit I/O lassen sich Eingabekanäle in Streams umwandeln, durch de Funktion `stream_of_channel`.
- ebenso lassen sich Strings durch die Funktion `stream_of_string` in Streams umwandeln.

Auswertung von Streams Man kann sich durch einen kleinen Test davon überzeugen, daß Streams verzögert ausgewertet werden. Hierzu bedient man sich einer `print`-Anweisung, die als Seiteneffekt ausgibt, wann bestimmte Ausdrücke ausgewertet werden[1]:

```
let rec ints_from n =
    [< '(print_int n; print_char ` `;flush std_out; n)
    ;  ints_from (n+1) >];;

let ints = ints_from 0;;
```

Der Interpretierer terminiert bei obiger Eingabe die Auswertung bereits, ohne daß irgendetwas gedruckt wurde. Es hat nämlich noch keine äußere Funktion die Auswertung von `ints` bis zu einem gewissen Grade gefordert. Im folgenden sehen wir, wie die Auswertung von Streams erzwungen werden kann.

[1] Man beachte in der folgenden Funktion auch, daß die Streamelemente den Typ `int` haben, obwohl ein Streamelement aus einer Folge von drei Anweisungen und einem Ausdruck gebildet wird.

Destruktives Pattern-Matching auf Streams Auf das erste Element eines Streams kann durch Pattern-Matching zugegriffen werden. Hierzu ist im Pattern ein Variablenname für das Element einzuführen. So läßt sich die Funktion, die das erste Element eines Streams selektiert, wie folgt definieren:

```
let next = function [< 'x >] -> x;;
```

Es ist hierbei zu beachten:

- Auch im Pattern-Matching sind Variablen für Streamelemente mit einem ' zu markieren.
- Für den Reststream muß keine Variable, noch nicht einmal ein Wildcard angegeben werden. Das obige Pattern matcht auch für Streams mit mehr als einen Element.

Das Pattern-Matching auf Streams hat aber nicht nur den Effekt der Selektion einer Alternative, sondern bewirkt zusätzlich noch einen Seiteneffekt. Hierzu wenden wir die obige Funktion mehrmals auf einem Stream an:

```
#let s = [< '0; '1; '2>];;

#next s;;
-: int = 0

#next s;;
-: int = 1

#next s;;
-: int = 2

#next s;;
Uncaught exception: Parse_failure
```

Wie zu sehen ist, ist der Aufruf der Funktion `next` nicht referenziell transparent. Die sukzessiven identische Aufrufe werden zu verschiedenen Ergebnissen ausgewertet. Dieses liegt daran, daß das Pattern-Matching in der Funktion `next` das Argument verändert. Streams sind also globale Variablen, auf denen ein Pattern-Matching eine destruktive Operation darstellt. Dieses ist nicht nur eine seiteneffektbehaftete Operation, sondern in dem Sinne auch eine sehr üble, daß sie nicht deutlich im Programmtext markiert ist. Das Pattern-Matching auf Streams verhält sich fundamental anders, als das Matching von Listen.
Das Pattern-Matching auf Streams ist nun aber nicht nur beschränkt auf den einfachen Zugriff auf Elemente, sondern kann zusätzlich noch eine Anwendung einer Funktion auf den Streams beinhalten:

```
let seq_a =
  let rec seq = function
      [< ''a'; seq l>] -> 'a'::l
     |[< >] -> []
  in function   [< ''a'; seq l>] -> 'a'::l
```

In dieser Funktion wird zunächst unterschieden, ob der Stream ein erstes Element hat, das der Buchstabe 'a' ist. Ist dieses der Fall, so wird die lokale Funktion `seq` auf den Reststream angewendet. Diese Anwendung verändert den Reststream! Wenn auf der rechten Seite der Funktionsdefinition auf die Variable `l`, die ursprünglich den Reststream gekennzeichnet hat, zugegriffen wird, so ist diese dann an das Ergebnis von der Anwendung der Funktion `seq` auf den Reststream gebunden; d.h. die Variable `l` bezeichnet auf der linken und auf der rechten Seite der Funktionsdefinition sogar Ausdrücke verschiedenen Typs. Links einen Stream und rechts eine Liste!

14.9 Beispielprogramme

Ein globaler Keller Wir geben im folgenden ein Beispiel für ein Modul, das einen globalen Keller als abstrakten Datentypen realisiert. Die einfachste Umsetzung eines Kellers wäre natürlich die Benutzung der Listenstruktur. Wir wollen hingegen einen Keller zur Verfügung stellen, der etwas weniger speicheraufwendig ist als eine Struktur, in der jedes Kellerelement mit einem eigenen Zeiger and den Rest des Kellers gebunden ist. Wir wollen den Keller vielmehr in Teilstücke, die als Felder im Speicher vorliegen, implementieren.
Zunächst definieren wir die Schnittstellenbeschreibung des abstrakten Datentyps Keller. Hierzu ist in Caml Light eine `*.mli` Datei zur Schnittstellenbeschreibung zu definieren. Wir schreiben die Datei `Piles.mli`:

```
(* Der abstrakte Datentyp kennt eine neue Ausnahme f\"ur den Zugriff
   auf einen leeren Keller. Solche Ausnahme werden in der
   Schnittstellenbeschreibung definiert *)

exception Pile_Vide;;

(* Schlie{\ss}lich besteht der abstrakte Datentyp Keller aus
   vier Funktionen *)

value cre_pile : 'a -> int
  -> (unit -> 'a) * ('a -> unit) * (unit -> 'a) * (unit -> bool);;
```

Damit stellt das Modul eine Funktion zum Erzeugen eines Kellers zur Verfügung: `cre_pile`. Sie hat zwei Argumente: Ein Argument, das benötigt wird, um den Typ der Kellerelemente zu bestimmen und ein zweites Argument, das angibt, wie groß die einzelnen Felder des Keller sind, in denen die Elemente nicht verzeigert gespeichert werden.
Das Ergebnis dieser erzeugenden Funktion sind die Funktionen, die der abstrakte Datentyp zur Verfügung stellt. In unserem Fall genau vier: eine zum Löschen des obersten Kellerelements, eine zum Einfügen eines neuen Kellerelements, eine zur Ausgabe des obersten Kellerelements, ohne dabei den Keller zu verändern und schließlich eine Funktion zum Test, ob der Keller leer ist.
Setzen wir jetzt das Modul für diese Schnittstelle um, d.h. wir implementieren die Funktion `cre_pile` im Modul `Piles.ml`:

```
let cre_pile x k =
```

Der Aufruf mit einer negativen Größe der Felder des Kellers führt zu einer Fehlermeldung:

```
  if k <= 0
     then
      failwith "cre_pile ben\"otigt eine positive Blockgr\"o{\ss}e"
```

Ansonsten werden drei globale veränderbare Datenobjekte erzeugt: eine Liste von Feldern, in denen der Keller gespeichert ist, eine Zahl, die angibt, wieviel Felder der Keller enthält und schließlich den aktuellen Index im obersten Feld für das oberste Element.

```
     else
      let liste_des_blocs_allous = ref ([])
      and nb_blocs_rservs       = ref 0
      and indice_dans_bloc        = ref (k-1)
      in
```

Wir schreiben zunächst die Funktion, die ein Element aus dem Keller extrahiert. Hier muß zunächst geprüft werden, ob der Keller leer ist, und gegebenenfalls eine Ausnahme geworfen werden:

```
      let dpilage () =
          if !nb_blocs_rservs = 0 then raise Pile_Vide
```

ansonsten ist im obersten Feld das oberste Element zu extrahieren und der Keller entsprechend um dieses Element zu verringern:

```
          else
          begin
           let a = (hd !liste_des_blocs_allous).(!indice_dans_bloc)
           in
           if !indice_dans_bloc = 0
           then
            begin
             indice_dans_bloc        := k-1;
             nb_blocs_rservs       := !nb_blocs_rservs-1;
             liste_des_blocs_allous := tl !liste_des_blocs_allous
            end
           else indice_dans_bloc     := !indice_dans_bloc-1;
           a
          end
```

Die Funktion, die den Keller nicht verändert, sondern nur das oberste Element angibt, läßt sich entsprechend einfach umsetzen:

```
and consultation () =
    if !nb_blocs_rservs=0 then raise Pile_Vide
    else (hd !liste_des_blocs_allous).(!indice_dans_bloc)
```

Die Funktion zum Einfügen eines neuen Kellerelements läuft analog zur Funktion zum Löschen eines Kellerelements:

```
and empilage a =
if !indice_dans_bloc <k-1 then
 begin
   indice_dans_bloc            := !indice_dans_bloc +1;
   (hd !liste_des_blocs_allous).(!indice_dans_bloc) <- a
 end
else
 begin
  let nouveau_bloc = make_vect k x
  in
  nb_blocs_rservs       := !nb_blocs_rservs+1;
  liste_des_blocs_allous
     := nouveau_bloc :: !liste_des_blocs_allous;
  indice_dans_bloc       := 0;
  nouveau_bloc.(0)        <- a
 end
```

Schließlich definieren wir noch den Test, ob der Keller leer ist:

```
and est_vide () = !nb_blocs_rservs = 0
```

laut der Spezifikation in der Schnittstellenbeschreibung gibt die Funktion `cre_pile` ein 4-Tupel von Funktionen zurück. Dieses 4-Tupel ist nun noch zu spezifizieren:

```
in (dpilage, empilage, consultation, est_vide);;
```

Jetzt können wir, wenn dieses benötigt wird, einen Keller erzeugen. Hierzu betrachte man den folgenden Dialog im Caml Light Interpretierer.

```
#exception Pile_Vide;;
Exception Pile_Vide defined.
#include "Piles.ml";;
cre_pile :
 'a -> int -> (unit -> 'a) * ('a -> unit)
             * (unit -> 'a) * (unit -> bool) = <fun>
- : unit = ()
#let (pop,push,scan,leer)=cre_pile 1 5;;
pop : unit -> int = <fun>
push : int -> unit = <fun>
scan : unit -> int = <fun>
```

```
leer : unit -> bool = <fun>
#leer();;
- : bool = true
#do_list push [1;2;3;4;5;6;7;8;9;354;345;7;12;54];;
- : unit = ()
#leer();;
- : bool = false
#pop();;
- : int = 54
#let (po,pu,sc,le) = cree_pile "ert" 5;;
po : unit -> string = <fun>
pu : string -> unit = <fun>
sc : unit -> string = <fun>
le : unit -> bool = <fun>
#le();;
- : bool = true
#pu "rtzu";;
- : unit = ()
#po();;
- : string = "rtzu"
#scan();;
- : int = 12
#scan();;
- : int = 12
```

Word-Count Im folgenden findet sich ein Programm, das ähnlich wie das Programm `wc` auf Unix die Anzahl der Zeichen, Wörter und Zeilen einer spezifizierten Datei angibt. Das ist ein Beispiel für eine relativ imperative Programmierung in Caml.
Wir deklarieren zunächst drei globale Variablen, die als Zähler für die bereits gefundenen Zeichen, Wörter und Zeilen fungieren. Sie sind initial auf 0 gesetzt:

```
let chars = ref 0
and words = ref 0
and lines = ref 0
;;
```

Ein simpler algebraischer Typ wird definiert, mit dessen Hilfe gekennzeichnet wird, ob gerade Zeichen innerhalb eines Wortes gelesen werden:

```
type state = Inside_word | Outside_word;;
```

Schließlich die Hauptfunktion, die aus einem Eingabekanal alle Zeichen liest und die globalen Zähler hochzählt:

```
let count_channel in_channel =
```

Eine lokale Hilfsfunktion wird deklariert, in der die freie Variable `in_channel` aus der globalen Funktion `count_channel` referenziert wird:

```
  let rec count status =
```

Lese das nächste Zeichen:

```
    let c = input_char in_channel in
```

Erhöhe den globalen Zeichenzähler um 1:

```
    incr chars;
```

Teste, ob das Zeichen ein Zeilenende oder ein Wortende markiert. Zähle entsprechend die globalen Zähler hoch und rufe die Funktion anschließend rekursiv auf.

```
    match c with
      `\n` ->
        incr lines; count Outside_word
    | ` ` | `\t` ->
        count Outside_word
    | _ ->
        if status = Outside_word then begin incr words; () end;
        count Inside_word
```

Fange in der globalen Funktion zunächst den geworfenen Fehler des Dateiendes (Ende des Eingabekanals) ab, um entsprechend mit dem Typ `unit` die Funktion zu beenden:

```
  in
    try
      count Outside_word
    with End_of_file ->
      ()
;;
```

Es folgt die Funktion, die die spezifizierte Datei als Eingabekanal öffnet und auf diesen Kanal die Funktion `count_channel` anwendet:

```
let count_file name =
  let ic = open_in name in
  count_channel ic;
  close_in ic
;;
```

Schließlich wird eine Funktion definiert, die das Ergebnis aus den globalen Variablen auf die Standardausgabe druckt:

```
let print_result () =
  print_int !chars; print_string " characters, ";
  print_int !words; print_string " words, ";
  print_int !lines; print_string " lines";
  print_newline()
;;
```

Es folgt in imperativer Manier die Sequenz der beiden obigen Funktionen:

```
let count name =
  count_file name;
  print_result ()
;;
```

Als letztes folgt ein Ausdruck, der als Gesamtergebnis des Programms auszuwerten ist. Dies entspricht der Definition einer Funktion `main` in Haskell oder auch C.
Hier wird das Kommandozeilenargument des Programmaufrufs gelesen, um für alle Argumente die Funktion `count` aufzurufen. Ist kein Argument auf der Kommandozeile angegeben, dann erfolgt der Aufruf von `count_channel` auf `std_in`.

```
if sys__interactive then () else
try
  if vect_length sys__command_line < 1 then
    count_channel std_in
  else
    for i = 1 to vect_length sys__command_line - 1 do
      chars:=0;words:=0;lines:=0;
      count sys__command_line.(i)
    done;
with sys__Sys_error s ->
  print_string "I/O error: ";
  print_string s;
  print_newline()
;;
```

Parser Als Beispiel für die Benutzung von Streams, soll in diesem Abschnitt gezeigt werden, wie sich Parser in Caml programmieren lassen.[2]
Ein Beispiel für einen sehr einfachen Parser haben wir bereits bei der Einführung der Streams kennengelernt. Die Funktion `seq_a` hat dort getestet, ob ein Stream mit einem oder mehreren Zeichen ‘a‘ anfängt und im Erfolgsfall eine Liste mit einer entsprechenden Anzahl ‘a‘ als Ergebnis gehabt. ist das erste Streamelement kein ‘a‘, dann matcht keine Alternative und es wird eine Ausnahme geworfen. Betrachten wir jetzt eine ähnliche Parserfunktion, die ohne eine Ausnahme zu

[2] Hier soll nicht verschwiegen werden, daß es in der Caml Light Umgebung auch Versionen von `lex` und `yacc`, die entsprechenden ML-Code generieren, gibt.

werfen endet, wenn der Eingabestream mit einer Sequenz von Zeichen 'a' gefolgt von einem 'b' beginnt:

```
let seq_a_b = function
  [<seq_a l; ''b'>] -> l@['b']
 |[<''b'>]          -> ['b']
```

Hier wird die Natur des Pattern-Matchings auf Streams noch einmal besonders deutlich. Zum Matchen der ersten Alternative wird zunächst die Funktion `seq_a` auf den Eingabestream angewendet. Das Ergebnis wird eine Liste sein, sofern keine Ausnahme geworfen wurde. Ergibt die Anwendung eine Ausnahme, so wird die nächste Alternative versucht zu matchen. Dabei ist dann der komplette Effekt der Anwendung von `seq_a` auf dem Stream wieder zurückgesetzt worden. Das Pattern-Matching enthält also auch eine verstecktes Backtracking. Wurde aber die Anwendung von `seq_a` ohne Ausnahme abgeschlossen, so wird auf dem Reststream, der bei dieser Anwendung noch nicht ausgewertet wurde, getestet, ob er mit einem 'b' startet. Ist dieses der Fall, so wird als Ergebnis die Liste der aus dem Stream gelesenen Zeichen ausgegeben.
Sollte schließlich keine der zwei Alternativen Matchen, so bleibt der Eingabestream unverändert und es wird eine Ausnahme geworfen.
Da Streampatterns streng sequentiell von rechts nach links ausgewertet werden, kann man sogar eine Variable mehrfach auf der linken Seite einer Funktionsdefinition stehen:

```
let rec stream_sum n = function
  [< '0;(stream_sum n) p >] -> p
| [< 'x;(stream_sum (n+x)) p -> p
| [< >] -> n;;

streams_sum 0 [<'0;'1;'8;'7>];;
```

Auch diese Funktion erscheint insbesondere Haskellprogrammierern zunächst sehr verwirrend. Das Ergebnis `p` in den ersten zwei Alternativen ist jeweils an den rekursiven Funktionsaufruf im Pattern gebunden. In der zweiten Alternative, wird im Pattern erst das erste Streamelement an die Variable `x` gebunden, und kann anschließend im weiteren Verlauf des Patterns referenziert werden.
Versuchen wir jetzt einen Parser zu schreiben, der einfache arithmetische Ausdrücke aus einem Stream von Token parst. Hierzu definieren wir zunächst den algebraischen Typ der Token:

```
type token =
  PLUS|MINUS|TIMES|DIV|LPAR|RPAR|Int of int;;
```

Das Ergebnis eines Parses soll ein entsprechender Syntaxbaum des arithmetischen Ausdruckes sein. Hierzu definieren wir auch einen entsprechenden algebraischen Typ:

```
type atree =
 Int of int
|Plus of atree * atree
|Minus of atree * atree
|Mult of atree * atree
|Div of atree * atree;;
```

Wir müssen nun noch spezifizieren, nach welcher Grammatik wir parsen wollen:

$$Expr \rightarrow Mult\ RestExpr$$

$$\begin{aligned} RestExpr \rightarrow & + Mult\ RestExpr \\ & \mid - Mult\ RestExpr \\ & \mid \epsilon \end{aligned}$$

$$Mult \rightarrow Atom\ RestMult$$

$$\begin{aligned} RestMult \rightarrow & * Atom\ RestMult \\ & \mid / Atom\ RestMult \\ & \mid \epsilon \end{aligned}$$

$$\begin{aligned} Atom \rightarrow & Int \\ & \mid (Expr) \end{aligned}$$

Tatsächlich läßt sich eine solche Grammatik direkt in eine Parser umsetzen.
Zunächst die Funktion zum Parsen der Strichrechnung:

```
let rec expr =
  let rec restexpr e1 = function
      [< 'PLUS;  mult e2; restexpr (Plus  (e1,e2)) e >] -> e
    | [< 'Minus; mult e2; restexpr (Minus (e1,e2)) e >] -> e
    | [< >] -> e1
in function [< mult e1; (restexpr e1) e2>] -> e2
```

Einen Ausdruck zu parsen bedarf nach unserer Grammatik zunächst eines erfolgreichen Parses der Punktrechnung. War dies erfolgreich, so binden wir im entsprechenden Pattern das Ergebnis an die Variable `e1`, die wir dann auch als Argument im weiteren Verlauf des Patterns referenzieren. Wir wenden auf den noch nicht abgearbeiteten Reststream den Parser `restexpr` an, wobei wir das Ergebnis des bereits erfolgten Parses als erstes Argument mitgeben.
Zum Parsen des Nichtterminals *RestExpr* wird zunächst nach den Terminalsymbol einer Strichrechnung gesucht, um dann entsprechend der Grammatik weitere Parser auf den Rest anzuwenden. Im ersten Argument wird dabei das schon geparste Ergebnis akkumuliert.
Entsprechend sieht die Funktion zum Parsen der Strichrechnung aus:

```
and mult =
    let rec restmult e1 = function
      [< 'TIMES; atom e2; restmult (Mult (e1,e2)) e >] -> e
    | [< 'DIV  ; atom e2; restmult (Div  (e1,e2)) e >] -> e
    | [< >] -> e1
in function [< atom e1; (restmult e1) e2>] -> e2
```

Schließlich ist noch der einfache Parser für das Nichtterminalsymbol *Atom* zu implementieren:

```
and atom = function
  [< 'INT n >] -> Int n
| [< 'LPAR; expr e; 'RPAR >] -> e;;
```

14.10 Kurzeinschätzung von Caml und SML ähnlichen Sprachen

Vorteile:

- Typsystem, Pattern-Matching und höhere Ordnung erlauben einen weitgehendst funktionalen Programmierstil in einem hohen Abstraktionsgrad.
- Einfaches I/O-Konzept.
- Weit verbreitet und viele größere Anwendungen bekannt, z.B. MMM ein Webbrowser, der Caml-Applets interpretieren kann.
- Programme lassen sich durch direktes Update von Datenstrukturen optimieren.
- Ausnahmebehandlung vereinfacht Programmierung.
- Durch Einfügen von Printanweisungen steht eine einfache Möglichkeit zum Debuggen zur Verfügung.

Nachteile:

- nicht referenziell transparent. Damit lassen sich nicht so einfach Programmtransformationen durchführen.
- Streams sind umständlich und nicht zu vergleichen mit einer Programmierung in einer verzögert ausgewerteten Sprache.
- Ein imperativer Programmierstil mit viel Seiteneffekten kann Programme schwer lesbar und verifizierbar machen.
- Kein Mechanismus zum Überladen von Funktionen vorhanden.
- Rekursionen sind explizit zu deklarieren. Keine allgemeine verschränkte Rekursion über mehrere Funktionen möglich.

14.11 Bemerkungen zu Programmtransformationen

Programmtransformationen im Sinne von KFP+ kann man in strikten funktionalen Programmiersprachen nur sehr eingeschränkt durchführen. Im allgemeinen muss bei Transformationen die Terminierung gesondert betrachtet werden.

Beispiel 14.1. `let funktionsif = function x y z ->  if x then y else z;;`

```
let rec botf    = function x -> botf x;;
let bot         = botf 1;;
let test        = functionsif True 1 bot;;
```

Die Auswertung von `test`terminiert nicht. Die Transformation der Definitionseinsetzung ergibt für test:

```
test = if True then 1 else bot;;
```

Danach terminiert `test`. Damit ist die Semantik des Programms verändert worden.
Die Veränderung durch diese Transformation in seiteneffektfreien Programmen betrifft (vermutlich) "nur" eine Veränderung bzgl Terminierung. Allerdings können auch Veränderungen bzgl. des Fehlerverhaltens (errors, exceptions) auftreten.

Beispiel 14.2. Wenn man Seiteneffekte erlaubt, dann kann sich auch der Wert verändern:

```
let doppelt    = function x -> x+x;;
let increase   = function x -> x := !x +1;;
let testa = ref 1;;
let test  = doppelt (increase testa);;
```

Die Variable `test` wertet zu 4 aus.
Nach einer Programmtransformation mittels Beta-Reduktion für `doppelt` ergibt sich:

```
 let test  =  (increase testa) + (increase testa);;
```

Offensichtlich ergibt die Auswertung jetzt einen andern Wert: $5 = 2 + 3$, da der Aufruf von `(increase testa)` verdoppelt wurde.
Eine Variante der Beta-Reduktion, die auch für Seiteneffekte funktioniert, ist die Vermeidung des expliziten Kopierens:

```
let test  = let x = (increase testa) in x + x;;
```

Es ergibt sich wieder der richtige Wert: 4.
Programmtransformationen bei Anwesenheit von Seiteneffekten muss i.a. die genaue Sequenz der Seiteneffektoperationen erhalten.

Literatur

[Ché95] Laurent Chéno. *Une boîte d'outils pour programmation en Caml.* Lycée Louis-Le-Grand, Paris, 1995.

[Lan66] P. Landin. The next 700 programming languages. In *Communications of the ACM*, volume 9, pages 157–164, 1966.

[Ler97] Xavier Leroy. The Caml Light system release 0.73. Institut National de Recherche en Informatique et Automatique, 1 1997.

[Mau95] Michel Mauny. *Functional programming using Caml Light*, 1 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* IT Press, Cambridge, Massachusetts, 1990.