Rheinland-Pfalz

Materialien

zum

Lehrplan Informatik

Grundfach

in der Oberstufe des Gymnasiums

(Mainzer Studienstufe)

Funktionale Programmierung

Landesmedienzentrum

Satz:
Gregor Noll, Erpel am Rhein

Druck und Vertrieb:
Landesmedienzentrum Rheinland-Pfalz
Hofstraße 257c, D-56077 Koblenz-Ehrenbreitstein
Telefon (0261) 97 02-3 00
Telefax (0261) 97 02-3 62

Mai 1999

Vorwort

Das hier vorgelegte Material zur „Funktionalen Programmierung” erschließt einen völlig andersartigen Zugang zur Computerprogrammierung als er bisher im traditionell an einer imperativen Sprache orientierten Informatikunterricht üblich ist.

Funktionen lassen sich in unserer Umwelt in vielfältigen Aspekten tagtäglich entdecken. Sie sind Schülerinnen und Schülern – nicht nur aus der Mathematik – bestens bekannt. Funktionales Denken, als Erfassen und Verstehen von funktionalen Abhängigkeiten in den unterschiedlichsten Kontexten, stellt eine wichtige Komponente zur Bewältigung unseres Lebens dar.

So ist es nicht verwunderlich, dass sich auch die Informatik dieser Denkweise bedient, um mit ihrer Hilfe Probleme zu modellieren und Computerlösungen zu formulieren.

Ausgehend von anschaulichen, einfach zu erstellenden, funktionalen Abhängigkeiten bis hin zu einem konkret ausformulierten Mini-Interpreter für eine Pascal-ähnliche Sprache wird hier ein Einstieg in dieses faszinierende Programmierparadigma vorgelegt.

Die dabei verwendete Programmiersprache CAML stellt die Annehmlichkeiten einer Windows-basierten Entwicklungsumgebung bereit und ist kostenlos aus dem Internet zu beziehen.

Nicht alle Aspekte des hier vorgestellten Materials sind (unterrichtlich) leicht zu vermitteln. Der hier beschriebene Weg reizt zur Nachahmung, ist aber noch zu wenig erprobt, um alle Schwierigkeiten, die in der Praxis auftauchen können, vorherzusehen und erschöpfend darstellen zu können. Trotzdem möchten wir Sie mit dieser Veröffentlichung ermuntern, eigene Versuche in der funktionalen Programmierung zu unternehmen. Über die Rückmeldung Ihrer Erfahrungen würden wir uns sehr freuen.

Arbeitsgruppe Informatik
des Landesmedienzentrums Rheinland-Pfalz

Funktionale Programmierung

Klaus-Peter Becker

unter Mitarbeit der Arbeitsgruppe Informatik
des Landesmedienzentrums Rheinland-Pfalz

Mitglieder der Arbeitsgruppe

Dr. Klaus-Peter Becker

Dr. Herbert Drumm

Josef Glöckler

Gregor Noll

Mario Spengler

Hermann Stimm

Inhaltsverzeichnis

Jede Aufgabe, die ich löste, wurde zu einer Regel,
die später zur Lösung anderer Aufgaben diente.

René Descartes

Seite

Inhaltsverzeichnis 6

1 Einführung 7

2 Funktionale Programmierung 9
2.1 Funktionale Programmierung – was ist das? 9
2.2 Worin unterscheiden sich funktionale und imperative Programmierung? 12
2.3 Was macht die funktionale Programmierung so interessant und wichtig? 14
2.4 Funktionale Programmierung – eine Programmierwelt für die Schule? 16

3 Mit Funktionen kann man Ein- und Ausgabesituationen modellieren 19
3.1 Von Berechnungsformeln zum funktionalen Programm 21
3.2 Das Funktionskonzept der funktionalen Programmierung 37
3.3 Funktionale Modellierung komplexer Ein- und Ausgabesituationen 41

4 Mit Funktionen kann man Datentypen und Algorithmen beschreiben 55
4.1 Modellierung des Datentyps „Warteschlange“ 56
4.2 Das Listenkonzept (von CAML) 58
4.3 Das Reduktionskonzept 61
4.4 Das Verfahren der rekursiven Problemreduktion 66
4.5 Ein komplexeres Programmierproblem 73

5 Mit Funktionen kann man komplexe Softwareprodukte entwerfen 81
5.1 Das Softwareprodukt 82
5.2 Das Speicherkonzept – Variablenzustände 85
5.3 Ein Interpreter für primitive Programme 89
5.4 Terme 92
5.5 Ein Interpreter für Zuweisungssequenzen 96
5.6 Anweisungen – Kontrollstrukturen 98
5.7 Prototyping 105
5.8 Funktionale und imperative Programmierung 110

6 Literatur 115

Hinweise zur Programmiersprache CAML – Bezugsquelle 116

1 Einführung

Der Lehrplan Informatik nennt als ein fundamentales Ziel des Informatikunterrichts, Fähigkeiten zur Problemlösung durch Anwendung von Methoden, Werkzeugen und Standardverfahren der Informatik zu entwickeln (vgl. [Lehrplan 93]). Hierbei spielen algorithmisch orientierte Verfahren der Problemlösung eine zentrale Rolle. Bei der Behandlung dieser algorithmisch orientierten Verfahren im Unterricht wird in der gängigen Unterrichtspraxis fast ausschließlich das imperative Algorithmuskonzept benutzt. Nichtimperative Konzepte finden dagegen kaum Beachtung. Dies ist eine einseitige Weichenstellung, die sich negativ auswirken kann:

Die Sprache, in der eine Problemlösung formuliert werden soll, hat einen prägenden Einfluss auf die Denkweisen beim Lösen des Problems selbst. Wenn man Problemlösungen ausschließlich imperativ formuliert, bilden sich einseitige Denkschemata aus. Da eine imperative Beschreibung von der Vorstellung eines anweisungsgesteuerten (konkreten oder abstrakten) Prozessors geprägt ist, wird das Denken so maschinenorientiert ausgerichtet. Zwar können alle algorithmisch lösbaren Probleme mit Hilfe imperativer Denkschemata gelöst werden, viele dieser Probleme lassen sich jedoch mit anderen, nichtimperativen Denkschemata adäquater lösen. Um einer einseitigen Prägung des Denkens entgegenzuwirken, sollten Schülerinnen und Schüler frühzeitig ein erweitertes Spektrum von Denkschemata kennen lernen.

Zu den nichtimperativen Denkschemata, die beim Lösen von Problemen eingesetzt werden, zählt zum einen das Logik-basierte Denken, zum anderen das funktionale Denken. Wie man mit Logik-basierten Mitteln erfolgreich Probleme bearbeiten kann, wird in der Handreichung zum Lehrplan „Wissensverarbeitung mit PROLOG“ (vgl. [Drumm & Stimm 95]) aufgezeigt. Die vorliegenden Materialien zum Lehrplan können benutzt werden, um einen Einblick in das Problemlösen mit funktionalen Mitteln zu geben.

Ziel der folgenden Ausführungen ist, einen schülergemäßen Zugang zur funktionalen Programmierung aufzuzeigen. Unter anderem wird die Beschreibung von Datentypen und Algorithmen mit Funktionen entwickelt. Es ist nicht beabsichtigt, funktionale Programmierung systematisch und vollständig darzustellen (vgl. hierzu z. B. [Bird & Walder 92], [Thiemann 94], [Wolff von Gudenberg 96]). Vielmehr werden einige Aspekte funktionaler Programmierung herausgegriffen und an ausgewählten Problemkontexten exemplarisch aufgezeigt. Die Vorgehensweise orientiert sich dabei an den grundlegenden Konzepten der funktionalen Programmierung. Diese werden allgemein (d. h. Programmiersprachen-unabhängig) dargestellt. Zusätzlich wird eine Implementierung in einer speziellen Programmiersprache aufgezeigt. Hierzu wird die funktionale Sprache CAML (Akronym für Categorical Abstract Machine Language) benutzt, die über das Internet kostenlos bezogen werden kann (siehe Seite 116). Es ist aber auch möglich, eine andere Sprache wie z. B. LOGO oder LISP zu verwenden. Die dazu notwendigen Änderungen können ohne Schwierigkeiten vorgenommen werden.

Eine Behandlung der funktionalen Programmierung ist nach dem Lehrplan Informatik in mehreren Phasen möglich. In der Einstiegsphase (Klassenstufe 11) kann eine funktio-

nale Programmiersprache – analog zur logischen Programmierung mit PROLOG (vgl. [Drumm & Stimm 95]) – als Modellierungswerkzeug benutzt werden. Für die Programmierphase (Klassenstufe 11) wird im Lehrplan zwar eine imperative Programmiersprache vorgesehen, andere Zugänge zur Programmierung – also auch der Zugang über funktionale Programmierung – werden aber zugelassen. Schließlich kann das Abschlussprojekt (Klassenstufe 13) so gewählt werden, dass hier Aspekte der funktionalen Programmierung integriert werden können. Die vorliegenden Materialien können (in Teilen) in all diesen Phasen verwendet werden.

Für die Bearbeitung der Materialien werden keine speziellen, über den normalen Informatikunterricht hinausgehenden Kenntnisse benötigt. Insbesondere werden keine Vorkenntnisse über funktionale Programmierung vorausgesetzt. Alle benötigten Sachverhalte werden in den Materialien bereitgestellt. Für weiterführende Fragestellungen finden sich entsprechende Literaturhinweise.

2 Funktionale Programmierung

Im Folgenden werden zur Orientierung die Grundzüge des funktionalen Programmieransatzes kurz dargestellt und mit dem imperativen Programmieransatz verglichen. Anschließend wird diskutiert, inwiefern funktionale Programmierung den Informatikunterricht des Gymnasiums bereichern kann.

2.1 Funktionale Programmierung – was ist das?

Funktionale Programmierung zeichnet sich dadurch aus, dass problemrelevante Abhängigkeiten zwischen einer gegebenen Ausgangssituation und einer gewünschten Endsituation mit Hilfe von Funktionen erfasst werden. Dieser Ansatz soll anhand eines Beispiels hier kurz erläutert werden.

Problem:

Eine Folge von Objekten – hier der Einfachheit halber Zahlen – soll sortiert werden; z. B.:
Ausgangssituation: `[6; 5; 3; 8]`
Endsituation: `[3; 5; 6; 8]`

Das Sortierproblem hat eine Vielzahl von Lösungen. Die Grundidee des hier zu betrachtenden Sortierverfahrens (Sortieren durch Einfügen) besteht darin, wie ein Skatspieler vorzugehen, der die Karten einzeln aufnimmt und jede in das bereits in der Hand befindliche Blatt an der richtigen Stelle einfügt. Geht man systematisch vor, so kann man jeweils das letzte bzw. erste Objekt der Ausgangsfolge an die richtige Stelle in der Zielfolge einfügen.

```
[6; 5; 3; 8]                                   [6; 5; 3; 8]
[6; 5; 3]    [8]                         [6]    [5; 3; 8]
[6; 5]    [3; 8]        bzw.             [5; 6]    [3; 8]
[6]    [3; 5; 8]                         [3; 5; 6]    [8]
    [3; 5; 6; 8]                         [3; 5; 6; 8]
```

Das folgende funktionale CAML-Programm soll diese Grundidee realisieren.

```
let rec Einfuegen = function
    (x, [])                  -> [x] |
    (x, erstes :: restListe) -> if x < erstes
                                then x :: (erstes :: restListe)
                                else erstes :: Einfuegen(x, restListe);;

let rec Sortieren = function
    []                  -> [] |
    erstes :: restListe -> Einfuegen(erstes, Sortieren(restListe));;

Sortieren([6; 5; 3; 8]);;
```

Das funktionale Programm besteht aus Funktionsdefinitionen und einem Funktionsaufruf. Die Funktionsdefinitionen legen die Operationen `Einfuegen` und `Sortieren` fest, der Funktionsaufruf `Sortieren([6; 5; 3; 8])` initiiert die Berechnung.

In der Definition der Funktion `Einfuegen` wird zunächst der Fall „Einfügen einer Zahl x in eine leere Liste [ ]" behandelt. Der Funktionswert kann hier unmittelbar angegeben werden. Im Fall „Einfügen einer Zahl x in eine nichtleere bereits sortierte Liste" wird die nichtleere Liste in der Form `erstes :: restListe` vorgegeben. Dies soll darlegen, dass die Liste aus einem ersten Element und einer Restliste aufgebaut ist. Ist die einzufügende Zahl x kleiner als das erste Element `erstes` der Liste, so ergibt sich der Funktionswert, indem man x in die Liste vor dieses erste Element setzt. Das Symbol `::` deutet das Aufnehmen als erstes Element in einer Liste an. Ist die einzufügende Zahl x hingegen nicht kleiner als das erste Element `erstes` der Liste, so ergibt sich der Funktionswert, indem man x in die Restliste `restListe` einfügt und dieser neuen Liste das erste Element `erstes` voranstellt.

In der Definition der Funktion `Sortieren` wird zunächst der triviale Fall „Sortieren einer leeren Liste" behandelt. Im Fall „Sortieren einer nichtleeren Liste" wird diese in der Form `erstes :: restListe` vorgegeben. Den Funktionswert erhält man, indem man das erste Element mit Hilfe der Funktion `Einfuegen` in die mit der Funktion `Sortieren` sortierte Restliste einfügt.

Bei der Auswertung eines Funktionsaufrufs werden die Funktionsdefinitionen nach und nach angewandt. Im vorliegenden Fall ergibt sich die folgende Berechnung:

```
   Sortieren([6;5;3;8])
-> Einfuegen(6, Sortieren([5;3;8]))
-> Einfuegen(6, Einfuegen(5, Sortieren([3;8])))
-> Einfuegen(6, Einfuegen(5, Einfuegen(3, Sortieren([8]))))
-> Einfuegen(6, Einfuegen(5, Einfuegen(3, Einfuegen(8, Sortieren([])))))
-> Einfuegen(6, Einfuegen(5, Einfuegen(3, Einfuegen(8, []))))
-> Einfuegen(6, Einfuegen(5, Einfuegen(3, [8])))
-> Einfuegen(6, Einfuegen(5, 3 :: (8 :: [])))
-> Einfuegen(6, Einfuegen(5, 3 :: [8]))
-> Einfuegen(6, Einfuegen(5, [3;8]))
-> Einfuegen(6, 3 :: Einfuegen(5, [8]))
-> Einfuegen(6, 3 :: (5 :: (8 :: [])))
-> Einfuegen(6, 3 :: (5 :: [8]))
-> Einfuegen(6, 3 :: [5; 8])
-> Einfuegen(6, [3; 5; 8])
-> 3 :: Einfuegen(6, [5; 8])
-> 3 :: (5 :: Einfuegen(6, [8]))
-> 3 :: (5 :: (6 :: (8 :: [])))
-> 3 :: (5 :: (6 :: [8]))
-> 3 :: (5 :: [6;8])
-> 3 :: [5;6;8]
-> [3;5;6;8]
```

Ein Berechnungsschritt (d. h.: eine Anwendung einer Funktionsdefinition) wird hier durch einen Pfeil angedeutet. Ein Berechnungsschritt der Gestalt `3 :: (5 :: [6;8]) -> 3 :: [5;6;8]` nutzt dabei die Definition der vordefinierten Funktion `::` aus.

Das Beispiel zeigt, dass man mit Funktionen programmieren kann: Mit Hilfe von Funktionen lassen sich (komplexe) Berechnungsverfahren beschreiben. Bei der Konstruktion der Berechnungsvorschrift nutzt man dabei die „Konstruktionsprinzipien" Komposition, Fallunterscheidung und Rekursion aus. Eine konkrete Berechnung initiiert man durch einen Funktionsaufruf. Der Funktionswert lässt sich ermitteln, indem man wiederholt Funktionsdefinitionen anwendet. Diese Aufgabe kann von einem geeigneten System übernommen werden. Funktionale Programmierung besteht demnach im Modellieren und Erstellen von Funktionsdefinitionen und dem anschließenden Erzeugen von relevanten Funktionsaufrufen.

2.2 Worin unterscheiden sich funktionale und imperative Programmierung?

Wir versuchen, den fundamentalen Unterschied anhand eines sehr einfachen Problems klarzumachen.

Problem:

Zwei Objekte sollen ausgetauscht werden.

Funktionale Lösung:

```
let Vertauschen = function
    (x, y) -> (y, x);;
```

Mit Hilfe einer Funktionsdefinition wird hier direkt das gewünschte Ein- / Ausgabeverhalten erfasst. Anders eine typisch imperative Lösung:

Imperative Lösung:

```
BEGIN
z := x;
x := y;
y := z
END.
```

Die Festlegung des Berechnungsverfahrens erfolgt hier mit Hilfe von Anweisungen. Diese dienen im Wesentlichen dazu, die Werte von Variablen zu ändern. Die folgende Auflistung zeigt, wie nach und nach der Zustand der im Programm benutzten Variablen mit Hilfe der im Programm vorkommenden Wertzuweisungen in der gewünschten Weise verändert wird.

Anweisung	Variablenzustand
	{x:[4]; y:[1]; z:[ ]}
z := x	{x:[4]; y:[1]; z:[4]}
x := y	{x:[1]; y:[1]; z:[4]}
y := z	{x:[1]; y:[4]; z:[4]}

Variablen repräsentieren Speicherplätze. Wir deuten dies durch die Schreibweise `x:[3]` an. Der Speicherplatz, den die Variable x bezeichnet, ist hier mit dem momentanen Wert 3 belegt.

Ein Variablenzustand beschreibt somit den momentanen (interessierenden) Speicherzustand. Berechnungen werden durch Befehle bzw. Anweisungen initiiert. Besondere Bedeutung haben die Wertzuweisungen. Diese Anweisungen verändern i. a. den Wert einer Variablen und damit die Speicherbelegung. Die Reihenfolge der abzuarbeitenden Wertzuweisungen wird durch Kontrollanweisungen festgelegt.

Programmieren mit Anweisungen kann demnach als maschinenorientierte Programmierung bezeichnet werden. Mit Hilfe von Anweisungen wird festgelegt, wie eine (hypothetische) Maschine Schritt für Schritt die gewünschte Operation ausführt. Anders das Programmieren mit Funktionen: Hier wird nicht festgelegt, wie eine Operation ausgeführt werden soll, sondern beschrieben, was die Operation leistet und zwar durch Funktionsdefinitionen.

Beim Festlegen von Berechnungsverfahren kann man sich also in sehr verschiedenen Welten bewegen: der Welt der Anweisungen (imperative Algorithmen) bzw. der Welt der funktionalen Beschreibungen (funktionale Algorithmen / funktionale Spezifikationen) – für weitere Details siehe Kapitel 5.8 ab Seite 110.

2.3 Was macht die funktionale Programmierung so interessant und wichtig?

Funktionale Programmierung erfolgt problemorientiert.

Wie oben gezeigt, wird beim funktionalen Programmieren festgelegt, was eine Operation leistet und nicht, wie sie ausgeführt werden kann. Funktionale Programmierung ist demnach eine Form der deklarativen Programmierung (im Gegensatz zur imperativen Programmierung). Diese Form der Programmierung ist umso günstiger, je komplexer das Problem bzw. dessen Löseverfahren ist. Die funktionale Beschreibung beschränkt sich auf das Erfassen der konzeptionellen Ideen. Die imperative Beschreibung muss dagegen mehr Festlegungen im Detail treffen. Für einen in beiden Programmierwelten erfahrenen Programmierer ist es infolgedessen oft sehr viel einfacher, ein komplexes Problem mit deklarativen Methoden zu lösen.

Funktionale Programme sind klar strukturiert.
Funktionale Programmierung kennt keine Seiteneffekte.

Strukturierung erlangt man meist durch Abstraktionsmechanismen. In der Entwicklung von Programmierkonzepten wurden eine Reihe solcher Abstraktionsmechanismen erfolgreich entwickelt. Beispielsweise führte das Ersetzen des GOTO – Sprungbefehls durch Kontrollbefehle zu mehr Klarheit in der Darstellung der algorithmischen Verfahren.

Eine Quelle der Unklarheit bilden in der imperativen Programmierung nach wie vor die Wertzuweisungen. Mit ihrer Hilfe können wenig transparente Seiteneffekte produziert werden. Beispielsweise erzeugen die beiden Funktionsaufrufe im folgenden imperativen Programm verschiedene Ausgaben:

```
PROGRAM Demo;

VAR z : integer;

FUNCTION f (x: integer) : integer;
  BEGIN
  f := x+z;
  z := z+1
  END;

BEGIN
z := 0;
WriteLn(f(0));
WriteLn(f(0))
END.
```

Funktionale Programmierung verzichtet auf dieses Mittel „Wertzuweisung“. Man beschränkt sich darauf, alle Operationen mit Hilfe des abstrakteren Mittels „Funktion“ zu beschreiben. Funktionen haben klare Schnittstellen, ihr Verhalten kann somit leicht eingesehen werden. Seiteneffekte – wie im obigen Programm – sind nicht möglich, da es

keine globalen Variablen gibt. Die Folge: Funktionale Programme sind meist leicht zu verstehen. Fehler können schnell erkannt und korrigiert werden.

Die Grundprinzipien der funktionalen Programmierung sind sehr einfach.

Zum Verständnis der Grundideen benötigt man nur den Funktionsbegriff. Auf diesem Begriff lässt sich ein ganzes Programmierparadigma aufbauen. Wie bereits gezeigt, kann man mit Funktionen programmieren. Eine genauere Analyse zeigt, dass Funktionen hierbei als ausführbare Spezifikationen betrachten werden: Funktionen dienen dazu, Ein- und Ausgabesituationen zu spezifizieren. Zum einen erfolgt dies modellierend, indem eine Ein- bzw. Ausgabesituation beschrieben wird. Hierbei steht der Zuordnungsaspekt im Vordergrund, Eingabedaten werden Ausgabedaten zugeordnet. Zum anderen erfolgt dies definierend: Mit Hilfe einer mathematischen Definition wird das gewünschte Ein- oder Ausgabeverhalten präzise festgelegt (*deklarativer Aspekt*). Die Festlegung muss ausführbar sein in dem Sinne, dass eine Anwendung der Funktionsdefinition als Reduktionsschema möglich ist (*operationaler Aspekt*). Den Aspekt „ausführbare Spezifikation“ macht man sich beim sog. *Rapid Prototyping* zu Nutze. Hier geht es darum, schnell einen Prototyp eines Softwareprodukts zu erstellen, das in seiner Funktionalität, aber evtl. noch nicht in allen Details mit dem gewünschten Endprodukt übereinstimmt. Es zeigt sich insbesondere in der KI, dass funktionale Programmierung hier erfolgreich eingesetzt werden kann.

Weitere Vorzüge der funktionalen Programmierung werden in der Fachliteratur (z. B. [Bird & Walder 92], [Thiemann 94], [Wolff von Gudenberg 96]) beschrieben.

2.4 Funktionale Programmierung – eine Programmierwelt für die Schule?

Weshalb ist die Behandlung der funktionalen Programmierung im Informatikunterricht sinnvoll?

Funktionale Programmierung fördert einen klaren Programmierstil und damit möglicherweise auch einen klaren Denkstil.

Funktionale Programmiersprachen erzwingen einen strukturierenden und abstrahierenden Programmierstil. Als Folge ergeben sich meist kurze, verständliche Programme und Problemlösungen. Fehler lassen sich leicht vermeiden bzw. lokalisieren.

Funktionale Programmierung fördert funktionales Denken.

Die Bedeutung des funktionalen Denkens wird seit der Meraner Reform zu Beginn dieses Jahrhunderts herausgestellt. Der Funktionsbegriff zählt seither zu den Leitgedanken des Mathematikunterrichts. Funktionales Programmieren basiert auf funktionalem Denken und ist somit geeignet, dieses zu fördern.

Funktionale Programmierung fördert einen verständigen Umgang mit Softwaretools.

In letzter Zeit gewinnen Werkzeuge im Mathematikunterricht immer mehr an Bedeutung. Die Arbeit mit Softwaretools zeigt dabei vielfach Züge funktionaler Programmierung auf. Benutzt man etwa das Computer-Algebra-System DERIVE beim Lösen eines mathematischen Problems, so wird man i. a. eine Reihe von Funktionen definieren und diese dann auf interessierende Argumente anwenden. Funktionale Programmierung kann also dazu beitragen, zu verstehen, wie man ein Werkzeug wie DERIVE Gewinn bringend nutzen kann.

Funktionale Programmierung zeigt eine weitere Dimension des Funktionsbegriffs auf.

Durch funktionales Programmieren lässt sich ein weiterer Einsatz des Funktionsbegriffs aufzeigen: Mit Funktionen kann man nicht nur Zusammenhänge erfassen, mit Funktionen kann man auch programmieren.

Funktionale Programmierung zeigt eine weitere Form der Programmierung auf.

Durch funktionales Programmieren gewinnt man einen tieferen Einblick in das algorithmischen Problemlösen: Problemlösungen können nicht nur mit Anweisungen formuliert werden, sondern auch – abstrakter – mit Hilfe von Funktionen.

Die genannten Aspekte machen deutlich, dass funktionale Programmierung eine Reihe von Vorzügen besitzt, die den Informatikunterricht und auch den Mathematikunterricht bereichern können. Aber kann funktionale Programmierung auch wirklich in der Schule erlernt werden? Bisher stellt die fachdidaktische Literatur nur wenig konkrete Unterrichtshilfen bereit. In [Schwill 93], [Fischbacher 97] und [Puhlmann 98] wird das Erstellen konkreter Materialien gefordert. Solche Materialien findet man in [ISB 97] und [Wagenknecht 94]. Dabei handelt es sich um einen systematischen, schulgemäßen Lehrgang zur funktionalen Programmierung. Die im Folgenden vorgestellten Materialien zeigen dagegen einen anderen Weg auf. Anhand ausgewählter Problemkontexte werden unterschiedliche Facetten der funktionalen Programmierung didaktisch aufbereitet. Die Grundidee „Mit Funktionen kann man programmieren“ wird dabei unter drei verschiedenen Aspekten mit wachsendem Komplexitätheitsgrad beleuchtet: Zunächst wird aufgezeigt, dass man mit Funktionen Ein- und Ausgabesituationen modellieren kann. Hier werden einfache Berechnungsformeln funktional interpretiert. In einem zweiten Schritt wird aufgezeigt, dass man mit Funktionen Algorithmen beschreiben und realisieren kann. Hier stehen typische Konzepte der funktionalen Programmierung wie Rekursion und Listenverarbeitung im Mittelpunkt. Schließlich wird aufgezeigt, dass man mit Funktionen komplexe Softwareprodukte entwerfen kann. Hierzu wird an einem Beispiel ein Einblick in das so genannte *Rapid Prototyping* vermittelt.

Raum für Notizen

3 Mit Funktionen kann man Ein- und Ausgabesituationen modellieren

Inhalte:

- Ein- und Ausgabesituationen werden mit Hilfe von Funktionen beschrieben.
- Funktionale Programmierung besteht darin, Funktionsdefinitionen und Funktionsaufrufe zu konzipieren.
- In der funktionalen Programmierung werden Funktionen als partielle Funktionen behandelt.
- Die Beschreibung von Eingabe- und Ausgabebereichen von Funktionen erfolgt mit Hilfe von Typen (Signaturen).
- Funktionen können beliebig geschachtelt werden (Kompositionsprinzip). Zur Modellierung von Fallunterscheidungen wird ein if-then-else-Konstrukt verwendet.

Bemerkungen:

Das Einstiegsbeispiel „Berechnung von Dachflächen" soll in die funktionale Programmierung einführen. Wir schlagen hier ein schrittweises Heranführen an die Grundideen der funktionalen Programmierung vor (siehe Abschnitt 3.1 ab Seite 21):

1. Schritt: Entwicklung von Berechnungsformeln
Mit Hilfe von Berechnungsformeln werden die problemrelevanten Berechnungen erfasst. Das Einstiegsproblem ist nicht trivial, aber doch so einfach, dass es von Schülerinnen und Schülern selbständig mit Mittelstufenkenntnissen gelöst werden kann.

2. Schritt: Funktionale Deutung der Formeln
Hier wird ein Sichtwechsel vorgenommen. Statt statischer Formeln werden jetzt dynamische Zuordnungen benutzt, um die beabsichtigten Berechnungen zu beschreiben.

3. Schritt: Deutung der Funktionsdefinitionen als funktionales Programm
Die Zuordnungssituationen werden informatisch als Ein- bzw. Ausgabesituationen interpretiert. Mit ihrer Hilfe werden aus Eingabewerten Ausgabewerte berechnet. Die konkrete Berechnung leistet ein Programmiersystem.

Anhand des Einstiegsbeispiels können bereits einige wesentliche Aspekte der funktionalen Programmierung eingeführt werden. Eine Zusammenstellung solcher Aspekte findet man in Abschnitt 3.2. ab Seite 37. In Abschnitt 3.3 ab Seite 42 wird ein geometrisches Problem zur Vertiefung dieser Aspekte skizziert.

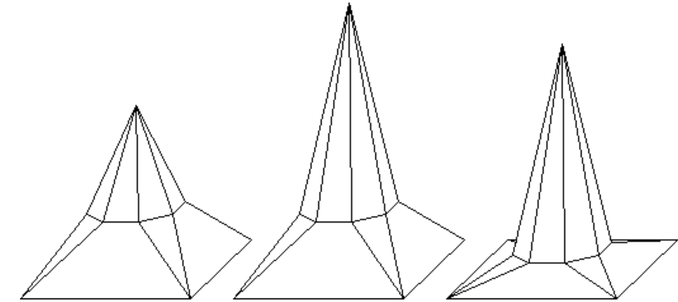
Die beiden Problemkontexte im vorliegenden Kapitel werden ohne die für die funktionale Programmmierung charakteristischen Konzepte „Liste“ und „Rekursion“ bearbeitet. Hierdurch wird erreicht, dass die Komplexität der funktionalen Programme zu Beginn recht niedrig bleibt. Das Hauptgewicht liegt in der funktionalen Modellierung von Ein- und Ausgabesituationen. Die Wahl geometrischer Kontexte hilft, die zu berechnenden Größen stets veranschaulichen und konkretisieren zu können.

3.1 Von Berechnungsformeln zum funktionalen Programm

Problem:

Berechnung von Dachflächen

Viele romanische Kirchen haben auf ihrem Turm ein Dach, das die folgende Gestalt hat:



Das Dach besteht aus einer achteckigen Pyramide und einem Übergangsstück, welches den Übergang vom quadratischen Turm zur achteckigen Grundfläche der Turmspitze bildet. Eine Dachdeckerfirma, die sich auf die Renovierung von Kirchturmdächern spezialisiert hat, benötigt für die Erstellung von Kostenvoranschlägen ein Programm, mit dessen Hilfe man für die gezeigte Dachform den Flächeninhalt der gesamten Dachfläche berechnen kann.

Ziel ist es im Folgenden, ein solches Programm zu entwickeln. Dabei soll ein Programmierkonzept benutzt werden, das ausschließlich auf dem Funktionsbegriff basiert.

1. Lösungsschritt:

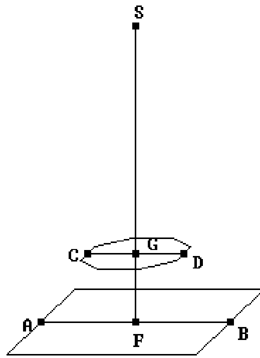
Entwicklung von Berechnungsformeln

Zunächst muss erkannt werden, wie die zu betrachtende Fläche strukturiert ist und von welchen Größen ihr Flächeninhalt abhängt.

Die Dachfläche setzt sich wie folgt zusammen: Das Pyramidenstück besteht aus acht Dreiecken, das Übergangsstück aus vier Dreiecken und vier Trapezen. Die Gesamtoberfläche A_{Dach} des Dachs hängt offensichtlich ab von

- der Breite des Grundquadrats $b_{Quadrat}$ (Länge der Strecke AB in der Skizze),
- der Breite des Achtecks $b_{Achteck}$ (Länge der Strecke CD in der Skizze),

- der Höhe des Übergangsstücks $h_{Übergang}$ (Länge der Strecke FG in der Skizze),
- der Höhe der Pyramide $h_{Pyramide}$ (Länge der Strecke GS in der Skizze).

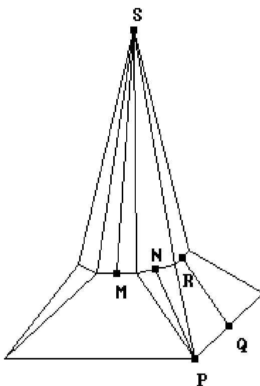


Zur Bestimmung des Flächeninhalts A_{Dach} müssen die folgenden Teilflächenstücke betrachtet werden:

- acht Pyramidendreiecke (Flächeninhalt: $A_{PDreieck}$),
- vier Übergangsdreiecke (Flächeninhalt: $A_{ÜDreieck}$),
- vier Übergangstrapeze (Flächeninhalt: $A_{ÜTrapez}$).

Die folgenden Hilfsgrößen werden zur Berechnung dieser Flächeninhalte eingeführt:

- Höhe des Übergangstrapezes $h_{ÜTrapez}$ (Länge der Strecke RQ in der Skizze),
- Höhe des Übergangsdreiecks $h_{ÜDreieck}$ (Länge der Strecke PN in der Skizze),
- Höhe des Pyramidendreiecks $h_{PDreieck}$ (Länge der Strecke MS in der Skizze),
- Seitenlänge des Achtecks $s_{Achteck}$.



Als Nächstes werden die benötigten Berechnungsformeln entwickelt. Zur Illustration der Verwendung der Berechnungsformeln wird eine Berechnung anhand konkreter Zahlenwerte durchgeführt. Wir gehen von den folgenden vorgegebenen Werten aus:

- $b_{Quadrat} = 4.0$
- $b_{Achteck} = 2.0$
- $h_{Übergang} = 3.0$
- $h_{Pyramide} = 10.0$

Mit Hilfe elementargeometrischer Überlegungen erhält man:

- $h_{ÜTrapez} = \sqrt{\left(\frac{b_{Quadrat} - b_{Achteck}}{2}\right)^2 + h_{Übergang}^2} = 3.16..$
- $h_{ÜDreieck} = \sqrt{\left(\frac{\sqrt{2} \cdot b_{Quadrat} - b_{Achteck}}{2}\right)^2 + h_{Überang}^2} = 3.51..$
- $h_{PDreieck} = \sqrt{\left(\frac{b_{Achteck}}{2}\right)^2 + h_{Pyraide}^2} = 10.04..$
- $s_{Achteck} = b_{Achteck} \cdot \tan(\pi / 8) = 0.82..$
- $A_{ÜTrapez} = \frac{1}{2} \cdot (b_{Quadrat} + s_{Achteck}) \cdot h_{ÜTrapez} = 7.63..$
- $A_{ÜDreieck} = \frac{1}{2} s_{Achteck} \cdot h_{ÜDreieck} = 1.45..$
- $A_{PDreieck} = \frac{1}{2} s_{Achteck} \cdot h_{PDreieck} = 4.16..$
- $A_{Dach} = 8 \cdot A_{PDreieck} + 4 \cdot (A_{ÜDreieck} + A_{ÜTrapez}) = 69.66..$

Mit Hilfe dieser Berechnungsformeln lässt sich jetzt für eine beliebige Dachform, die die vorgegebene Struktur hat, die Gesamtoberfläche des Daches aus den gegebenen Größen berechnen.

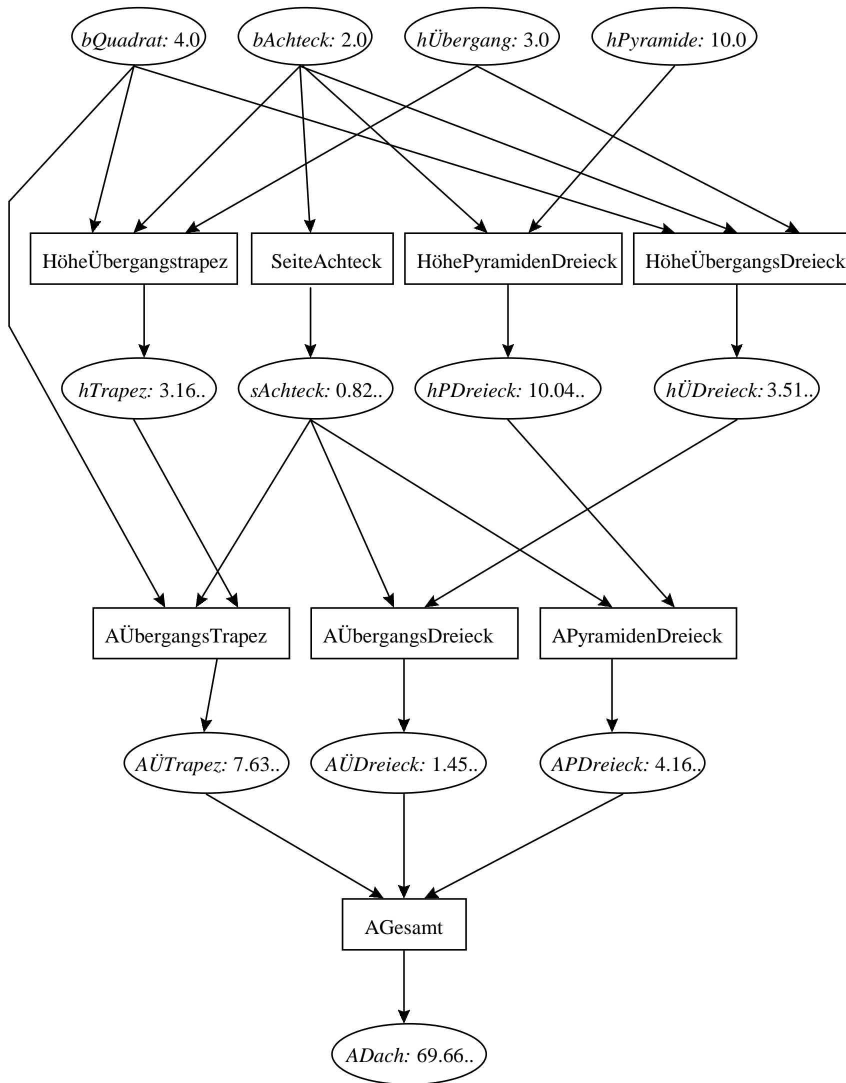
2. Lösungsschritt:

Funktionale Deutung der Formeln

Wir deuten die Berechnungsformeln jetzt als Zuordnungssituationen. Zu diesem Zweck analysieren wir zunächst die Abhängigkeit zwischen den beteiligten Größen. Diese sind in der folgenden Übersicht zusammengestellt.

Übersicht:

funktionale Modellierung der Abhängigkeiten zwischen den Bestimmungsstücken

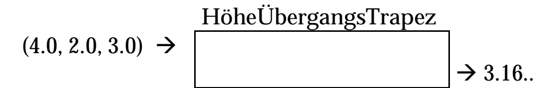


Erläuterung:

Die Größe h_{Trapez} ergibt sich aus den Größen $b_{Quadrat}$, $b_{Achteck}$ und $h_{Übergang}$. Die entsprechende Zuordnung wird durch eine Funktion mit der Bezeichnung „Höhe-Übergangstrapez“ beschrieben. Diese Funktion wird in der Übersicht durch eine „black-box“ repräsentiert.

Die benötigten Funktionen werden jetzt modelliert und definiert. Die Modellierung dient dazu, das Verhalten der intendierten Zuordnung möglichst genau zu beschreiben.

Modellierung:



Verhaltensbeschreibung: Die Funktion „HöheÜbergangsTrapez“ ordnet der Breite des Grundquadrats, der Breite des Achtecks und der Höhe des Übergangsstücks die Höhe des Übergangstrapezes zu.

Definition:

HöheÜbergangsTrapez(bQuadrat, bAchteck, hÜbergang) =

$$\sqrt{\left(\frac{\text{bQuadrat} - \text{bAchteck}}{2}\right)^2 + \text{hÜbergang}^2}$$

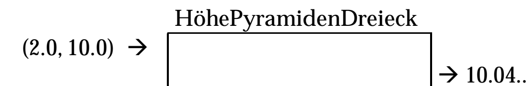
Modellierung:

Verhaltensbeschreibung: Die Funktion „SeiteAchteck“ ordnet der Breite des Achtecks die Länge der Seite des Achteck zu.

Definition:

SeiteAchteck(bAchteck) = bAchteck · TAN(π / 8)

Modellierung:



Verhaltensbeschreibung: Die Funktion „HöhePyramidenDreieck“ ordnet der Breite des Achtecks und der Höhe der Pyramide die Höhe des Pyramidendreiecks zu.

Definition:

$$\text{HöhePyramidenDreieck(bAchteck, hPyramide)} = \sqrt{\left(\frac{\text{bAchteck}}{2}\right)^2 + \text{hPyramide}^2}$$

Modellierung:

(4.0, 2.0, 3.0) → [HöheÜbergangsDreieck] → 3.51..

Verhaltensbeschreibung: Die Funktion „HöheÜbergangsDreieck“ ordnet der Breite des Grundquadrats, der Breite des Achtecks und der Höhe des Übergangsstücks die Höhe des Übergangsdreiecks zu.

Definition:

HöheÜbergangsDreieck(bQuadrat, bAchteck, hÜbergang) =

$$\sqrt{\left(\frac{\sqrt{2}\cdot\text{bQuadrat} - \text{bAchteck}}{2}\right)^2 + \text{hÜbergang}^2}$$

Modellierung:

(4.0, 0.82.., 3.16..) → [AÜbergangsTrapez] → 7.63..

Verhaltensbeschreibung: Die Funktion „AÜbergangsTrapez“ ordnet der Grundseite, Gegenseite und Höhe des Übergangstrapezes den Flächeninhalt des Übergangstrapezes zu.

Definition:

AÜbergangsTrapez(bQuadrat, sAchteck, hÜTrapez) =

$$\frac{1}{2}\cdot(\text{bQuadrat} + \text{sAchteck})\cdot\text{hÜTrapez}$$

Modellierung:

(0.82.., 3.51..) → [AÜbergangsDreieck] → 1.45..

Verhaltensbeschreibung: Die Funktion „AÜbergangsDreieck“ ordnet der Grundseite und der Höhe des Übergangsdreiecks den Flächeninhalt des Übergangsdreiecks zu.

Definition:

$$\text{AÜbergangsDreieck(sAchteck, hÜDreieck)} = \frac{1}{2}\cdot\text{sAchteck}\cdot\text{hÜDreieck}$$

Modellierung:

(0.82.., 10.04..) → [APyramidenDreieck] → 4.16..

Verhaltensbeschreibung: Die Funktion „APyramidenDreieck“ ordnet der Grundseite und der Höhe des Pyramidendreiecks den Flächeninhalt des Pyramidendreiecks zu.

Definition:

$$\text{APyramidenDreieck(sAchteck, hPDreieck)} = \frac{1}{2}\cdot\text{sAchteck}\cdot\text{hPDreieck}$$

Modellierung:

(7.63.., 1.45.., 4.16..) → [AGesamt] → 69.66..

Verhaltensbeschreibung: Die Funktion „AGesamt“ ordnet den Flächeninhalten des Übergangstrapezes, des Übergangsdreiecks und des Pyramidendreiecks den Gesamtflächeninhalt des Dachs zu.

Definition:

AGesamt(aÜTrapez, aÜDreieck, aPDreieck) =

$$4\cdot(\text{aTrapez} + \text{aÜDreieck}) + 8\cdot\text{aPDreieck}$$

Durch das Einführen von Funktionen ergibt sich ein neuer Beschreibungsformalismus: Die einzelnen Daten (wie z. B. der Wert 3.16..) werden nicht mehr durch Bezeichner (in der Übersicht oben: *hÜTrapez*) beschrieben, sondern durch funktionale Ausdrücke (hier: HöheÜbergangsTrapez(4.0, 2.0, 3.0)). Die folgende Übersicht macht diesen Wechsel deutlich.

Übersicht:

Bestimmungsstück	Wert	funktionale Beschreibung
$b_{Quadrat}$	4.0	
$b_{Achteck}$	2.0	
$h_{Übergang}$	3.0	
$h_{Pyramide}$	10.0	
$s_{Achteck}$	0.82..	SeiteAchteck(2.0)
$h_{ÜTrapez}$	3.16..	HöheÜbergangsTrapez(4.0, 2.0, 3.0)
$h_{ÜDreieck}$	3.51..	HöheÜbergangsDreieck(4.0, 2.0, 3.0)
$h_{PDreieck}$	10.04..	HöhePyramidenDreieck(2.0, 10.0)
$A_{ÜTrapez}$	7.63..	AÜbergangsTrapez(3.16.., 4.0, 0.82..)
$A_{ÜDreieck}$	1.45..	AÜbergangsDreieck(0.82.., 3.51..)
$A_{PDreieck}$	4.16..	APyramidenDreieck(0.82.., 10.04..)
A_{Dach}	69.66..	AGesamt(7.63.., 1.45.., 4.16..)

Die interessierenden Größen können also mit Hilfe von Funktionstermen beschrieben werden. Diese Tatsache erscheint selbstverständlich, bereitet Schülerinnen und Schülern aber oft Schwierigkeiten. Sie sollte daher thematisiert werden.

3. Lösungsschritt:

Deutung als funktionales Programm

Im zweiten Lösungsschritt wurden die Berechnungsformeln mathematisch als Zuordnungssituationen gedeutet. Im Folgenden sollen diese Zuordnungssituationen informatisch als Ein-/Ausgabe-Situationen interpretiert werden. Mit ihrer Hilfe werden aus Eingabewerten Ausgabewerte berechnet. Die konkrete Berechnung soll ein Programmiersystem leisten. Wir verwenden hier das funktionale Programmiersystem CAML (Bezugsquelle siehe Seite 116; andere Systeme wie z. B. LOGO, LISP, DERIVE, Pascal eignen sich ebenfalls). Die Funktionsdefinitionen müssen hierzu an die Syntax der gewählten Programmiersprache CAML angepasst werden.


```
let pi = 3.14159265;;

let sqr = function
  x -> x *. x;;

let HöheÜbergangsTrapez = function
  (bQuadrat, bAchteck, hÜbergang) ->
    sqrt(sqr((bQuadrat -. bAchteck) /. 2.0) +. sqr(hÜbergang));;

let SeiteAchteck = function
  bAchteck -> bAchteck *. tan(pi /. 8.0);;

let HöhePyramidenDreieck = function
  (bAchteck, hPyramide) ->
    sqrt(sqr(bAchteck /. 2.0) +. sqr(hPyramide));;

let HöheÜbergangsDreieck = function
  (bQuadrat, bAchteck, hÜbergang) ->
    sqrt(sqr((sqrt(2.0) *. bQuadrat -. bAchteck) /. 2.0) +.
    sqr(hÜbergang));;

let AÜbergangsTrapez = function
  (bQuadrat, sAchteck, hÜTrapez) -> 0.5 *. (bQuadrat +. sAchteck) *.
    hÜTrapez;;

let AÜbergangsDreieck = function
  (sAchteck, hÜDreieck) -> 0.5 *. sAchteck *. hÜDreieck;;

let APyramidenDreieck = function
  (sAchteck, hPDreieck) -> 0.5 *. sAchteck *. hPDreieck;;

let AGesamt = function
  (aTrapez, aÜDreieck, aPDreieck) ->
    4.0 *. (aTrapez +. aÜDreieck) +. 8.0 *. aPDreieck;;
```

Den zu übersetzenden Definitionen sind hier zwei Hilfsdefinitionen vorangestellt. Ihre Bedeutung ergibt sich direkt aus der Definition.

Beim Lesen der Definitionen fällt auf, dass Rechenzeichen wie + oder * alle mit einem Punkt versehen sind: Statt + wird hier +. geschrieben. Dies ist eine Besonderheit von CAML. CAML achtet sehr strikt auf Typen. Für das Rechnen mit Gleitkommazahlen werden ausschließlich Rechenzeichen benutzt, die mit einem Punkt versehen sind. Es findet keine – wie in anderen Programmiersprachen übliche – Überladung von Rechenzeichen statt. Des Weiteren nimmt CAML keine Typumwandlung vor. CAML macht also nicht automatisch aus einer ganzen Zahl 4 die Gleitkommazahl 4.0, wenn es der Kontext erfordert. Der Benutzer muss selbst darauf achten, dass alle Terme typkonform erstellt werden. Schließlich ist zu beachten, dass jede Funktionsdefinition mit ;; abgeschlossen wird.

Es ist günstig, die Funktionsdefinitionen mit Hilfe eines Texteditors zu erstellen, sie in einer eigenen Datei mit der Dateierweiterung .ml abzuspeichern und mit dem CAML-Befehl „include“ (siehe CAML-Menü) einzulesen. Bei dieser Vorgehensweise erzeugt CAML die folgende Rückmeldung:

```
>       Caml Light version 0.71

#include "C:/caml/Programme/Dach1.ml";;
pi : float = 3.14159265
sqr : float -> float = <fun>
HöheÜbergangsTrapez : float * float * float -> float = <fun>
SeiteAchteck : float -> float = <fun>
HöhePyramidenDreieck : float * float -> float = <fun>
HöheÜbergangsDreieck : float * float * float -> float = <fun>
AÜbergangsTrapez : float * float * float -> float = <fun>
AÜbergangsDreieck : float * float -> float = <fun>
APyramidenDreieck : float * float -> float = <fun>
AGesamt : float * float * float -> float = <fun>
- : unit = ()
#
```

Wie ist diese Rückmeldung zu verstehen? CAML gibt zunächst den Aufruf zum Laden der Datei wieder (`#include "C:/caml/Programme/Dach.ml";;`). Anschließend liefert CAML eine Kurzbeschreibung der definierten Konstanten (erkennbar am Gleichheitszeichen) und Funktionen (erkennbar am Zusatz `= <fun>`). Den Abschluss bildet eine hier nicht weiter interessierende Typangabe (`- : unit = ()`) und ein Prompt (`#`). Für unsere Zwecke sind die Kurzbeschreibungen der Konstanten und Funktionen von Interesse.

Eine Konstantenbeschreibung wiederholt den Namen und Wert der Konstanten. Zusätzlich wird der Typ der Konstanten angegeben. Im vorliegenden Fall (`pi : float = 3.14159265`) wird hier angezeigt, dass die Konstante `pi` vom Typ `float` (floating point number) ist, also eine Gleitkommazahl darstellt, und den Wert `3.14159265` hat.

Bei einer Funktionsbeschreibung werden der Typ der möglichen Eingabeobjekte und der möglichen Ausgabeobjekte angegeben. Im Fall `HöheÜbergangsTrapez : float * float * float -> float = <fun>` wird beispielsweise angezeigt, dass ein Eingabeobjekt ein Tripel aus Gleitkommazahlen sein muss und dass ein Ausgabeobjekt eine Gleitkommazahl ist. Auf die genauere Bedeutung dieser Angaben wird noch eingegangen.

CAML liefert also für jede Definition eine Schnittstellenbeschreibung. Bei Konstanten wird der Typ der Konstanten ermittelt, bei Funktionen der Typ des Eingabebereichs und des Ausgabebereichs. Eine solche Schnittstellenbeschreibung mittels Typen wird auch *Signatur* genannt. Die Funktion `HöheÜbergangsTrapez` hat also die Signatur `float * float * float -> float.`

Mit Hilfe der definierten Funktionen sollen jetzt Berechnungen angestellt werden. Dies ist mit dem CAML-System interaktiv möglich: Man gibt im unteren Fenster den interessierenden Funktionsaufruf ein und erhält von CAML im oberen Fenster eine entsprechende Rückmeldung.

Beispiel:

Im Eingabefenster (unteres Fenster) erscheint:

```
HöheÜbergangsTrapez(4.0, 2.0, 3.0);;
```

Im Ausgabefenster (oberes Fenster) erscheint:

```
#HöheÜbergangsTrapez(4.0, 2.0, 3.0);;
- : float = 3.16227766017
```

CAML wiederholt den eingegebenen Funktionsaufruf und berechnet den Funktionswert (erkennbar an `- :`). Auch hier erzeugt CAML zusätzlich die Typangabe.

Im Folgenden werden einige Funktionsaufrufe und die berechneten Funktionswerte wiedergegeben.

```
HöheÜbergangsTrapez(4.0, 2.0, 3.0);;
- : float = 3.16227766017

SeiteAchteck(2.0);;
- : float = 0.828427123695

HöhePyramidenDreieck(2.0, 10.0);;
- : float = 10.0498756211

HöheÜbergangsDreieck(4.0, 2.0, 3.0);;
- : float = 3.51328133666

AÜbergangsTrapez(4.0, SeiteAchteck(2.0),
                 HöheÜbergangsTrapez(4.0, 2.0, 3.0));;
- : float = 7.63441361351

AÜbergangsDreieck(SeiteAchteck(2.0),
                  HöheÜbergangsDreieck(4.0, 2.0, 3.0));;
- : float = 1.45524877623

APyramidenDreieck(SeiteAchteck(2.0),
                  HöhePyramidenDreieck(2.0, 10.0));;
- : float = 4.16279477715

AGesamt(
  AÜbergangsTrapez(
    4.0,
    SeiteAchteck(2.0),
    HöheÜbergangsTrapez(4.0, 2.0, 3.0)),
  AÜbergandsDreieck(
    SeiteAchteck(2.0),
    HöheÜbergangsDreieck(4.0, 2.0, 3.0)),
  APyramidenDreieck(
    SeiteAchteck(2.0),
    HöhePyramidenDreieck(2.0, 10.0)));;
- : float = 69.6610077761
```

Die letzte Berechnung verdient besondere Beachtung. Für Schülerinnen und Schüler ist es zunächst ungewohnt, dass man Funktionen derart verschachtelt aufrufen kann. Man sollte daher im Unterricht auf diesen, für das funktionale Programmieren sehr wesentlichen Punkt genauer eingehen.

Hilfreich ist es hierbei, sich das Gleichwertigkeitsprinzip klarzumachen. Wir zeigen es zunächst anhand eines Beispiels. Der Term

```
AGesamt(7.63, 1.45, 4.16)
```

beschreibt (näherungsweise) den Gesamtflächeninhalt des Daches bei den gegebenen Größen (s. o.). Hierbei steht der Zahlenwert `7.63` für den Flächeninhalt des Übergangstrapezes. Diesen kann man ebenfalls durch den Term `AÜbergangsTrapez(4.0, 0.82, 3.16)` (näherungsweise) beschreiben. Ersetzt man jetzt den Zahlenwert `7.63` durch den Term `AÜbergangsTrapez(4.0, 0.82, 3.16)`, so erhält man den Term

```
AGesamt(AÜbergangsTrapez(4.0, 0.82, 3.16), 1.45, 4.16),
```

der ebenfalls den Gesamtflächeninhalt des Dachs (näherungsweise) beschreibt. Durch entsprechende Ersetzungsschritte kann man sich davon überzeugen, dass auch der folgende Term den Gesamtflächeninhalt des Daches beschreibt:

```
AGesamt(
  AÜbergangsTrapez(
    4.0,
    SeiteAchteck(2.0),
    HöheÜbergangsTrapez(4.0, 2.0, 3.0)),
  AÜbergandsDreieck(
    SeiteAchteck(2.0),
    HöheÜbergangsDreieck(4.0, 2.0, 3.0)),
  APyramidenDreieck(
    SeiteAchteck(2.0),
    HöhePyramidenDreieck(2.0, 10.0)));;
```

Das Gleichwertigkeitsprinzip erlaubt es, innerhalb eines Terms Gleiches durch Gleiches zu ersetzen, ohne den Wert des Terms zu verändern.

Gleichwertigkeitsprinzip:

Die linke und rechte Seite einer Definitionsgleichung sind gleichwertig, sie beschreiben dasselbe. Man kann sie infolgedessen beliebig austauschen, ohne den Wert eines Ausdrucks zu verändern.

Terme können beliebig komplex aufgebaut werden. Beim Aufbau eines Terms muss nur darauf geachtet werden, dass der Term das beschreibt, was er beschreiben soll.

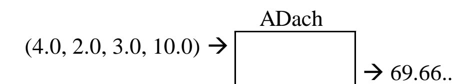
Vertiefungen:

Im Folgenden sollen jetzt weiterführende Überlegungen angestellt werden. Ziel ist es, die bisher erstellte Lösung zu verbessern. Hierbei wird auch das Schachteln von Funktionen vertieft.

Abstraktion durch Modularisierung

Die erstellten Funktionen zur Berechnung der Dachoberfläche haben den Nachteil, dass eine konkrete Berechnung die Eingabe eines recht komplexen Terms erfordert (s.o.). Dieser Nachteil soll jetzt behoben werden. Zunächst modellieren wir eine neue Funktion, die eine direkte Berechnung der Dachoberfläche beschreibt.

Modellierung:



Verhaltensbeschreibung: Die Funktion „ADach“ ordnet den Breiten des Grundquadrats und des Achtecks und den Höhen des Übergangsstücks und der Pyramide den Flächeninhalt des Dachs zu.

Bei der Definition abstrahieren wir den oben gezeigten Funktionsaufruf, indem wir diesen mittels der neuen Funktion „ADach“ beschreiben.

Definition:

```
ADach(bQuadrat, bAchteck, hÜbergang, hPyramide) =

AGesamt(
  AÜbergangsTrapez(
    bQuadrat,
    SeiteAchteck(bAchteck),
    HöheÜbergangsTrapez(bQuadrat, sAchteck, hÜbergang)),
  AÜbergandsDreieck(
    SeiteAchteck(bAchteck),
    HöheÜbergangsDreieck(bQuadrat, bAchteck, hÜbergang)),
  APyramidenDreieck(
    SeiteAchteck(bAchteck),
    HöhePyramidenDreieck(bAchteck, hPyramide)))
```

Die Implementierung dieser Definition erfolgt standardmäßig. Mit Hilfe von Funktionsaufrufen kann man jetzt direkt Flächenberechnungen vornehmen:

```
ADach(4.0, 2.0, 3.0, 10.0);;
- : float = 69.6610077761

ADach(5.0, 2.0, 3.0, 12.0);;
- : float = 85.5086823735

ADach(5.0, 4.0, 7.0, 12.0);;
- : float = 197.806902848
```

Generalisierung und Spezialisierung

Die Definitionen der Funktionen „AÜbergangsDreieck“ und „APyramidenDreieck“ sind strukturell identisch. Dieser Sachverhalt kann durch Einführung einer allgemeinen Funktion zur Berechnung des Flächeninhalts eines Dreiecks berücksichtigt werden.

Modellierung:

(0.82.., 10.04..) → [ADreieck] → 4.16..

Verhaltensbeschreibung: Die Funktion „ADreieck“ ordnet der Grundseite und der Höhe des Dreiecks den Flächeninhalt des Dreiecks zu.

Definition:

$$\text{ADreieck}(g, h) = \frac{1}{2} \cdot g \cdot h$$

Analog kann man eine allgemeine Funktion zur Berechnung des Flächeninhalts eines Trapezes einführen.

Modellierung:

(3.16.., 4.0, 0.82..) → [ATrapez] → 7.63..

Verhaltensbeschreibung: Die Funktion „ATrapez“ ordnet der Grundseite, Gegenseite und Höhe des Übergangstrapezes den Flächeninhalt des Trapezes zu.

Definition:

$$\text{ATrapez}(a, c, h) = \frac{1}{2} \cdot (a + c) \cdot h$$

Beim Berechnen der verschiedenen Höhen wird stets der Satz des Pythagoras benutzt. Hier kann man eine allgemeine Funktion zur Bestimmung der Länge der Hypotenuse eines rechtwinkligen Dreiecks einführen.

Modellierung:

(1.0, 10.0) → [LHypotenuse] → 10.04..

Verhaltensbeschreibung: Die Funktion „LHypotenuse“ ordnet den Längen der Katheten eines rechtwinkligen Dreiecks die Länge der Hypotenuse zu.

Definition:

$$\text{LHypotenuse}(a, b) = \sqrt{a^2 + b^2}$$

Wie sich mit Hilfe dieser verallgemeinerten Funktionen die Bestimmungsstücke beschreiben lassen, soll in der folgenden Übersicht aufgezeigt werden.

Übersicht:

Bestimmungsstück	Wert	funktionale Beschreibung
$b_{Quadrat}$	4.0	
$b_{Achteck}$	2.0	
$h_{Übergang}$	3.0	
$h_{Pyramide}$	10.0	
$s_{Achteck}$	0.82..	SeiteAchteck(2.0)
$h_{ÜTrapez}$	3.16..	LHypotenuse((4.0 - 2.0) / 2, 3.0)
$h_{ÜDreieck}$	3.51..	LHypotenuse(($\sqrt{2}$ *4.0 - 2.0) / 2, 3.0)
$h_{PDreieck}$	10.04..	LHypotenuse((2.0 / 2), 10.0)
$A_{ÜTrapez}$	7.63..	ATrapez(4.0, 2.0, 3.0)
$A_{ÜDreieck}$	1.45..	ADreieck(SeiteAchteck(2.0), LHypotenuse(($\sqrt{2}$ *4.0 - 2.0) / 2, 3.0))
$A_{PDreieck}$	4.16..	ADreieck(SeiteAchteck(2.0), LHypotenuse((2.0 / 2), 10.0))

Bestimmungsstück	Wert	funktionale Beschreibung

A_{Dach}	69.66..	AGesamt(ATrapez(4.0, 2.0, 3.0), ADreieck(SeiteAchteck(2.0), LHypotenuse(($\sqrt{2}$ *4.0 - 2.0) / 2, 3.0)), ADreieck(SeiteAchteck(2.0), LHypotenuse((2.0 / 2), 10.0)))

Arbeiten mit einem funktionalen Programmiersystem

Wir fassen abschließend das Charakteristische der funktionalen Programmierung zusammen.

Ein funktionales Programm besteht aus (allgemeinen) Funktionsdefinitionen und (speziellen) Funktionsaufrufen.

Funktionale Programmierung besteht somit darin, Funktionen zu konzipieren und mit ihrer Hilfe den interessierenden Weltausschnitt zu beschreiben.

Für den Umgang mit dem speziellen Programmiersystem CAML sei auf die Handbücher verwiesen (siehe Seite 116).

3.2 Das Funktionskonzept der funktionalen Programmierung

Partielle Funktionen

In der funktionalen Programmierung werden Funktionen als partielle Funktionen behandelt.

Die Schülerinnen und Schüler bringen aus dem Mathematikunterricht Kenntnisse über den Funktionsbegriff mit. An diese Kenntnisse kann zunächst angeknüpft werden.

Eine Funktion wird in der Mathematik (üblicherweise) durch eine Zuordnungsvorschrift sowie eine Beschreibung der Definitionsmenge und der Zielmenge festgelegt: Jedem Element der Definitionsmenge wird genau ein Element der Zielmenge zugeordnet.

Beispiel:

$f: R_0^+ \to R$ (Definitionsmenge, Zielmenge)

$f: x \mapsto \sqrt{x}$ (Zuordnungsvorschrift)

Welches Funktionskonzept wird in der funktionalen Programmierung (hier: CAML) benutzt? Aufschluss hierüber erhält man, wenn man sich die Rückmeldung von CAML zur Definition der Wurzelfunktion anschaut:

```
#let Wurzel = function x -> sqrt(x);;
Wurzel : float -> float = <fun>
```

CAML erzeugt Ein- und Ausgabebereiche für die Funktion `Wurzel`. Der Eingabebereich beschreibt hier aber nicht die Definitionsmenge der Funktion; negative Eingabewerte werden nicht ausgeschlossen. Der Eingabebereich stellt eine Obermenge der Definitionsmenge dar. Es gibt im vorliegenden Fall Eingabewerte, für die kein Funktionswert existiert. Die Funktion `Wurzel` , betrachtet als Funktion über der Menge der Gleitkommazahlen, ist somit keine total definierte, sondern eine partiell definierte Funktion. In der funktionalen Programmierung wird – anders als in der Mathematik – üblicherweise das Funktionskonzept „partielle Funktion“ benutzt. Der Grund hierfür ist einfach: Die Programmiersysteme sind nicht in der Lage, für beliebige Funktionen jeweils die korrekten Definitionsmengen zu bestimmen. Sie beschränken sich daher darauf, geeignete Obermengen dieser Definitionsmengen als Eingabebereiche festzulegen. Für die korrekte Wahl der Eingabeobjekte hat der Benutzer Sorge zu tragen.

Konstanten

Konstanten werden als spezielle Funktionen behandelt.

Beispiel:

```
pi : float = 3.14159265
```

Konstanten sind Funktionen ohne Eingabebereich.

Signaturen

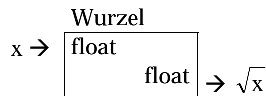
Die Beschreibung von Eingabe- und Ausgabebereichen erfolgt mit Hilfe von Typen.

Beispiel:

```
Wurzel : float -> float = <fun>
```

„float" ist der Bezeichner eines vordefinierten Standardtyps für Gleitkommazahlen. Eine Auflistung der Eingabe- und Ausgabebereiche einer Funktion mittels Typen bezeichnet man als *Signatur* der Funktion. Im Beispiel ist `float -> float` die Signatur der Funktion `Wurzel`. Die Signatur liefert eine genaue Schnittstellenspezifikation der betroffenen Funktion. Wir können sie in unser Modellierungskonzept wie folgt integrieren:

Modellierung:



Verhaltensbeschreibung: Bei Eingabe einer (nichtnegativen) Gleitkommazahl x liefert Wurzel(x) die Quadratwurzel von x. Die Eingabe einer negativen Zahl führt zu einer Fehlermeldung.

Im Mathematikunterricht begegnen den Schülerinnen und Schüler in erster Linie Funktionen mit einem Argument, das über einem Zahlbereich definiert ist. In der funktionalen Programmierung treten dagegen sehr oft Funktionen mit mehreren Argumenten auf, die z. T. über sehr komplexen Bereichen definiert sind. Wir werden hier nur Funktionen mit einem Argument betrachten. Dieses kann aber eine komplexe Struktur haben, wie etwa die Zusammenfassung `((2.0,3.0),(5.0,1.0))` der Koordinatenpaare zweier Punkte.

Beispiel:

```
#let Abstand = function
  ((a,b),(c,d)) -> sqrt((c -. a) *. (c -. a) +. (d -. b) *. (d -. b));;
Abstand : (float * float) * (float * float) -> float = <fun>
```


Zulässige Eingaben der Funktion „Abstand" sind Paare bestehend aus Paaren von Objekten vom Typ „float". Wozu die Erzeugung der Signatur gut ist, zeigt das folgende

Beispiel:

```
#Abstand(2.0,3.0,5.0,1.0);;
Toplevel input:
>Abstand(2.0,3.0,5.0,1.0);;
>        ^^^^^^^^^^^^^^^
This expression has type float * float * float * float,
but is used with type (float * float) * (float * float).

#Abstand((2,3),(5,1));;
Toplevel input:
>Abstand((2,3),(5,1));;
>         ^
This expression has type int,
but is used with type float.

#Abstand((2.0,3.0),(5.0,1.0));;
- : float = 3.60555127546
```

Hier wurden zwei fehlerhafte und ein korrekter Funktionsaufruf eingegeben. Anhand der Signaturen können die Fehler sofort vom CAML-System erkannt werden.

Beim ersten Funktionsaufruf wurde die Struktur „Paar bestehend aus zwei Paaren" nicht beachtet. Der eingegebene Funktionsaufruf hat die Struktur „Tupel bestehend aus vier Objekten vom Typ float".

Beim zweiten Funktionsaufruf wurden ganze Zahlen (vom Typ „int") an Stelle von Gleitkommazahlen (vom Typ „float") benutzt.

Die sehr strikte Verarbeitung von Typen in CAML lässt frühzeitig Fehler erkennen. Zunächst erscheint dies lästig. Bei größeren Programmieraufgaben sind diese Typinformationen aber sehr hilfreich, insbesondere wenn Fehler gesucht werden müssen.

Typvariablen

CAML erzeugt möglichst allgemeine Ein- und Ausgabebereiche. Anhand der im Funktionsterm benutzten Operationen ermittelt CAML diese Bereiche. Im Fall der Funktion „Wurzel" (s.o) erkennt CAML an der Operation „sqrt", dass hier Objekte vom Typ „float" verarbeitet werden sollen. Kann CAML anhand der Operationen einem Bereich keinen speziellen Typ zuordnen, so wird der nicht näher bekannte Typ mit Hilfe einer Typvariablen beschrieben. Die Funktion „Vertausche" liefert hierfür ein

Beispiel:

```
#let Vertausche = function
  (x, y) → (y, x);;
Vertausche : 'a * 'b -> 'b * 'a = <fun>
```

Hier werden vom CAML-System automatisch Typvariablen ‘a und ‘b für die (evtl. verschiedenen) Typen der Komponenten des Paares (x, y) generiert. Bei der Erzeugung des

Ausgabebereichs müssen diese eingeführten Typvariablen natürlich in konformer Weise berücksichtigt werden.

Funktionsterme

Funktionen können beliebig geschachtelt werden (Kompositionsprinzip). Zur Modellierung von Fallunterscheidungen wird ein if-then-else-Konstrukt verwendet.

Beim Schachteln von Funktionen muss darauf geachtet werden, dass die jeweiligen Typen berücksichtigt werden.

Beim Aufstellen von Funktionstermen werden i.a. auch vordefinierte Funktionen benutzt. Vordefinierte Funktionen werden durch die vordefinierten Typen festgelegt. Eine nicht vollständige Übersicht findet man weiter unten.

Funktionen können auch über Fallunterscheidungen festgelegt werden. Hierzu gibt es in CAML (wie in allen gängigen funktionalen Programmiersprachen) ein if-then-else-Konstrukt. Dieses Konstrukt kann als eine vordefinierte Funktion mit der folgenden Signatur angesehen werden:

```
if-then-else: bool * 'a * 'a -> 'a = <fun>
```

Funktionsterme sind somit aus Funktionen (vordefinierte Funktionen, if-then-else-Funktion, neu-definierte Funktionen, vordefinierte Konstanten, neu-definierte Konstanten) und Variablen aufgebaut.

Vordefinierte einfache Datentypen in CAML

Ein Datentyp wird festgelegt durch eine Menge von Objekten sowie Operationen zur Bearbeitung der Objekte.

Im Folgenden werden einige fundamentale vordefinierte Datentypen in CAML aufgelistet Die Übersicht stellt aber keine vollständige Auflistung dar. Für detailliertere Beschreibungen sei auf das Handbuch verwiesen.

Übersicht:

Vordefinierte einfache Datentypen in CAML

Datentyp bool:

Objekte:	true, false
Operationen:	not : bool → bool
	&, or : bool, bool → bool
	=, <> : bool, bool → bool
	... siehe Handbuch ...

Datentyp int:

Objekte:	ganze Zahlen, z. B.: 3, -4
Operationen:	- : int → int
	+, -, *, /, mod : int, int → int
	=, <> : int, int → bool
	<, <=, >, >= : int, int → bool
	... siehe Handbuch ...

Man beachte, dass die Operationen = und <> nicht mit einem Punkt versehen werden.

Datentyp float:

Objekte:	Dezimalzahlen, z. B.: 3.14, -6.566
Operationen:	-. : float → float
	+., -., *., /. : float, float → float
	=, <>, <., <=., >., >=. : float, float → bool
	sqrt : float → float
	... siehe Handbuch ...

Datentyp char:

Objekte:	Zeichen, z. B.: ‘c’
Operationen:	... siehe Handbuch ...

Datentyp string:

Objekte:	Zeichenketten, z. B.: „Hund“
Operationen:	... siehe Handbuch ...

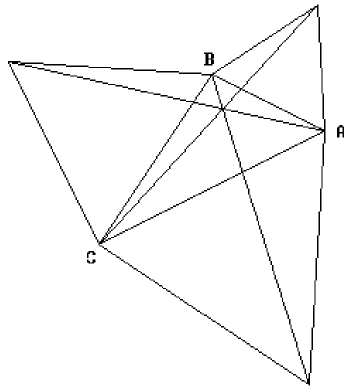
3.3 Funktionale Modellierung komplexer Ein- und Ausgabesituationen

Problem:

Optimaler Standort

Ein Unternehmen hat drei Niederlassungen in verschiedenen Städten in Deutschland. Das Unternehmen beabsichtigt, ein Zentrallager an einem neuen Standort zu bauen. Der Standort des Zentrallagers soll so gewählt werden, dass die entstehenden Fahrtkosten bei der Belieferung der Niederlassungen möglichst klein werden. Es kann davon ausgegangen werden, dass die Niederlassungen gleich oft beliefert werden müssen. Gesucht ist ein Verfahren, mit dem man den optimalen Standort bestimmen kann.

Lösung (geometrisches Konstruktionsverfahren):



Die Standorte bilden die Eckpunkte eines Dreieck ABC. Wir betrachten zunächst den Fall, dass keiner der drei Winkel im Dreieck ABC größer oder gleich 120° ist.

Über den drei Seiten werden jeweils gleichseitige Dreiecke (nach außen hin) konstruiert.

Die entstehenden neuen Eckpunkte werden mit den ihnen gegenüberliegenden Eckpunkten des Ausgangsdreiecks verbunden („Torricelli-Konstruktion").

Diese Verbindungslinien schneiden sich in einem Punkt - dem sog. **Fermatpunkt**. Der Fermatpunkt ist der gesuchte Punkt im Dreieck.

Ist einer der Winkel im Dreieck ABC größer oder gleich 120° (wir nennen einen solchen Winkel hier stumpf), so ist der gesuchte Punkt der, bei dem der stumpfe Winkel vorliegt.

Wie dieses Konstruktionsverfahren von Schülerinnen und Schülern entdeckt werden kann und wie die Optimalitätseigenschaft des Fermatpunkts bewiesen werden kann wird in [Euteneuer 99] und [Müller-Ewertz 99] gezeigt. Ziel der folgenden Betrachtungen soll es sein, den optimalen Standort (Fermatpunkt) bei drei beliebig gegebenen Ausgangspunkten automatisiert rechnerisch zu bestimmen.

Überprüfung der Torricelli-Konstruktion

Zunächst soll mit Hilfe eines funktionalen Programms die geometrisch konstruierte Lösung des Optimierungsproblems überprüft werden. Insbesondere soll dieses Programm die Möglichkeit eröffnen, den eingangs diskutierten Sonderfall (Dreieck mit stumpfem Winkel) genauer zu untersuchen.

Wir modellieren zunächst das Ein-/Ausgabe-Verhalten einer Funktion, die die Gesamtentfernung bei beliebigen Niederlassungskonstellationen beschreibt.

Modellierung:

```
                GesamtEntfernung
(X,A,B,C) → | (float * float) * (float * float) *  |
            | (float * float) * (float * float)    |
            |                                float | → GesamtEntfernung(X,A,B,C)
```

Verhaltensbeschreibung: Bei Eingabe eines beliebigen Standortes X und dreier verschiedener Niederlassungen A,B,C liefert die Funktion „GesamtEntfernung" die Gesamtentfernung der drei Niederlassungen zum Standort X.

Man beachte, dass das Eingabeobjekt hier ein Quadrupel bestehend aus vier Zahlenpaaren ist. Dies zeigt sich in der im Kasten angedeuteten Signatur: Gesamtentfernung: (float * float) * (float * float) * (float * float) * (float * float) → float. Jedes Zahlenpaar beschreibt einen Punkt bzw. dessen Ortsvektor.

Zur Erstellung der Funktionsdefinition betrachten wir zunächst den Spezialfall, dass sich die Niederlassungen (in einem geeigneten Koordinatensystem) an den Orten A(3 | 1), B(1 | 2), C(-1 | -1) befinden. Für diesen Fall ergibt sich die folgende Formel zur Berechnung der Gesamtentfernung:

$$\underbrace{\sqrt{(x_1-1)^2+(x_2-2)^2}}_{\text{Abstand XA}}+\underbrace{\sqrt{(x_1-3)^2+(x_2-1)^2}}_{\text{Abstand XB}}+\underbrace{\sqrt{(x_1+1)^2+(x_2+1)^2}}_{\text{Abstand XC}}.$$

Diese lässt sich direkt verallgemeinern.

Definition:

$$\text{GesamtEntfernung}((x1, x2), (a1, a2), (b1, b2), (c1, c2)) =$$
$$\sqrt{(x1 - a1)^2 + (x2 - a2)^2} + \sqrt{(x1 - b1)^2 + (x2 - b2)^2} + \sqrt{(x1 - c1)^2 + (x2 - c2)^2}$$

Die entwickelte Definition ist zwar korrekt, die Korrektheit ist aber für Außenstehende schwer nachzuvollziehen. Es soll daher eine zweite, verständlichere Definition entwickelt werden.

Die Strukturierung der Formel zeigt, dass zur Berechnung der Gesamtentfernung dreimal eine Abstandsberechnung durchgeführt werden muss. Diese Tatsache soll im Folgenden berücksichtigt werden. Wir führen eine Hilfsfunktion Abstand ein, mit deren Hilfe die erforderlichen Abstandsberechnungen durchgeführt werden sollen.

Modellierung:

	Abstand	
(P,Q) →	(float * float) * (float * float) float	→ Abstand(P,Q)

Verhaltensbeschreibung: Bei Eingabe zweier Punkte P und Q (bzw. deren Ortsvektoren) liefert die Funktion „Abstand“ den Abstand der beiden Punkte.

Definition:

$$\text{Abstand}((p1, p2), (q1, q2)) = \sqrt{(p1 - q1)^2 + (p2 - q2)^2}$$

Mit dieser zusätzlich eingeführten Funktion „Abstand” lässt sich die Berechnung der Gesamtentfernung wie folgt beschreiben.

Definition:

GesamtEntfernung(X,A,B,C) = Abstand(X,A) + Abstand(X,B) + Abstand(X,C)

Implementierung:

```
let sqr = function
  x -> x *. x;;
sqr : float * float -> float = <fun>

let Abstand = function
  ((p1, p2), (q1, q2)) -> sqrt(sqr(q1 -. p1) +. sqr(q2 -. p2));;
Abstand : (float * float) * (float * float) -> float = <fun>

let GesamtEntfernung = function
  (x,a,b,c) -> Abstand(x,a) +. Abstand(x,b) +. Abstand(x,c);;
GesamtEntfernung :
 (float * float) * (float * float) * (float * float) * (float * float) ->
 float = <fun>
```


Mit Hilfe dieser Funktionen kann man jetzt Abstandsberechnungen durchführen:

```
GesamtEntfernung((5.0,2.0),(3.0,1.0),(5.0,1.0),(7.0,1.0));;
- : float = 5.472135955
GesamtEntfernung((5.0,1.0),(3.0,1.0),(5.0,1.0),(7.0,1.0));;
- : float = 4.0
```

Berechnung des Fermatpunktes

Im Folgenden soll der optimale Standort mit Hilfe eines funktionalen Programms bestimmt werden. Wir modellieren hierzu die folgende Funktion:

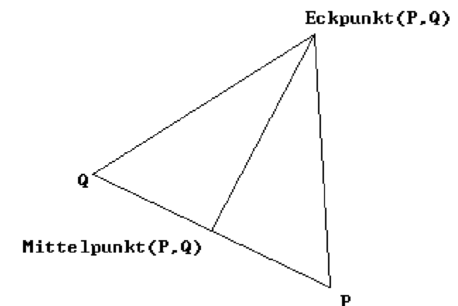
Modellierung:

	Fermatpunkt	
(A,B,C) →	(float*float)*(float*float)*(float*float) float*float	→ Fermatpunkt(A,B,C)

Verhaltensbeschreibung: Bei Eingabe von drei verschiedenen Niederlassungen (Punkten bzw. deren Ortsvektoren) A,B,C liefert die Funktion „Fermatpunkt“ den Standort, bei dem die Gesamtentfernung der drei Niederlassungen zum Standort minimal ist. *Wir gehen im Folgenden stets davon aus, dass die Punkte ABC in dieser Reihenfolge ein standardmäßig bezeichnetes Dreieck ergeben.*

Die Konstruktion (s. o.) zeigt auf, welche Teilprobleme zu lösen und somit mittels Funktionen zu beschreiben sind. Zum einen muss eine gegebene Strecke zu einem gleichseitigen Dreieck ergänzt werden. Zum anderen muss der Schnittpunkt von zwei Strecken (Geraden) bestimmt werden. Dabei zeigt sich, dass eine Implementierung von Grundoperationen der Vektorrechnung notwendig ist (s. Seite 48). Erst anschließend ist die Implementierung der Funktion „Fermatpunkt” möglich (s. Seite 53).

Wir betrachten in einem ersten Schritt die Konstruktion eines gleichseitigen Dreiecks bei gegebener Strecke PQ.



Hierzu bestimmen wir den Mittelpunkt der Strecke PQ. Durch diesen Mittelpunkt errichten wir eine Orthogonale zu PQ. Auf dieser Orthogonalen konstruieren wir den fehlenden Eckpunkt des gleichseitigen Dreiecks. Es ist zu beachten, dass die Strecke PQ zu zwei gleichseitigen Dreiecken ergänzt werden kann. Hier soll nur das gleichseitige Dreieck betrachtet werden, das „rechts" von PQ liegt, wenn man die Strecke in Richtung von P nach Q durchläuft.

Zur Bestimmung der Koordinaten der modellierten Punkte verwenden wir Vektorrechnung. Wir verzichten bei der Darstellung von Vektoren auf die Pfeile. Der Ortsvektor eines Punktes P wird hier stets mit p bezeichnet. Wir beschreiben den Vektor, der durch eine Drehung um 90° nach rechts aus v hervorgeht, mit $v^{\perp}$.

Wir modellieren und definieren zunächst Funktionen zur Bestimmung der Hilfspunkte.

Modellierung:

Mittelpunkt

(p,q) → | (float*float)*(float*float) float*float | → Mittelpunkt(p,q)

Verhaltensbeschreibung: Bei Eingabe der Ortsvektoren von zwei Punkten P und Q liefert die Funktion „Mittelpunkt" den Ortsvektor des Mittelpunkts der Strecke PQ.

Definition:

Mittelpunkt(p, q) = 0.5(p + q)

Modellierung:

Eckpunkt

(p,q) → | (float*float)*(float*float) float*float | → Eckpunkt(p,q)

Verhaltensbeschreibung: Bei Eingabe der Ortsvektoren von zwei Punkten P und Q liefert die Funktion „Eckpunkt" den Ortsvektor eines der beiden Punkte, die die Strecke PQ zu einem gleichseitigen Dreieck ergänzen. Der Eckpunkt wird hier so gewählt, dass er „rechts" von PQ liegt, wenn man die Strecke in Richtung von P nach Q durchläuft.

Definition:

Eckpunkt(p, q) = Mittelpunkt(p, q) + $0.5\sqrt{3}\,(q - p)^{\perp}$

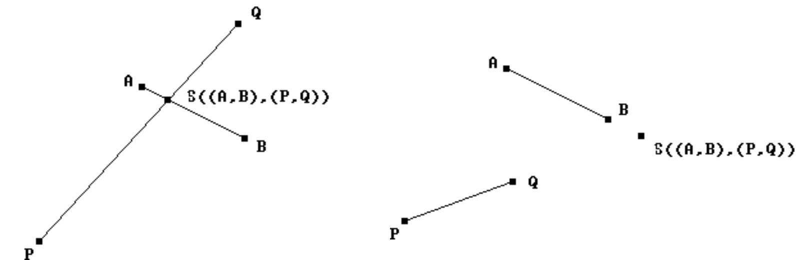
In einem nächsten Schritt ist das Problem „Schnitt von zwei Strecken" zu lösen.

Modellierung:

Schnittpunkt

((a,b),(p,q)) → | ((float*float)*(float*float))* ((float*float)*(float*float)) float*float | → Schnittpunkt((a,b),(p,q))

Verhaltensbeschreibung: Bei Eingabe von zwei nichtparallelen Strecken, gegeben durch je zwei Punkte bzw. durch ihre Ortsvektoren, liefert die Funktion „Schnittpunkt" den Schnittpunkt der beiden Strecken.



Um eine Definition herzuleiten, benötigen wir Wissen über Geradendarstellungen.

Es gilt (in Vektordarstellung):

$$g_{AB} : x = a + r(b - a)$$
$$g_{PQ} : x = p + s(q - p)$$

Die Schnittbedingung lautet: $a + r(b - a) = p + s(q - p)$

Multiplikation mit $(q - p)^{\perp}$ und Auflösen nach r liefert:

$$r = \frac{(p-a)\cdot(q-p)^{\perp}}{(b-a)\cdot(q-p)^{\perp}}$$

Hierbei muss vorausgesetzt werden, dass der Nenner ungleich Null ist. Dies ist erfüllt, wenn die Punkte A und B bzw. P und Q verschieden sind und wenn die Strecken AB und PQ nicht parallel verlaufen.

Durch Einsetzen in $g_{AB}: x = a + r(b-a)$ erhält man den Ortsvektor des gesuchten Schnittpunktes für den Fall, dass der Nenner ungleich Null ist:

$$a + \frac{(p-a)\cdot(q-p)^{\perp}}{(b-a)\cdot(q-p)^{\perp}}(b-a)$$

Definition:

$$\text{Schnittpunkt}((a, b), (p, q)) = a + \frac{(p-a)\cdot(q-p)^{\perp}}{(b-a)\cdot(q-p)^{\perp}}(b-a)$$

Jetzt kann eine erste Definition der Funktion „Fermatpunkt" gegeben werden. Diese berücksichtigt aber nur den Fall, dass keiner der drei Winkel im Dreieck 120° beträgt oder übersteigt.

Definition:

Fermatpunkt(a, b, c) = Schnittpunkt((c, Eckpunkt(a, b)), (a, Eckpunkt(b, c)))

Grundoperationen der Vektorrechnung

Zur Implementierung der oben erstellten Funktionen in CAML müssen zunächst die Vektoroperationen mittels geeigneter Funktionen realisiert werden.

Modellierung:

Add
(p, q) → (float*float)*(float*float) | float*float → Add(p, q)

Verhaltensbeschreibung: Bei Eingabe von zwei Vektoren p und q liefert die Funktion „Add" die Summe der beiden Vektoren.

Definition:

Add((p1, p2), (q1, q2)) = (p1 + q1, p2 + q2)

Modellierung:

Mul
(c, p) → float*(float*float) | float*float → Mul(c, p)

Verhaltensbeschreibung: Bei Eingabe einer Zahl c und eines Vektors p liefert die Funktion „Mul" das Produkt von c und p.

Definition:

Mul(c, (p1, p2)) = (c*p1, c*p2)

Modellierung:

Sub
(p, q) → (float*float)*(float*float) | float*float → Sub(p, q)

Verhaltensbeschreibung: Bei Eingabe von zwei Vektoren p und q liefert die Funktion „Sub" die Differenz der beiden Vektoren.

Definition:

Sub(p, q) = Add(p, Mul(-1, q))

Modellierung:

Orthogonale
p → (float*float) | float*float → $p^{\perp}$

Verhaltensbeschreibung: Bei Eingabe eines Vektors p liefert die Funktion „Orthogonale" den Vektor, den man erhält, wenn man p um 90° nach rechts dreht.

Definition:

Orthogonale(p1, p2) = (p2, -p1)

Modellierung:

Skalarprodukt
(p, q) → (float*float)*(float*float) | float*float → Skalarprodukt(p, q)

Verhaltensbeschreibung: Bei Eingabe von zwei Vektoren p und q liefert die Funktion „Skalarprodukt" das Skalarprodukt der beiden Vektoren.

Definition:

Skalarprodukt((p1, p2), (q1, q2)) = p1*q1 + p2*q2

Implementierung der Funktionen zur Bestimmung des Fermatpunktes

Mit den oben erstellten Hilfsfunktionen zur Vektorrechnung lassen sich die Definitionen der Funktionen „Mittelpunkt" und „Eckpunkt" wie folgt darstellen.

Definition:

Mittelpunkt(p, q) = Mul(0.5, Add(p, q))

Definition:

Eckpunkt(p, q) = Add(Mittelpunkt(p, q), Mul(0.5* $\sqrt{3}$, Orthogonale(Sub(q, p))))

Implementierung:

```
let Abstand = function
  ((p1, p2), (q1, q2)) -> sqrt(sqr(q1 -. p1) +. sqr(q2 -. p2));;

let Add = function
  ((p1, p2), (q1, q2)) -> (p1 +. q1, p2 +. q2);;

let Mul = function
  (c, (p1, p2)) -> (c *. p1, c *. p2);;

let Sub = function
  (p, q) -> Add(p, Mul(-1.0, q));;

let Orthogonale = function
  (p1, p2) -> (p2, -. p1);;

let Skalarprodukt = function
  ((p1, p2), (q1, q2)) -> p1 *. q1 +. p2 *. q2;;

let Mittelpunkt = function
  (p, q) -> Mul(0.5, Add(p, q));;

let Eckpunkt = function
  (p, q) ->
    Add(Mittelpunkt(p, q), Mul(sqrt(3.0) *. 0.5, Orthogonale(Sub(q, p))));;

let Schnittpunkt = function
  ((a, b), (p, q)) ->
    Add(a,
          Mul(
              Skalarprodukt(Sub(p, a), Orthogonale(Sub(q, p)))
              /.
              Skalarprodukt(Sub(b, a), Orthogonale(Sub(q, p))),
                             Sub(b, a)));;

let Fermatpunkt = function
  (a, b, c) -> Schnittpunkt(c, Eckpunkt(a, b),(a, Eckpunkt(b, c));;
```


Die Funktionen werden getestet und die berechneten Werte mit den zeichnerisch ermittelten Werten verglichen. Kontrollberechnungen ergänzen die Korrektheitsanalysen.

```
let A = (3.0, 1.0);;
A : float * float = 3.0, 1.0

let B = (1.0, 2.0);;
B : float * float = 1.0, 2.0

let C = (-1.0, -1.0);;
C : float * float = -1.0, -1.0

Mittelpunkt(A,B);;
- : float * float = 2.0, 1.5

Eckpunkt(A,B);;
- : float * float = 1.0, 2.0

Abstand(A, Eckpunkt(A,B));;
- : float = 2.2360679775

Abstand(B, Eckpunkt(A,B));;
- : float = 2.2360679775

Schnittpunkt((A,B),(A,C));;
- : float * float = 3.0, 1.0

Fermatpunkt(A,B,C);;
- : float * float = 1.19076794658, 1.39818425622
```

Der Fall stumpfwinkliger Dreiecke

Abschließend soll der Fall eines stumpfwinkligen Dreiecks (d. h.: einer der Winkel beträgt mindestens 120°) betrachtet werden.

Zunächst werden Funktionen konzipiert, mit deren Hilfe ein stumpfer Winkel ermittelt werden kann.

Modellierung:

```
       Betrag
p → | (float*float)            |
    |             float*float  | → | p |
```

Verhaltensbeschreibung: Bei Eingabe eines Vektors p liefert die Funktion „Betrag" den Betrag des Vektors p.

Definition:

Betrag(p) = $\sqrt{p \cdot p}$

Modellierung:

```
           Winkel
(p, q) → | (float*float)*(float*float) |
         |                      float  | → Winkel(p, q)
```

Verhaltensbeschreibung: Bei Eingabe von zwei Vektoren p und q liefert die Funktion „Winkel" den Winkel zwischen den beiden Vektoren.

Definition:

Winkel(p, q) = $\arccos\left(\frac{p \cdot q}{|p||q|}\right)$

Modellierung:

```
            stumpf
(a,b,c) → | (float*float)*(float*float)*(float*float) |
          |                                     bool  | → stumpf(a,b,c)
```

Verhaltensbeschreibung: Bei Eingabe der Ortsvektoren von drei Punkten A,B,C entscheidet die Funktion „stumpf", ob der Winkel bei B mindestens 120° beträgt.

Definition:

stumpf(a, b, c) = (Winkel(a-b, c-b) $\geq 2\pi / 3$)

Bei dieser Modellierung wurde der vordefinierter Datentyp „bool" für Wahrheitswerte benutzt.

Die Funktion „Fermatpunkt" kann jetzt allgemein definiert werden. Hierbei wird ein „if-then-else-Konstrukt" zur Erzeugung einer Fallunterscheidung benutzt, das in analoger Form auch in CAML zur Verfügung steht.

Definition:

```
Fermatpunkt(a, b, c) =
 if stumpf(a, b, c)
 then b
 else if stumpf(b, c, a)
      then c
      else if stumpf(c, a, b)
           then a
           else Schnittpunkt((c, Eckpunkt(a, b)), (a, Eckpunkt(b, c)))
```

Implementierung:

```
let Betrag = function
  p -> sqrt(Skalarprodukt(p, p));;

let Winkel = function
  (p, q) -> acos(Skalarprodukt(p, q) /. (Betrag(p) *. Betrag(q)));;

let stumpf = function
  (a, b, c) -> (Winkel(Sub(a, b), Sub(c, b)) >=. (2.0 /. 3.0 *. pi));;

let Fermatpunkt = function
  (a, b, c) ->
  if stumpf(a, b, c)
  then b
  else if stumpf(b, c, a)
       then c
       else if stumpf(c, a, b)
            then a
            else Schnittpunkt((c, Eckpunkt(a,b)), (a, Eckpunkt(b, c)));;
```

Testaufrufe:

```
Fermatpunkt((3.0, 1.0), (1.0, 2.0), (-1.0, -1.0));;
- : float * float = 1.19076794658, 1.39818425622

Fermatpunkt((0.0, 0.0), (4.0, 0.0), (2.0, 0.0));;
- : float * float = 2.0, 0.0

Fermatpunkt((0.0, 0.0), (4.0, 0.0), (2.0, 1.0));;
- : float * float = 2.0, 1.0

Fermatpunkt((0.0, 0.0), (4.0, 0.0), (2.0, 2.0));;
- : float * float = 2.0, 1.15470053838
```

Raum für Notizen

4 Mit Funktionen kann man Datentypen und Algorithmen beschreiben

Inhalte:

- Datentypen: Ein Datentyp wird festgelegt durch eine Menge von Objekten sowie Operationen zur Bearbeitung der Objekte.
- Komplexe Datenobjekte werden mit Hilfe von Konstruktoren aus einfacheren aufgebaut. Mit Hilfe von Selektoren kann man auf die einzelnen Bestandteile eines komplexen Datenobjektes zugreifen.
- Datenstruktur „Liste“
- Reduktionskonzept: Die Definition von Funktionen erfolgt mittels Reduktionsregeln. Diese stellen die Algorithmen im hier dargelegten Programmierkonzept dar. Die Berechnung von Funktionswerten erfolgt durch wiederholte Anwendung der Reduktionsregeln.
- Verfahren der rekursiven Problemreduktion: Man versucht, Problemreduktionsschemata zu entwerfen. Eine Strategie besteht darin, das Problem auf ein sich entsprechendes, aber „verkleinertes“ Problem zu reduzieren. Die Problemreduktionsschemata werden mit Hilfe von Reduktionsregeln dargestellt.

Bemerkungen:

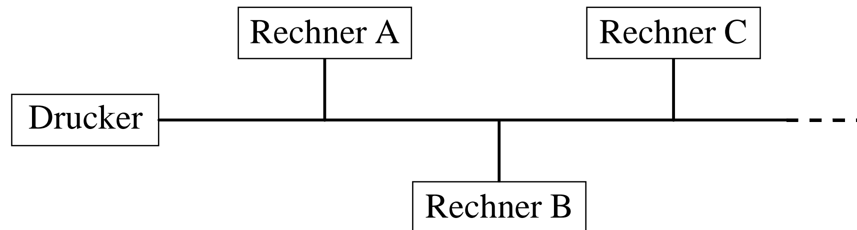
Wir verwenden hier systematisch die Vorstellung von Reduktionsregeln, um Funktionsdefinitionen darzustellen. Dieses Konzept lässt sich auf recht einfache Weise fundieren und formalisieren (vgl. [Avenhaus 95]).

Zur Erzeugung komplexer funktionaler Algorithmen benötigt man Funktionskomposition, Fallunterscheidung und Rekursion. Diese Konzepte bereiten Schülerinnen und Schülern keine großen Schwierigkeiten, wenn sie behutsam eingeführt werden. Zur Behandlung der Rekursion empfiehlt es sich, systematisch Korrektheitsüberlegungen anzustellen (s. S. 67ff).

4.1 Modellierung des Datentyps „Warteschlange“

Problem:

Ein Drucker kann von mehreren Rechnern mit Druckaufträgen versorgt werden.



Hierbei kann es vorkommen, dass neue Aufträge erteilt werden, während der Drucker noch einen Auftrag bearbeitet. Man benötigt eine Art Zwischenspeicher, in dem die neu hinzukommenden Aufträge abgelegt werden, bis sie vom Drucker bearbeitet werden können. Dieser Zwischenspeicher wird im Folgenden konzipiert.

Lösung:

Wir erfassen zunächst allgemein die Eigenschaften, die dieser Zwischenspeicher haben soll. Der Zwischenspeicher hat die Struktur einer Warteschlange. Diese arbeitet nach dem FIFO-Prinzip: Wer zuerst ankommt („first in“), wird zuerst bearbeitet („first out“). Realisierungen dieser Struktur findet man auch im Alltag: Waschstraße, Postschalter, ...

Im Folgenden wird ein Datentyp „Schlange“ allgemein (d. h. implementationsunabhängig) modelliert, mit dessen Hilfe man den Zwischenspeicher abstrakt beschreiben kann. Hierbei wird das Konzept „Datentyp“ wie folgt benutzt:

Ein Datentyp wird festgelegt durch eine Menge von Objekten sowie Operationen zur Bearbeitung der Objekte.

Die Operationen werden hier mit Hilfe von Funktionen modelliert.

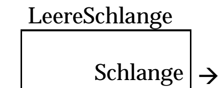
Datentyp Schlange

Objekte:

(Warte-) Schlangen sind endliche, aber beliebig lange Folgen / Sequenzen von Objekten (z. B. Druckaufträge: A C D B B A).
Der Datentyp der Objekte der Schlange wird hier nicht näher spezifiziert.

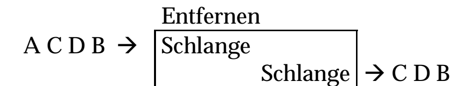
Operationen:

Modellierung:



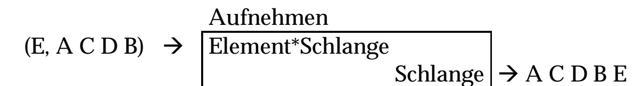
Verhaltensbeschreibung: Es wird eine leere Schlange erzeugt.

Modellierung:



Verhaltensbeschreibung: Das erste Element der Schlange wird entfernt.

Modellierung:



Verhaltensbeschreibung: Das neue Objekt wird als letztes Element in die Schlange aufgenommen.

Es stellt sich jetzt das Problem, wie man Schlangen implementieren kann. Man benötigt eine Datenstruktur, die beliebig lange Schlangen beschreiben kann. Funktionale Programmiersprachen stellen die Datenstruktur „Liste“ zur Verfügung. Diese eignet sich sehr gut, den skizzierten Datentyp zu realisieren. Bevor diese Realisierung hier aufgezeigt wird, soll zunächst das Listenkonzept von CAML vorgestellt werden.

4.2 Das Listenkonzept (von CAML)

Eine Liste in CAML besteht aus einer beliebigen, aber endlichen Anzahl von Elementen, die alle den gleichen Typ haben. Hier einige Beispiele mit ihren CAML-Typangaben

Listenobjekt	**Typ**
[1; 2; 3; 4]	int list
[„ein“; „schöner“; „Hund“]	string list
[ ]	‘a list
[(„Marschall“, 8); („Ronaldo“, 6)]	(string * int) list
[[1; 2]; [3]; [ ]; [4; 3; 2; 1]]	(int list) list

Datenstruktur: Liste

Konstruktoren:

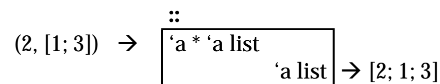
(a) leere Liste

Modellierung:

Verhaltensbeschreibung: [ ] erzeugt eine leere Liste.

(b) Hinzufügen eines Elements

Modellierung:



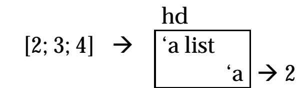
Verhaltensbeschreibung: Die Funktion **::** fügt ein Objekt vorne in eine Liste ein.

Beachte: Die vordefinierte Operation **::** ist für beliebige Objekttypen definiert. In der gezeigten Modellierung wird ihr Verhalten für Objekte vom Typ „int“ aufgezeigt.

Selektoren:

(a) Listenkopf (head)

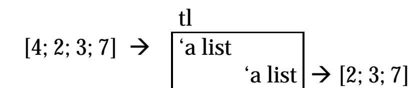
Modellierung:



Verhaltensbeschreibung: „hd“ liefert das erste Listenelement (falls die Liste nicht leer ist).

(b) Listenrumpf (tail)

Modellierung:



Verhaltensbeschreibung: „tl“ entfernt das erste Listenelement (falls die Liste nicht leer ist).

Beachte: „hd“ und „tl“ sind vordefinierte Funktionen.

Zur Bedeutung der Konstruktoren und Selektoren

Mit Hilfe der Konstruktoren lässt sich jede Liste aufbauen / *konstruieren*.

Beispiel: 4 :: (3 :: (1 :: (7 :: [ ]))) erzeugt [4; 3; 1; 7]

Mit Hilfe der Selektoren lässt sich jedes Listenelement aus einer Liste herausgreifen / *selektieren*.

Beispiel: hd(tl(tl([4; 3; 1; 7]))) erzeugt 1

Allgemein lässt sich die Bedeutung von Konstruktoren und Selektoren wie folgt beschreiben:

Komplexe Datenobjekte werden mit Hilfe von Konstruktoren aus einfacheren aufgebaut. Mit Hilfe von Selektoren kann man auf die einzelnen Bestandteile eines komplexen Datenobjektes zugreifen.

Wir illustrieren abschließend die Verarbeitung von Listen mit Hilfe der oben beschriebenen Konstruktoren und Selektoren anhand einiger Beispiele.

Term	**Wert des Terms**
3 :: [2; 1]	[3; 2; 1]
1 :: 2 :: 3 :: [ ]	[1; 2; 3]
hd (tl ([3; 5; 2]))	5
tl (4 :: 6 :: [3 ; 5])	[6; 3; 4]
hd ([[] ; [1]; [1; 2]])	[]
„A“ :: tl ([„A“; „D“; „A“; „C“])	[„A“; „D“; „A“; „C“]
hd (tl (tl ([„Peter“; „schläft“; „gern“])))	„gern“
hd (tl (hd ([[2; 3; 4]; [1]]))) :: [ ]	[3]
hd ([„A“; „B“]) :: tl ([„A“; „B“])	[„A“; „B“]
hd ([„A“; „B“]) :: hd (tl ([„A“; „B“])) :: [ ]	[„A“; „B“]

Übersicht:

Vordefinierte strukturierte Datentypen in CAML

Datenstruktur Paar:

Konstruktor:	(_ , _) : ‘a, ‘b → ‘a * ‘b
Selektoren:	fst : ‘a * ‘b → ‘a
	snd : ‘a * ‘b → ‘b

Beispiel: Datentyp string * int:

Objekte:	z. B.: („Willi“, 26), („Anna“, 12)
Operationen:	fst : string * int → string
	snd : string * int → int

Datenstruktur Tupel:

Konstruktor:	(_ , _ , _) bzw. (_ , _ , _ , _) usw.
Selektoren:	es gibt keine vordefinierten Selektoren

Beispiel: Datentyp string * string * int:

Objekte:	(„Olaf“, „Marschall“, 8)
Operationen:	

Datenstruktur Liste:

Konstruktoren:	[ ] : → ‘a list
	:: : ‘a * ‘a list → ‘a list
Selektoren:	hd: ‘a list → ‘a
	tl: ‘a list → ‘a list

Beispiel: Datentyp int list:

Objekte:	[ ], [2; 4; 7]	
Operationen	[ ] : → int list	
	:: : int * int list → int list	Bsp.: 3 **::** [4; 2] → [3; 4; 2]
	hd: int list → int	Bsp.: hd([3; 4; 2]) → 3
	tl: int list → int list	Bsp.: tl([3; 4; 2]) → [4; 2]

4.3 Das Reduktionskonzept

Der Datentyp „Schlange“ soll mit Hilfe des vorgestellten Listenkonzepts implementiert werden. Wir betrachten aber nur den Spezialfall einer Druckerwarteschlange.

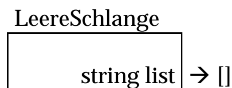
Es stellt sich zunächst die Frage, wie eine Schlange (z. B.: A C D F A D E) mit Hilfe der zur Verfügung stehenden Datenstrukturen dargestellt werden kann. Die einzelnen Objekte einer Warteschlange sollen hier vom Typ „string“ sein. Eine Schlange kann somit als ein Objekt vom Typ „string list“ (Liste aus Zeichenketten) dargestellt werden.

Beispiel: Die Schlange A C D F A D E wird also durch die Liste [„A“; „C“; „D“; „F“; „A“; „D“; „E“] dargestellt.

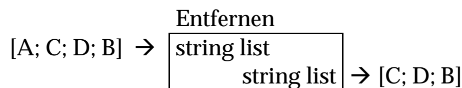
Im Folgenden werden hier im Text bei der Darstellung von Zeichenketten die Anführungszeichen weggelassen. Statt [„A“; „C“; „D“; „F“; „A“; „D“; „E“] wird [A; C; D; F; A; D; E] geschrieben. Man beachte aber, dass CAML eine solch vereinfachende Schreibweise nicht zulässt.

Die Modellierungen der Operationen „LeereSchlange“, „Entfernen“ und „Aufnehmen“ können jetzt wie folgt an das Listenkonzept angepasst werden:

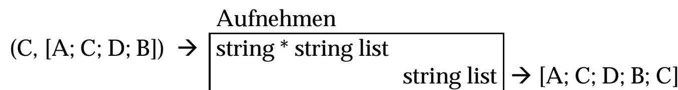
Modellierung:



Modellierung:



Modellierung:



In einem nächsten Schritt sollen diese Operationen / Funktionen definiert werden.

Definition:

LeereSchlange = []

Die Operation „Entfernen“ kann auf zwei verschiedene Weisen definiert werden.

Definition:

```
Entfernen(L) =
  if L = []
  then []
  else tl(L)
```

Definition:

Entfernen([]) → []
Entfernen(e :: r) → r

Worin unterscheiden sich diese Definitionen? Beide benutzen eine Fallunterscheidung: Die Fälle „leere Liste“ und „nichtleere Liste“ werden getrennt behandelt.

Die erste Definition benutzt die if-then-else-Operation, um die Fälle zu unterscheiden. Mit Hilfe dieser Operation gelingt es, die beiden Fälle durch eine einzige Zuordnungsvorschrift zu beschreiben.

Die zweite Definition benutzt zwei unterschiedliche Zuordnungsvorschriften. Der Gültigkeitsbereich der Vorschriften wird mit Hilfe von Listenmustern geregelt. Das Muster [] stellt die leere Liste dar, das Muster e :: r steht für Listen, die aus einem ersten Element e und einer Restliste r bestehen. Die Restliste darf natürlich leer sein. Jede nichtleere Liste lässt sich in der Form e :: r darstellen. Beispiele: [A; C; B] = A :: [C; B] bzw. [A] = A :: [].

Die Fallunterscheidung wird also durch verschiedene Konstruktionsprinzipien erzeugt. Besondere Beachtung verdient der Fall nichtleerer Listen. In der ersten Definition wird hier mit Hilfe eines Selektors, in der zweiten mit Hilfe eines Konstruktors gearbeitet.

Die Verwendung von Mustern zur Erzeugung von Fallunterscheidungen liefert sehr verständliche und gut lesbare Definitionen. Deshalb wird diese Art der Formalisierung heute bevorzugt verwendet.

Es stellt sich hier die Frage, wie ein Muster aufgebaut werden kann. Wir benutzen die folgende Konstruktionsregel: *Muster dürfen nur Variablen und Konstruktoren enthalten.*

Das Bestimmen von Funktionswerten mit Funktionsdefinitionen, die Muster enthalten, erfordert, Gleichungen zu lösen. Will man beispielsweise die Zuordnungsvorschrift Entfernen(e :: r) → r benutzen, um den Funktionswert Entfernen([A; B; C]) zu

bestimmen, so muss man die Gleichung e :: r = [A; B; C] nach e und r auflösen. Im vorliegenden Fall erhält man die Lösung e = A und r = [B; C].

In einem zweiten Schritt soll jetzt die Operation „Aufnehmen“ realisiert werden. Hier ergeben sich Schwierigkeiten: Das Listenkonzept stellt nur eine Operation zum Einfügen an die erste Stelle zur Verfügung, benötigt wird aber eine Operation zum Aufnehmen an der letzten Stelle. Wir stellen eine Lösung vor:

Definition:

Aufnehmen(a, []) → [a]
Aufnehmen(a, e :: r) → e :: Aufnehmen(a, r)

Die erste Zuordnungsvorschrift ist direkt verständlich. Die zweite Zuordnungsvorschrift wird verständlich, wenn man sie als *Reduktionsregel* interpretiert. Eine Reduktionsregel stellt ein Problemreduktionsschema dar. Im vorliegenden Fall wird das Problem „Aufnehmen von a in e :: r“ reduziert auf das Problem „Einfügen von e in das Ergebnis vom Aufnehmen von a in r“. Beide Probleme sind äquivalent. Es handelt sich hier um eine rekursive Definition: Das Ausgangsproblem wird auf ein gleichwertiges Problem zurückgeführt. Dieses enthält als Teilproblem ein dem Ausgangsproblem entsprechendes Problem in verkleinerter Form.

Durch wiederholtes Anwenden von Reduktionsregeln kann man den gewünschten Funktionswert bestimmen.

Die Anwendung auf konkrete Berechnungsausdrücke erzeugt so genannte *Reduktionsschritte*. Beispielsweise erhält man:

Aufnehmen(C, []) → [C]
Aufnehmen(C, [A; C; D; B]) → A :: Aufnehmen(C, [C; D; B])

Die wiederholte Anwendung von Reduktionsregeln führt zu *Reduktionsketten*:

Aufnehmen(C, [A; C; D; B]) → A :: Aufnehmen(C, [C; D; B])
→ A :: (C :: Aufnehmen(C, [D; B]))
→ A :: (C :: (D :: Aufnehmen(C, [B])))
→ A :: (C :: (D :: (B :: Aufnehmen(C, []))))
→ A :: (C :: (D :: (B :: [C])))
→ A :: (C :: (D :: [B; C]))
→ A :: (C :: [D; B; C])
→ A :: [C; D; B; C]
→ [A; C; D; B; C]

In den ersten vier Schritten wurden die zweite Regel (s. o.) benutzt. Der fünfte Schritt erfolgte mit Hilfe der ersten Regel (s. o.). Die folgenden Schritte wurden mit Hilfe vordefinierter Regeln (z. B.: B :: [C] → [B; C]) ausgeführt.

Wir fassen das auf Reduktion beruhende Berechnungskonzept kurz zusammen.

Reduktionskonzept:

- ***Die Definition von Funktionen erfolgt mittels Reduktionsregeln. Diese stellen die Algorithmen im hier dargelegten Programmierkonzept dar.***
- ***Die Berechnung von Funktionswerten erfolgt durch wiederholte Anwendung der Reduktionsregeln. Man erhält hierdurch Reduktionsketten. Das letzte, nicht mehr reduzierbare Glied der Kette stellt den zu berechnenden Funktionswert dar.***

Implementierung des Datentyps „Schlange“:

```
let LeereSchlange = [];;
LeereSchlange : 'a list = []

let Entfernen = function
  [] -> [] |
  e::r -> r;;
Entfernen : 'a list -> 'a list = <fun>

let rec Aufnehmen = function
  (a, []) -> [a] |
  (a, e::r) -> e :: Aufnehmen(a,r);;
Aufnehmen : 'a * 'a list -> 'a list = <fun>
```

Testaufrufe:

```
let S1 = LeereSchlange;;
S1 : 'a list = []

let S2 = Aufnehmen("A",S1);;
S2 : string list = ["A"]

let S3 = Aufnehmen("C",S2);;
S3 : string list = ["A"; "C"]

let S4 = Aufnehmen("A",S3);;
S4 : string list = ["A"; "C"; "A"]

let S5 = Entfernen(S4);;
S5 : string list = ["C"; "A"]

let S6 = Aufnehmen("A",S5);;
S6 : string list = ["C"; "A"; "A"]

let S7 = Aufnehmen("D",S6);;
S7 : string list = ["C"; "A"; "A"; "D"]

let S8 = Entfernen(S7);;
S8 : string list = ["A"; "A"; "D"]
```

Beachte:
- Zum Abtrennen verschiedener Regeln wird ein senkrechter Strich benutzt.
- Bei rekursiven Definitionen wird der Zusatz „rec“ benutzt.

4.4 Das Verfahren der rekursiven Problemreduktion

Für die Verwaltung einer Druckerwarteschlange sind weitere Operationen von Interesse. Man möchte beispielsweise

- bestimmen, wie viele Aufträge in der Warteschlange sind,
- bestimmen, an welcher Stelle sich der erste Auftrag von Benutzer X in der Warteschlange befindet,
- bestimmen, wie viele Aufträge Benutzer X in der Schlange hat,
- den ersten Auftrag von Benutzer X löschen,
- alle Aufträge von Benutzer X löschen,
- alle Benutzer bestimmen, die einen Auftrag in der Warteschlange haben.

Im Folgenden sollen hierzu geeignete Operationen modelliert, definiert und implementiert werden.

Wie viele Aufträge sind in der Warteschlange?

Modellierung:

Länge

[A; B; C; A;] → | string list / int | → 4

Verhaltensbeschreibung: Die Funktion „Länge“ bestimmt, wie viele Aufträge sich in der Schlange befinden.

Es soll eine Definition zur Funktion „Länge“ entwickelt werden. Wir benutzen hierzu das Verfahren der rekursiven Problemreduktion. Die Grundidee dieses Verfahrens ist recht einfach: Versuche, das Problem auf ein sich entsprechendes, aber „verkleinertes“ Problem zu reduzieren. Stelle Problemreduktionsschemata mit Hilfe von Reduktionsregeln dar.

Man findet oft geeignete Reduktionsregeln, indem man typische Reduktionsschritte modelliert und diese dann verallgemeinert. Zunächst versucht man, typische Problemsituationen zu finden. Im vorliegenden Fall ergibt sich etwa:

Problemsituation 1: Länge([]) „leere Liste“
Problemsituation 2: Länge([A; C; D; B]) „nichtleere Liste“

Ein Reduktionsschritt für die 1. Problemsituation lässt sich direkt angeben:

Länge([]) → 0

Die zweite Problemsituation erfordert mehr Überlegungen. Eine Problemreduktion würde sich wie folgt ergeben:

Länge([A; C; D; B]) → Länge([C; D; B])

Dieser Reduktionsschritt ist aber nicht korrekt, wie eine Auswertung der Terme direkt zeigt:

$$\underbrace{\text{Länge}([A;C;D;B])}_{4} \rightarrow \underbrace{\text{Länge}([C;D;B])}_{3}$$

Er wird korrekt, wenn man ihn wie folgt abändert:

$$\underbrace{\text{Länge}([A;C;D;B])}_{4} \rightarrow \underbrace{1 + \underbrace{\text{Länge}([C;D;B])}_{3}}_{4}$$

Es ergeben sich somit die folgenden

Reduktionsschritte:

Länge([]) → 0
Länge([A; C; D; B]) → 1 + Länge([C; D; B])

Diese werden jetzt zu Reduktionsregeln verallgemeinert.

Definition (mit Hilfe von Reduktionsregeln):

Länge([]) → 0
Länge(e :: r) → 1 + Länge(r)

Implementierung:

```
let rec Länge = function
  []   -> 0 |
  e::r -> 1+Länge(r);;
Länge : 'a list -> int = <fun>
```

Testaufrufe:

```
let Schlange = ["A"; "C"; "F"; "C"; "B"];;
Schlange : string list = ["A"; "C"; "F"; "C"; "B"]

Länge(Schlange);;
- : int = 5
```

Die wesentlichen Züge des oben angewandten Verfahrens fassen wir hier noch einmal zusammen:

Verfahren der rekursiven Problemreduktion

Grundidee

Man versucht, *Problemreduktionsschemata* zu entwerfen. Eine Strategie besteht darin, das Problem auf ein sich entsprechendes, aber „verkleinertes“ Problem zu reduzieren. Die Problemreduktionsschemata stellt man mit Hilfe von *Reduktionsregeln* dar. Man findet oft geeignete Reduktionsregeln, indem man typische Reduktionsschritte modelliert und diese dann verallgemeinert.

Reduktionsketten

Hat man solche Problemreduktionsschemata gefunden, so lassen sich diese Schemata wiederholt anwenden, bis keine weitere Reduktion mehr möglich ist. Man erhält hierdurch Reduktionsketten. Das letzte, nicht mehr reduzierbare Glied der Kette stellt den zu berechnenden Funktionswert dar.

Korrektheitsanalyse

Eine Reduktionsregel muss korrekt sein. D. h., der Term auf der linken und der Term auf der rechten Regelseite müssen äquivalent sein. Es ist günstig, sich diese Form der Korrektheit anhand der Reduktionsschritte exemplarisch wie folgt klar zu machen.

$$\underbrace{\text{Aufnehmen}(C,[A;C;D;B])}_{[A;C;D;B;C]} \rightarrow \underbrace{A::\underbrace{\text{Aufnehmen}(C,[C;D;B])}_{[C;D;B;C]}}_{[A;C;D;B;C]}$$

Die Auswertung der Terme findet gemäß der Modellierung statt. Erhält man auf diese Weise nicht dieselben Werte, so ist der modellierte Reduktionsschritt nicht korrekt. Im Folgenden soll unter einer Korrektheitsanalyse stets eine solche exemplarische Termauswertung verstanden werden.

Terminationsanalyse

Die wiederholte Anwendung von Reduktionsschemata sollte nach endlich vielen Schritten zu einer Situation führen, in der das verbleibende Problem direkt gelöst werden kann. Ansonsten erhält man eine nichtterminierende Reduktionskette. Meist ergibt sich die Termination aus der Tatsache, dass in jedem Reduktionschritt die Anzahl der Listenelemente um eins verringert wird. Dies kann nur endlich oft erfolgen.

Dieses Verfahren soll jetzt benutzt werden, um die restlichen Operationen zu realisieren.

An welcher Stelle befindet sich der erste Auftrag von X in der Warteschlange?

Modellierung:

Stelle

(C, [A; B; A; C; H; C]) → | string * (string list) → int | → 4

Verhaltensbeschreibung: Die Funktion „Stelle“ bestimmt, an welcher Stelle sich der erste Auftrag des angegebenen Benutzers befindet.

Reduktionsschritte:

$$\underbrace{\text{Stelle}(C,[\,])}_{0} \rightarrow 0$$

$$\underbrace{\text{Stelle}(C,[A;B;A;C;H;C])}_{4} \rightarrow \underbrace{1+\underbrace{\text{Stelle}(C,[B;A;C;H;C])}_{3}}_{4}$$

$$\underbrace{\text{Stelle}(C,[C;H;C])}_{1} \rightarrow 1$$

Definition(mit Hilfe von Reduktionsregeln):

Stelle(x, []) → 0
Stelle(x, e :: r) → if x = e then 1 else 1 + Stelle(x, r)

Wie viele Aufträge hat X in der Schlange?

Modellierung:

Anzahl

(C, [A; B; A; C; H; C]) → | string * (string list) → int | → 2

Verhaltensbeschreibung: Die Funktion „Anzahl“ bestimmt, wie viele Aufträge des eingegebenen Benutzers sich in der Schlange befinden.

Reduktionsschritte:

$$\underbrace{\text{Anzahl}(C,[\,])}_{0} \rightarrow 0$$

$$\underbrace{\text{Anzahl}(C,[A;B;A;C;H;C])}_{2} \rightarrow \underbrace{\text{Anzahl}(C,[B;A;C;H;C])}_{2}$$

$$\underbrace{\text{Anzahl}(C,[C;H;C])}_{2} \rightarrow \underbrace{1+\underbrace{\text{Anzahl}(C,[H;C])}_{1}}_{2}$$

Definition (mit Hilfe von Reduktionsregeln):

Anzahl(x, [ ]) → 0
Anzahl(x, e :: r) → if x = e then 1 + Anzahl(x, r) else Anzahl(x, r)

Lösche den ersten Auftrag von X.

Modellierung:

	LöscheErstes	
(C, [A; B; A; C; H; C]) →	string * (string list) → string list	→ [A; B; A; H; C]

Verhaltensbeschreibung: Die Funktion „LöscheErstes“ löscht den ersten in der Schlange vorkommenden Auftrag des eingegebenen Benutzers.

Reduktionsschritte:

$$\underbrace{\text{LöscheErstes(C, [ ])}}_{[\,]} \rightarrow [\,]$$

$$\underbrace{\text{LöscheErstes(C, [A; B; A; C; H; C])}}_{\text{[A; B; A; H; C]}} \rightarrow \underbrace{\text{A :: }\underbrace{\text{LöscheErstes(C, [B; A; C; H; C])}}_{\text{[B; A; H; C]}}}_{\text{[A; B; A; H; C]}}$$

$$\underbrace{\text{LöscheErstes(C, [C; H; C])}}_{\text{[H; C]}} \rightarrow \text{[H; C]}$$

Definition (mit Hilfe von Reduktionsregeln):

LöscheErstes(x, [ ]) → [ ]
LöscheErstes(x, e :: r) → if x = e then r else e :: LöscheErstes(x, r)

Lösche alle Aufträge von X.

Modellierung:

	LöscheAlle	
(C, [A; B; A; C; H; C]) →	string * (string list) → string list	→ [A; B; A; H]

Verhaltensbeschreibung: Die Funktion „LöscheAlle“ löscht alle Aufträge des eingegebenen Benutzers.

Reduktionsschritte:

$$\underbrace{\text{LöscheAlle(C, [ ])}}_{[\,]} \rightarrow [\,]$$

$$\underbrace{\text{LöscheAlle(C, [A; B; A; C; H; C])}}_{\text{[A; B; A; H]}} \rightarrow \underbrace{\text{A :: }\underbrace{\text{LöscheAlle(C, [B; A; C; H; C])}}_{\text{[B; A; H]}}}_{\text{[A; B; A; H]}}$$

$$\underbrace{\text{LöscheAlle(C, [C; H; C])}}_{\text{[H]}} \rightarrow \underbrace{\text{LöscheAlle(C, [H; C])}}_{\text{[H]}}$$

Definition (mit Hilfe von Reduktionsregeln):

LöscheAlle(x, [ ]) → [ ]

LöscheAlle(x, e :: r) → if x = e then LöscheAlle(x, r) else e :: LöscheAlle(x, r)

Bestimme alle Benutzer, die einen Auftrag in der Warteschlange haben.

Modellierung:

	Benutzer	
[A; C; D; A; B; B] →	string list → string list	→ [A; C; D; B]

Verhaltensbeschreibung: Die Funktion „Benutzer“ bestimmt alle Benutzer, die einen Auftrag in der eingegebenen Schlange haben.

Reduktionsschritte:

$$\underbrace{\text{Benutzer([ ])}}_{[\,]} \rightarrow [\,]$$

$$\underbrace{\text{Benutzer([A;C;D;A;B;B])}}_{\text{[A;C;D;B]}} \rightarrow \underbrace{\text{A::}\underbrace{\text{Benutzer(}\underbrace{\text{LöscheAlle(A,[C;D;A;B;B])}}_{\text{[C;D;B;B]}}}_{\text{[C;D;B]}}}_{\text{[A;C;D;B]}}$$

Beachte: Die gezeigte Problemreduktion ist „echt verkleinernd“, da die Operation „Benutzer“ auf eine verkürzte Liste angewandt wird.

Definition (mit Hilfe von Reduktionsregeln):

Benutzer([ ]) → [ ]

Benutzer(e :: r) → e :: Benutzer(LöscheAlle(e, r))

Implementierung:

```
let rec Länge = function
  []   -> 0 |
  e::r -> 1+Länge(r);;
Länge : 'a list -> int = <fun>

let rec Stelle = function
  (x, [])   -> 0 |
  (x, e::r) -> if x=e then 1 else 1+Stelle(x,r);;
Stelle : 'a * 'a list -> int = <fun>

let rec Anzahl = function
  (x, [])   -> 0 |
  (x, e::r) -> if x=e then 1+Anzahl(x,r) else Anzahl(x,r);;
Anzahl : 'a * 'a list -> int = <fun>

let rec LöscheErstes = function
  (x, [])   -> [] |
  (x, e::r) -> if x=e then r else e::LöscheErstes(x,r);;
LöscheErstes : 'a * 'a list -> 'a list = <fun>

let rec LöscheAlle = function
  (x, [])   -> [] |
  (x, e::r) -> if x=e then LöscheAlle(x,r) else e::LöscheAlle(x,r);;
LöscheAlle : 'a * 'a list -> 'a list = <fun>

let rec Benutzer = function
  []   -> [] |
  e::r -> e::Benutzer(LöscheAlle(e,r));;
Benutzer : 'a list -> 'a list = <fun>
```

Testaufrufe:

```
let Schlange = ["A"; "C"; "A"; "F"; "C"; "A"; "D"; "F"; "B"];;
Schlange : string list = ["A"; "C"; "A"; "F"; "C"; "A"; "D"; "F"; "B"]

Länge(Schlange);;
- : int = 9

Stelle("F",Schlange);;
- : int = 4

Anzahl("A",Schlange);;
- : int = 3

LöscheErstes("C",Schlange);;
- : string list = ["A"; "A"; "F"; "C"; "A"; "D"; "F"; "B"]

LöscheAlle("C",Schlange);;
- : string list = ["A"; "A"; "F"; "A"; "D"; "F"; "B"]

Benutzer(Schlange);;
- : string list = ["A"; "C"; "F"; "D"; "B"]
```


4.5 Ein komplexeres Programmierproblem

Situationsbeschreibung:

Ein Drucker soll wieder von mehreren Rechnern aus benutzt werden können. Die Druckaufträge sollen jetzt mit Prioritätsangaben versehen werden (z. B.: dringend, kann warten, ...).

Aufgaben:

Es soll zunächst ein allgemeiner (implementationsunabhängiger) Datentyp „PSchlange“ zur Modellierung von Prioritätswarteschlangen entworfen werden. Dieser soll dann in CAML implementiert werden.

Zunächst ist es günstig, die Prioritätsangaben mit Hilfe natürlicher Zahlen zu kodieren, um sie besser vergleichen zu können. Hier soll folgende Vereinbarung gelten:
Je kleiner die Zahl ist, desto höher ist die Priorität.

Datentyp PSchlange

Objekte:

Objekte vom Typ „PSchlange” sind endliche, aber beliebig lange Folgen von Druckaufträgen der Form (Auftraggeber, Prioritätsangabe), wobei die Prioritätsangaben in den Folgenelementen aufsteigend sortiert sind.

Beispiel: (A, 2) (B, 2) (C, 3) (A, 3)

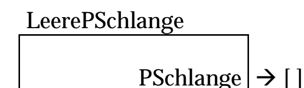
Ein Objekt vom Typ „PSchlange“ kann in CAML mit Hilfe der Datenstrukturen „Tupel“ und „Liste“ dargestellt werden.

Beispiel: [(„A“, 2); („B“, 2); („C“, 3); („A“, 3)]

Wie oben verzichten wir hier im Text auf die Darstellung der Anführungszeichen bei Objekten vom Typ „string“.

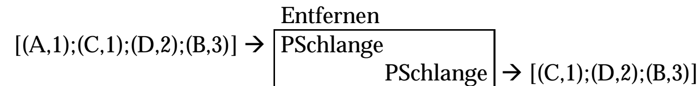
Operationen:

Modellierung:



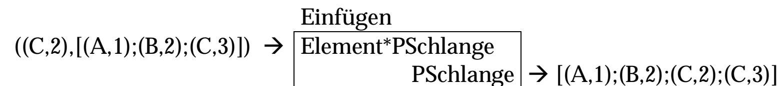
Verhaltensbeschreibung: Die Operation „LeerePSchlange“ erzeugt eine leere Prioritätswarteschlange.

Modellierung:



Verhaltensbeschreibung: Das erste Element der Schlange wird entfernt.

Modellierung:



Verhaltensbeschreibung: Das neue Objekt wird so eingefügt, dass es das letzte Element mit der angegebenen Priorität ist.

Die Definition der Operation „LeerePSchlange“ und der Operation „Entfernen“ erfolgt analog zu den bisher betrachteten Warteschlangen. Neu zu konzipieren ist nur die Operation „Einfügen“.

Reduktionsschritte:

$\underbrace{\text{Einfügen((C,2), [ ])}}_{[(C,2)]} \rightarrow [(C,2)]$

$\underbrace{\text{Einfügen((C,2), [(A,1);(B,2);(C,3)])}}_{[(A,1);(B,2);(C,2);(C,3)]} \rightarrow \underbrace{(A,1) :: \underbrace{\text{Einfügen((C,2), [(B,2);(C,3)])}}_{[(B,2);(C,2);(C,3)]}}_{[(A,1);(B,2),(C,2);(C,3)]}$

$\underbrace{\text{Einfügen((C,2), [(A,3);(B,3);(C,4)])}}_{[(C,2);(A,3);(B,3);(C,4)]} \rightarrow \underbrace{(C,2) :: [(A,3);(B,3);(C,4)]}_{[(C,2);(A,3);(B,3);(C,4)]}$

Definition (mit Hilfe von Reduktionsregeln):

Einfügen((a,i), []) → [(a,i)]

Einfügen((a,i), (b,j)::r) → if i < j then (a,i) :: (b,j) :: r else (b,j) :: Einfügen((a,i),r)

Implementierung:

```
let LeerePSchlange = [];;
LeerePSchlange : 'a list = []

let Entfernen = function
  [] -> [] |
  e::r -> r;;
Entfernen : 'a list -> 'a list = <fun>
```



```
let rec Einfügen = function
  ((a,i), [])       -> [(a,i)] |
  ((a,i), (b,j)::r) -> if i<j
                       then (a,i)::(b,j)::r
                       else (b,j)::Einfügen((a,i),r);;
Einfügen : ('a * 'b) * ('a * 'b) list -> ('a * 'b) list = <fun>
```

Testaufrufe:

```
let S1 = LeerePSchlange;;
S1 : 'a list = []

let S2 = Einfügen(("A",3),S1);;
S2 : (string * int) list = ["A", 3]

let S3 = Einfügen(("C",2),S2);;
S3 : (string * int) list = ["C", 2; "A", 3]

let S4 = Einfügen(("A",2),S3);;
S4 : (string * int) list = ["C", 2; "A", 2; "A", 3]

let S5 = Entfernen(S4);;
S5 : (string * int) list = ["A", 2; "A", 3]

let S6 = Einfügen(("A",1),S5);;
S6 : (string * int) list = ["A", 1; "A", 2; "A", 3]

let S7 = Einfügen(("D",1),S6);;
S7 : (string * int) list = ["A", 1; "D", 1; "A", 2; "A", 3]

let S8 = Entfernen(S7);;
S8 : (string * int) list = ["D", 1; "A", 2; "A", 3]
```

Die eingangs beschriebene Situation soll weiter verallgemeinert werden.

Situationsbeschreibung:

Ein Drucker soll wieder von mehreren Rechnern aus benutzt werden können. Die Druckaufträge sollen jetzt mit Zeit- und Prioritätsangaben versehen werden (z. B.: 7.12 - dringend, 8.13 - kann warten, ...). Der Drucker soll vom „Zeitbetrieb“ (wer zuerst sendet, wird auch zuerst beim Drucken berücksichtigt) auf den „Prioritätsbetrieb“ (hohe Priorität wird bevorzugt behandelt) und auch vom „Prioritätsbetrieb“ auf den „Zeitbetrieb“ umschaltbar sein.

Die zu bearbeitenden Objekte sind jetzt endliche, aber beliebig lange Sequenzen von Druckaufträgen der Form (Auftraggeber, Auftragszeitpunkt, Prioritätsangabe).

Beispiel: (A, 6.12, 2)

In CAML können diese mit Hilfe der Datenstrukturen „Tupel“ und „Liste“ wie folgt dargestellt werden:

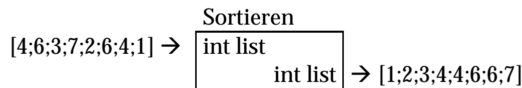
Beispiel: [(A, 6.12, 2); (B, 10.45, 2); (C, 11.11, 3); (A, 12.00, 1)]

Ein Warteschlangenobjekt ist entweder nach Zeitpunkten oder nach Prioritätsangaben geordnet. Das Umschalten zwischen den beiden Betrieben erfordert einen Sortiervorgang. Dieser soll im Folgenden genauer betrachtet werden.

Wir betrachten zunächst eine vereinfachte Situation:

Es soll eine Liste bestehend aus Zahlen aufsteigend sortiert werden.

Modellierung:



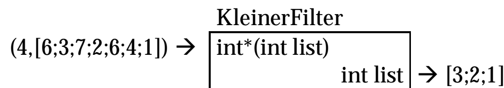
Verhaltensbeschreibung: Die eingegebene Liste soll aufsteigend sortiert werden.

Es gibt eine Vielzahl von Sortierverfahren. Wir betrachten hier nur das Verfahren **Quicksort**. Die Grundidee von Quicksort besteht darin, ein so genanntes Pivotelement zu bestimmen und die Liste gemäß dieses Pivotelement zu zerlegen in

- die Elemente der Liste, die kleiner als das Pivotelement sind,
- das Pivotelement selbst und
- die Elemente der Liste außer dem Pivotelement, die größergleich dem Pivotelement sind.

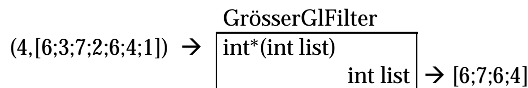
Wir modellieren zunächst zwei Operationen, um die jeweiligen Elemente aus der Ausgangsliste herauszufiltern.

Modellierung:



Verhaltensbeschreibung: Die Funktion „KleinerFilter“ bestimmt alle Elemente der eingegebenen Liste, die kleiner als das eingegebene Pivotelement sind.

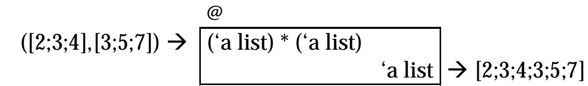
Modellierung:



Verhaltensbeschreibung: Die Funktion „GrößerGlFilter“ bestimmt alle Elemente der eingegebenen Liste, die größergleich als das eingegebene Pivotelement sind.

Die folgende vordefinierte Hilfsoperation wird benötigt, um Listen aneinander zu hängen.

Modellierung:



Verhaltensbeschreibung: Die vordefinierte Funktion @ hängt Listen beliebigen Typs aneinander.

Implementierung:

```
let rec KleinerFilter = function
  (x,[])   -> [] |
  (x,e::r) -> if e < x
              then e::KleinerFilter(x,r)
              else KleinerFilter(x,r);;
KleinerFilter : 'a * 'a list -> 'a list = <fun>

let rec GrösserGlFilter = function
  (x,[])   -> [] |
  (x,e::r) -> if e >= x
              then e::GrösserGlFilter(x,r)
              else GrösserGlFilter(x,r);;
GrösserGlFilter : 'a * 'a list -> 'a list = <fun>

let rec QuickSort = function
  []   -> [] |
  e::r -> QuickSort(KleinerFilter(e,r))
          @ [e] @
          QuickSort(GrösserGlFilter(e,r));;
QuickSort : 'a list -> 'a list = <fun>
```

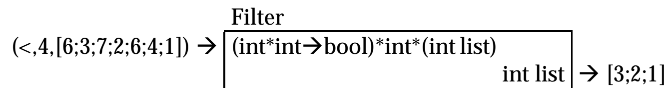
Testaufrufe:

```
let L = [6;5;7;4;1;10;5;6;3;8;2];;
L : int list = [6; 5; 7; 4; 1; 10; 5; 6; 3; 8; 2]

Kleiner(6,tl(L));;
- : int list = [5; 4; 1; 5; 3; 2]

GrösserGleich(6,tl(L));;
- : int list = [7; 10; 6; 8]

QuickSort(L);;
- : int list = [1; 2; 3; 4; 5; 5; 6; 6; 7; 8; 10]
```

Bei der Realisierung der Funktionen „KleinerFilter“ und „GrösserGlFilter“ fällt auf, dass diese Operationen auf sehr ähnliche Weise definiert sind. Der einzige Unterschied besteht darin, dass einmal die „Kleiner”-Relation, das andere mal die „GrößerGleich”-Relation zum Herausfiltern von Elementen aus der Liste benutzt wird.

Wir modellieren jetzt eine allgemeine Filteroperation, die für eine beliebig eingegebene Vergleichsoperation p und ein beliebig eingegebenes Vergleichselement x das Herausfiltern vornimmt.

Modellierung:



Verhaltensbeschreibung: Die Funktion „Filter“ bestimmt alle Elemente der eingegebenen Liste, die in der Relation Kleiner” zu dem eingegebene Pivotelement stehen.

Beachte: Hier wird als Eingabeobjekt u. a. ein Funktionsobjekt benutzt: Die einzugebende Relation ist eine Funktion vom Typ (int*int)→bool.

Implementierung:

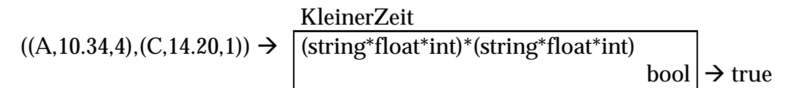
```
let rec Filter = function
  (p,x,[])   -> [] |
  (p,x,e::r) -> if p(e,x)
                then e::Filter(p,x,r)
                else Filter(p,x,r);;
Filter : ('a * 'b -> bool) * 'b * 'a list -> 'a list = <fun>

let Kleiner = function
  (a,b) -> (a < b);;
Kleiner : 'a * 'a -> bool = <fun>

let GrösserGl = function
  (a,b) -> (a >= b);;
GrösserGl : 'a * 'a -> bool = <fun>

let rec QuickSort = function
  []   -> [] |
  e::r -> QuickSort(Filter(Kleiner,e,r))
          @ [e] @
          QuickSort(Filter(GrösserGl,e,r));;
QuickSort : 'a list -> 'a list = <fun>
```

Testaufrufe:

```
let L = [6;5;7;4;1;10;5;6;3;8;2];;
L : int list = [6; 5; 7; 4; 1; 10; 5; 6; 3; 8; 2]

Filter(Kleiner,6,tl(L));;
- : int list = [5; 4; 1; 5; 3; 2]

Filter(GrösserGl,6,tl(L));;
- : int list = [7; 10; 6; 8]

QuickSort(L);;
- : int list = [1; 2; 3; 4; 5; 5; 6; 6; 7; 8; 10]
```

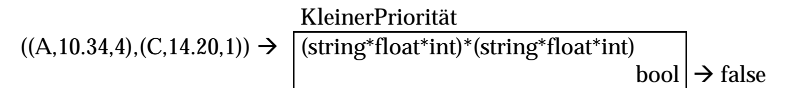

Wir kommen jetzt zum eingangs auf Seite 75 diskutierten Problem zurück. Die Schwierigkeit besteht jetzt darin, geeignete Vergleichsoperationen für Zeit- und Prioritätsangaben zu definieren.

Modellierung:



Verhaltensbeschreibung: Die Funktion „KleinerZeit“ vergleicht Druckaufträge hinsichtlich des Zeitpunktes. Bei gleichem Zeitpunkt (realistisch?) entscheidet die Priorität.

Modellierung:



Verhaltensbeschreibung: Die Funktion „KleinerPriorität“ vergleicht Druckaufträge hinsichtlich der Priorität. Bei gleicher Priorität entscheidet der Zeitpunkt.

Analog werden die Operationen „GrösserGlZeit“ und „GrösserGlPriorität“ modelliert. Mit Hilfe der oben erstellten Quicksort-Funktion kann das Umschalten jetzt direkt realisiert werden.

Implementierung:

```
let rec Filter = function
  (p,x,[])   -> [] |
  (p,x,e::r) -> if p(e,x)
                then e::Filter(p,x,r)
                else Filter(p,x,r);;
Filter : ('a * 'b -> bool) * 'b * 'a list -> 'a list = <fun>

let KleinerZeit = function
  ((n,t,p),(n',t',p')) -> ((t <. t') or ((t = t') & (p < p')));;
KleinerZeit : ('a * float * 'b) * ('c * float * 'b) -> bool = <fun>

let GrösserGlZeit = function
  ((n,t,p),(n',t',p')) -> ((t >. t') or ((t = t') & (p >= p')));;
GrösserGlZeit : ('a * float * 'b) * ('c * float * 'b) -> bool = <fun>

let KleinerPriorität = function
  ((n,t,p),(n',t',p')) -> ((p < p') or ((p = p') & (t <. t')));;
KleinerPriorität : ('a * float * 'b) * ('c * float * 'b) -> bool = <fun>

let GrösserGlPriorität = function
  ((n,t,p),(n',t',p')) -> ((p > p') or ((p = p') & (t >=. t')));;
GrösserGlPriorität : ('a * float * 'b) * ('c * float * 'b) -> bool = <fun>

let rec QuickSortZeit = function
  []   -> [] |
  e::r -> QuickSortZeit(Filter(KleinerZeit,e,r))
```

```
          @ [e] @
          QuickSortZeit(Filter(GrösserGlZeit,e,r));;
QuickSortZeit : ('a * float * 'b) list -> ('a * float * 'b) list = <fun>

let rec QuickSortPriorität = function
  []   -> [] |
  e::r ->  QuickSortPriorität(Filter(KleinerPriorität,e,r))
          @ [e] @
          QuickSortPriorität(Filter(GrösserGlPriorität,e,r));;
QuickSortPriorität : ('a * float * 'b) list -> ('a * float * 'b) list = <fun>
```

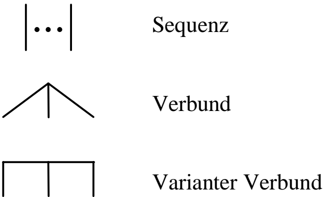
Testaufrufe:

```
let S1 =
 [
  ("A", 6.10,3);
  ("B", 7.12,2);
  ("A", 7.18,2);
  ("C", 7.18,1);
  ("B", 9.00,4);
  ("B",10.05,3);
  ("A",11.11,1);
  ("B",12.00,2)
 ];;
S1 : (string * float * int) list =
 ["A", 6.1, 3; "B", 7.12, 2; "A", 7.18, 2; "C", 7.18, 1; "B", 9.0, 4;
  "B", 10.05, 3; "A", 11.11, 1; "B", 12.0, 2]

let S2 = QuickSortPriorität(S1);;
S2 : (string * float * int) list =
 ["C", 7.18, 1; "A", 11.11, 1; "B", 7.12, 2; "A", 7.18, 2; "B", 12.0, 2;
  "A", 6.1, 3; "B", 10.05, 3; "B", 9.0, 4]

let S3 = QuickSortZeit(S2);;
S3 : (string * float * int) list =
 ["A", 6.1, 3; "B", 7.12, 2; "C", 7.18, 1; "A", 7.18, 2; "B", 9.0, 4;
  "B", 10.05, 3; "A", 11.11, 1; "B", 12.0, 2]
```


5 Mit Funktionen kann man komplexe Software-produkte entwerfen

Inhalte:

- Ein Interpreter für eine Pascal-artige Programmiersprache.
- Imperative und deklarative Programmierung: Bei der imperativen Programmierung wird beschrieben, wie berechnet werden soll. Bei der deklarativen Programmierung wird beschrieben, was berechnet werden soll.
- Prototyping: Bei der Entwicklung komplexer Systeme werden frühzeitig ablauffähige Modelle (sog. „Prototypen“) erstellt, um mit diesen experimentieren zu können.
- Komplexe Datenobjekte werden mit Hilfe von Datenstrukturen konstruiert. Hier werden insbesondere die Datenstrukturen Sequenz, Verbund und varianter Verbund benutzt. Zur Veranschaulichung dieser Datenstruktur verwenden wir hierbei die folgende Notation (vgl. [Hubwieser & Broy 95]):



Bemerkungen:

Programmiersprachen werden hier auf zwei Ebenen thematisiert. Mit Hilfe einer funktionalen Sprache (CAML) wird ein Interpreter für eine imperative (Pascal-artige) Sprache entworfen. Ein Grundverständnis der Grundkonzepte imperativer Programmierung wird dabei vorausgesetzt. Die im Folgenden aufgeführten Überlegungen können auch dazu benutzt werden, die Grundkonzepte imperativer Programmierung zu vertiefen. Es bietet sich an, im Anschluss an das Erstellen des Interpreters einen Vergleich zwischen imperativer und funktionaler Programmierung vorzunehmen.

5.1 Das Softwareprodukt

Ziel: Ein Mini-Interpreter

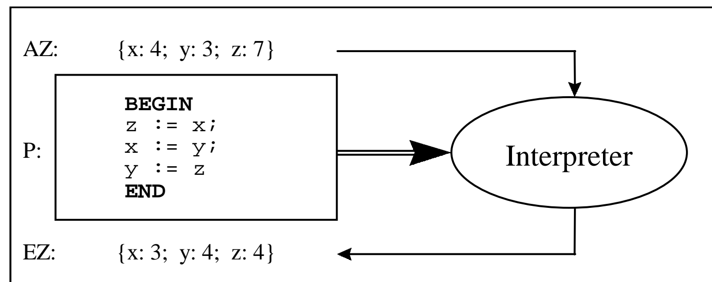
Es soll ein Interpreter für eine neue, Pascal-artige Programmiersprache entwickelt werden.

Zunächst ist zu klären, was man unter einem Interpreter versteht. Ein *Interpreter* ist ein Programm, das Programme einer bestimmten Programmiersprache ausführt bzw. auswertet. Dieser Sachverhalt soll zunächst anhand eines konkreten Beispiels entwickelt werden. Der zu konzipierende Interpreter soll vereinfachte Pascal-Programme wie etwa das Folgende ausführen können.

```
BEGIN
z := x;
x := y;
y := z
END
```

Die Vereinfachung besteht hier u. a. darin, dass auf Deklarationen verzichtet wird.

Was wird hier unter „ausführen" verstanden? Das oben wiedergegebene Programm dient dazu, die Werte der Variablen x und y auszutauschen. Die Werte dieser Variablen sollen im Folgenden durch sog. Variablenzustände erfasst werden. Die Schreibweise {x: 4; y: 3; z: 7} soll z. B. einen Zustand beschreiben, bei dem die Variable x den Wert 4, die Variable y den Wert 3 und die Variable z den Wert 7 hat. Ein Interpreter, der das obige Programm ausführen kann, weist somit das folgende Verhalten auf:



Der Interpreter erhält ein Programm P und einen Variablenzustand AZ (Ausgangszustand) als Eingabe. Die Auswertung dieser Daten liefert einen neuen Variablenzustand EZ (Endzustand), der sich durch Abarbeiten des Programms, ausgehend vom ursprünglichen Variablenzustand, ergibt.

Dabei ergeben sich die folgenden Aufgaben:

Aufgaben:

- Die Sprachelemente der vereinfachten Programmiersprache müssen festgelegt werden.
- Das Ausführungsprogramm, das die Programme der vereinfachten Programmiersprache auswertet, muss entwickelt werden.

Die Sprachelemente:

Die Sprache soll Wertzuweisungen ermöglichen und die wesentlichen Kontrollstrukturen (Sequenz, Fallunterscheidung, Wiederholung) zur Verfügung stellen. Folgende „Programme" soll die Sprache z. B. zulassen:

Programm 1:

```
BEGIN
z := x;
x := y;
y := z
END
```

Programm 2:

```
BEGIN
p := 1;
WHILE u > 0 DO
      BEGIN
      IF u mod 2 = 1 THEN
            BEGIN
            u := u - 1;
            p := p * b
            END;
      u := u div 2;
      b := b * b
      END
END
```

Verzichtet wird auf eine Vielfalt von Datentypen. Der einzige Typ, den die Sprache zulässt, soll der Datentyp „integer" sein.

Verzichtet wird des Weiteren auf ein Prozedurkonzept.

Verzichtet wird schließlich auch auf Ein- und Ausgabeanweisungen. Wir gehen davon aus, dass dem Interpreter ein Ausgangszustand (wie in der Skizze oben) übergeben werden kann und dass der Interpreter den berechneten Endzustand als Ergebnis seiner Berechnung ausgibt. Ein- und Ausgaben können so zwar nicht interaktiv in ein Programm integriert werden. Eine einmalige Eingabe (zu Beginn der Abarbeitung) und eine einmalige Ausgabe (nach dem Ende der Abarbeitung) können aber simuliert werden.

Zum Ausführungsprogramm:

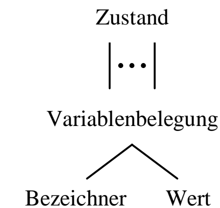
Die Aufgabe, einen Interpreter für Programme der gezeigten Art zu erstellen, ist recht komplex. Um mit dieser Aufgabe zurecht zu kommen, werden zunächst nur sehr einfache Programme betrachtet. In weiteren Schritten werden die Einschränkungen nach und nach aufgehoben. Begonnen wird mit Sequenzen aus Wertzuweisungen der Gestalt „Variablenbezeichner1 := Variablenbezeichner2“.

5.2 Das Speicherkonzept – Variablenzustände

Wir entwickeln hier einen Datentyp „Variablenzustand“ (oder kurz: „Zustand”), mit dessen Hilfe die momentanen Werte der interessierenden Variablen erfasst werden sollen.

Objekte:

Ein *Variablenzustand* ist eine Sequenz von Variablenbelegungen, jeweils bestehend aus einen Bezeichner und dem zugehörigen Wert. Beispiel: {x: 4; y: 3; z: 7}. Die Struktur der Objekte kann wie folgt veranschaulicht werden:



Operationen:

Die Abarbeitung eines solchen, aus primitiven Wertzuweisungen bestehenden Programms erfordert Operationen zur Bearbeitung von Variablenzuständen. Diese sollen jetzt modelliert werden. Wir betrachten zunächst die Abarbeitung des oben gezeigten Beispielprogramms.

```
{x: 4;  y: 3;  z: 7}
z := x
{x: 4;  y: 3;  z: 4}
x := y
{x: 3;  y: 3;  z: 4}
y := z
{x: 3;  y: 4;  z: 4}
```

Es stellt sich das Problem, welche Operationen zur Bearbeitung von Variablenzuständen benötigt werden, um einen Auswertungsschritt wie den folgenden zu ermöglichen.

```
{x: 3;  y: 3;  z: 4}
y := z
{x: 3;  y: 4;  z: 4}
```

Zunächst muss die rechte Seite der Wertzuweisung ausgewertet werden. Hierzu muss der Wert der entsprechenden Variablen ermittelt werden. Als Nächstes muss der Variablen auf der linken Seite der Zuweisung der ermittelte Wert zugeordnet werden. Es muss also ein neuer Variablenzustand gebildet werden

Modellierung:

VariablenWert

(y, {x: 3; y: 3; z: 4}) → | Bezeichner * Zustand → Wert | → 3

Verhaltensbeschreibung: Bei Eingabe eines Variablenbezeichners und eines Zustands wird der Wert der Variablen ausgegeben.

Modellierung:

NeuerZustand

((y, 6), {x: 3; y: 3; z: 4}) → | (Bezeichner * Wert) * Zustand → Zustand | → {x: 3; y: 6; z: 4}

Verhaltensbeschreibung: Bei Eingabe eines Bezeichner-Wert-Paares und eines Zustandes wird der Zustand ausgegeben, der entsteht, wenn dem eingegebenen Bezeichner der eingegebene Wert neu zugeordnet wird. Der alte Wert geht dabei verloren.

Wir benutzen in unserer konzeptionellen Darstellung zwei weitere Datentypen „Bezeichner“ und „Wert“. Diese werden unten genauer spezifiziert.

Der nächste Schritt besteht darin, die modellierten Objekte und Operationen zu implementieren.

Hierzu werden die modellierten Objekte mit Hilfe der vordefinierten Typen und Datenstrukturen von CAML beschrieben. Ferner werden Regeln zur Berechnung der modellierten Operationen entwickelt.

Objekte:

Eine Zustandsbeschreibungen der Gestalt {x: 3; y: 3; z: 4} lässt sich mit Hilfe der vordefinierten Typen und Datenstrukturen wie folgt darstellen: [(„x“, 3); („y“, 3); („z“, 4)]. Der Typ „Zustand” kann also in der Form „(string * int) list“ implementiert werden. Die Einschränkung, dass alle Elemente einer Liste den gleichen Typ haben müssen, hat zur Folge, dass den Variablenbezeichnern nur ganze Zahlen als Werte zugeordnet werden können. Diese Einschränkung soll zunächst hingenommen werden.

Operationen:

Definition / Reduktionsregeln:

VariablenWert(bez, []) → ?

VariablenWert(bez, (bez’, wert’) :: restZustand) →

if bez = bez’
then wert’
else VariablenWert(bez,restZustand)

Definition / Reduktionsregeln:

NeuerZustand((bez, wert), [ ]) → [(bez, wert)]

NeuerZustand((bez, wert), (bez’, wert’) :: restZustand) →
if bez = bez’
then (bez, wert) :: restZustand
else (bez’, wert’) :: NeuerZustand (bez, wert, restZustand)

Korrektheitsüberlegungen:

$$\underbrace{\text{VariablenWert("y",[("y";2),("z",4)])}}_{2} \rightarrow 2$$

$$\underbrace{\text{VariablenWert("y",[("x",3);("y";2),("z",4)])}}_{2} \rightarrow \underbrace{\text{VariablenWert("y",[("y";2),("z",4)])}}_{2}$$

$$\underbrace{\text{NeuerZustand(("y",5),[("y";2);...])}}_{\text{[("y";5);...]}} \rightarrow \underbrace{\text{[("y",5)]::[...]}}_{\text{[("y",5);...]}}$$

$$\underbrace{\text{NeuerZustand(("y",5),[("x",3);("y";2);...])}}_{\text{[("x",3);("y";5);...]}} \rightarrow \underbrace{\text{("x",3)::}\underbrace{\text{NeuerZustand(("y",5),[("y";2);...])}}_{\text{[("y",5);...)}}}_{\text{[("x",3);("y";5);...]}}$$

Implementierung:

```
let rec VariablenWert = function
   (bez, [ ] )
      -> raise (Failure "Variable nicht vorhanden") |
   (bez, (bez', wert') :: restZustand)
      -> if bez = bez'
              then wert'
              else VariablenWert (bez, restZustand);;

VariablenWert : 'a * ('a * 'b) list -> 'b = <fun>
```

```
let rec NeuerZustand = function
   ((bez, wert), [ ])
      -> (bez, wert) :: [ ] |
   ((bez, wert), (bez', wert') :: restZustand)
      -> if bez = bez'
            then (bez, wert) :: restZustand
            else (bez', wert') :: NeuerZustand((bez, wert),
                                                restZustand);;

NeuerZustand : ('a * 'b) * ('a * 'b) list -> ('a * 'b) list = <fun>
```

Test der Speicher-Operationen:

```
let VarZustand = [ ("x", 3); ("y", 3); ("z", 4)];;
VarZustand : (string * int) list = ["x", 3; "y", 3; "z", 4]

VariablenWert ("y", VarZustand);;
- : int = 3

NeuerZustand(("y", 4), VarZustand);;
- : (string * int) list = ["x", 3; "y", 4; "z", 4]
```

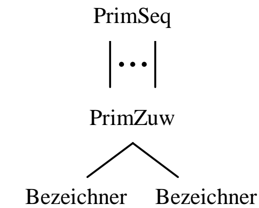

5.3 Ein Interpreter für primitive Programme

Der Interpreter soll zunächst nur Sequenzen aus Wertzuweisungen der Gestalt „Variablenbezeichner1 := Variablenbezeichner2“ ausführen können. Wir nennen Wertzuweisungen dieser einfachen Gestalt *primitive Wertzuweisungen* und Sequenzen aus primitiven Wertzuweisungen *primitive Programme*.

Zur Beschreibung dieser Objekte entwickeln wir die Datentypen „PrimZuw(eisung)“ und „PrimSeq(uenz)“.

Objekte:

Ein Objekt vom Typ „PrimZuw“ ist eine Zuweisung der Gestalt y := z. Einem Variablenbezeichner wird der Wert eines (meist anderen) Variablenbezeichners zugewiesen. Man beachte, dass rechts vom Zuweisungszeichen nur Variablenbezeichner stehen dürfen - hieraus resultiert der Zusatz „primitiv“. Ein Objekt vom Typ „PrimSeq“ ist eine Sequenz der Gestalt BEGIN z := x; x := y; y := z END bestehend aus primitiven Zuweisungen. Die Struktur dieser Objekte lässt sich wie folgt veranschaulichen:

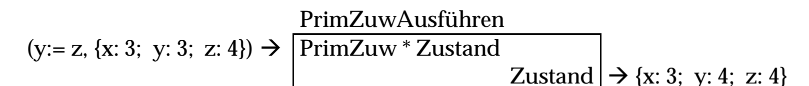


Eine primitive Zuweisung der Gestalt y := z lässt sich mit Hilfe der vordefinierten Typen und Datenstrukturen wie folgt darstellen: („y“, „z“). Man erhält Objekte vom Typ „string*string“.

Ein primitives Programm der Gestalt BEGIN z := x; x := y; y := z END lässt sich wie folgt darstellen: [(„z“, „x“); („x“, „y“); („y“, „z“)]. Man erhält Objekte von Typ „(string*string) list“.

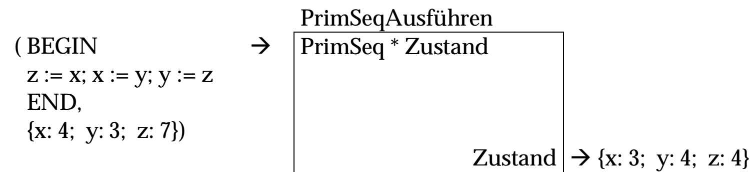
Operationen:

Modellierung:



Verhaltensbeschreibung: Bei Eingabe einer primitiven Zuweisung und eines Zustandes liefert die Funktion „PrimZuwAusführen“ den neuen Zustand, der sich durch Auswertung der eingegebenen primitiven Zuweisung ergibt.

Modellierung:



Verhaltensbeschreibung: Bei Eingabe einer primitiven Sequenz und eines Zustandes liefert die Funktion „PrimSeqAusführen“ den neuen Zustand, der sich durch Auswertung des eingegebenen primitiven Programms ergibt.

Definition / Reduktionsregeln:

PrimZuwAusführen((bez1, bez2), zustand) →
NeuerZustand((bez1, VariablenWert(bez2, zustand)), zustand)

Definition / Reduktionsregeln:

PrimSeqAusführen([], zustand) → zustand

PrimSeqAusführen(zuw :: restZuweisungen, zustand) →
PrimSeqAusführen(restZuweisungen, PimZuwAusführen(zuw, zustand))

Korrektheitsüberlegungen:

$\underbrace{\text{PrimZuwAusführen}(y := z, \{x{:}3; y{:}4; z{:}7\})}_{\{x:3;y:7;z:7\}} \rightarrow$

$\underbrace{\text{NeuerZustand}((y, \underbrace{\text{VariablenWert}(z, \{x{:}3; y{:}4; z{:}7\})}_{7}), \{x{:}3; y{:}4; z{:}7\})}_{\{x:3;y:7;z:7\}}$

$\underbrace{\text{PrimSeqAusführen}([z := x; x := y; y := z], \{x{:}4; y{:}3; z{:}7\})}_{\{x:3;\ y:4;\ z:4\}} \rightarrow$

$\underbrace{\text{PrimSeqAusführen}([x := y; y := z], \underbrace{\text{PrimZuwAusführen}(z := x, \{x{:}3; y{:}4; z{:}7\})}_{\{x:4;\ y:3;\ z:4\}})}_{\{x:3;\ y:4;\ z:4\}}$

Implementierung:

```
(* Interpreter für primitive Programme *)

let PrimZuwAusführen = function
  ((bez1, bez2), zustand) ->
    NeuerZustand((bez1, VariablenWert(bez2, zustand)), zustand);;
PrimZuwAusführen : ('a * 'a) * ('a * 'b) list -> ('a * 'b) list = <fun>

let rec PrimSeqAusführen = function
  ([], zustand) -> zustand |
  (zuw::restZuweisungen, zustand) ->
   PrimSeqAusführen(restZuweisungen, PrimZuwAusführen(zuw,zustand));;
PrimSeqAusführen : ('a * 'a) list * ('a * 'b) list ->
  ('a * 'b) list = <fun>
```

Testaufrufe:

```
let VarZustand = [ ("x", 3); ("y", 7); ("z", 4) ];;
VarZustand : (string * int) list = ["x", 3; "y", 7; "z", 4]

let PrimProgr  = [ ("z", "x"); ("x", "y"); ("y", "z") ];;
PrimProgr : (string * string) list = ["z", "x"; "x", "y"; "y", "z"]

PrimSeqAusführen(PrimProgr, VarZustand);;
- : (string * int) list = ["x", 7; "y", 3; "z", 3]
```

In der folgenden Übersicht fassen wir die bisher eingeführten Objekttypen noch einmal zusammen.

Übersicht:

konzeptionell		**Implementierung in CAML**	
Objekt	*Typ*	*Objekt*	*Typ*
x	Bezeichner	„x“	string
3	Wert	3	int
{x: 4; y: 3; z: 7}	Zustand	[(„x“,4); („y“,3); („z“, 7)]	(string*int) list
z := x	PrimZuw	(„z“, „x“)	string*string

BEGIN
z := x;
x := y;
y := z
END
	PrimSeq	[
		(„z“, „x“);
		(„x“, „y“);
		(„y“, „z“)
		]
			(string*string) list

5.4 Terme

Die Sprache soll um die Möglichkeit erweitert werden, dass komplexe Terme auf der rechten Seite in Zuweisungen stehen können. Ein erster Implementierungsversuch der neu zu betrachtenden Objekte zeigt die Schwierigkeiten auf.

Programm	Implementierung	Typ
BEGIN	[	
x := 3;	(„x“, 3);	string * int
y := x;	(„y“, „x“);	string * string
y := y + 1	(„y“, („y“, „+“, 1))	string * (string * string * int)
END	]	

Die Implementierung ist nicht korrekt. Die einzelnen Zuweisungen sind von unterschiedlichem Typ. Das eingeführte Listenkonzept erlaubt es nicht, diese Zuweisungen in einer Liste auf die oben gezeigte Weise zusammenzufassen.

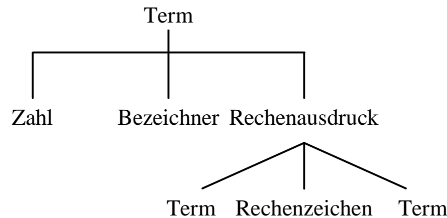
Bei der Beseitigung der entstandenen Schwierigkeit hilft die folgende Idee: Es wird ein neuer, einheitlicher Typ für Termobjekte eingeführt. Zunächst muss hierzu geklärt werden, was ein Term ist. Beispiele für Terme sind: 4, x, x+y, 3*(x div 2), ... Eine Analyse zeigt die folgende Struktur auf:

Objekte:

Ein Objekt vom Typ *Term* ist entweder

- eine Zahl (z. B.: 3) oder
- ein Variablenbezeichner (z. B.: x) oder
- ein Rechenausdruck (z. B.: y + 1) bestehend aus einem Term (hier: y), einem Rechenzeichen (hier: +) und einem Term (hier: 1).

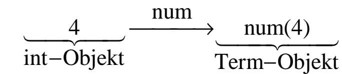
Bei der Struktur handelt es sich um einen varianten Verbund, der rekursiv aufgebaut ist:



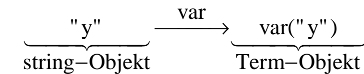
Zu klären bleibt, wie man in CAML einen neuen Typ konstruiert, mit dem man solch komplexe Term-Objekte beschreiben kann.

Man definiert hierzu ***Konstruktoren***, die aus bereits existierenden Objekten neue Term-Objekte machen.

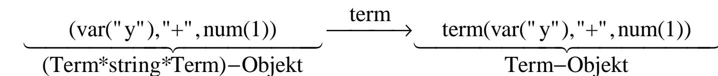
Der Konstruktor *num* macht aus einem int-Objekt ein Term-Objekt:



Der Konstruktor *var* macht aus einem string-Objekt ein Term-Objekt:



Der Konstruktor *term* macht aus einem einem string-Objekt (Rechenzeichen) und zwei Term-Objekten ein neues Term-Objekt:



In CAML lässt sich dies wie folgt realisieren:

```
type Term =
   num of int |
   var of string |
   term of Term * string * Term ;;
```

Man beachte, dass CAML Groß- und Kleinschreibung unterscheidet. „Term“ ist der Name des neu definierten Typs, „term“ ist der Name des neu eingeführten Konstruktors.

Zur Illustration übersetzen wir einige Terme in CAML-Objekte.

Term (konzeptionell)	Term-Objekt in CAML
3-x	`term(num(3),"-",var("x"))`
15	`num(15)`
x+y	`term(var("x"),"+",var("y"))`
3-(x*z)	`term(num(3), "-",term(var("x"),"*",var("z")))`

(x+4) div (y+2)	`term(`
	`term(var("x"), "+",num(4)),`
	`"div",`
	`term(var("y"), "+",num(2))`
	`)`

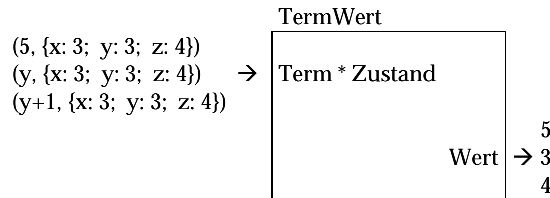
Die Einführung des neuen Datentyps „Term“ löst die eingangs skizzierte Schwierigkeit.

Programm	Implementierung	Typ
BEGIN	[	
x := 3;	(„x“, 3);	string * Term
y := x;	(„y“, „x“);	string * Term
y := y + 1	(„y“, („y“, „+“, 1))	string * Term
END	]	

Zur Auswertung von Termen wird eine neue Operation eingeführt.

Operationen:

Modellierung:



Verhaltensbeschreibung: Bei Eingabe eines Terms und eines Zustandes wird der Wert des Terms bezüglich dieses Zustands ausgegeben.

Definition / Reduktionsregeln:

TermWert(num(wert), zustand) → wert

TermWert(var(bez), zustand) → VariablenWert(bez, zustand)

TermWert(term(term1, "+", term2), zustand) →
TermWert(term1, zustand) + TermWert(term2, zustand)

...

Korrektheitsüberlegung:

$$\underbrace{\text{TermWert(num(3),[("x",3);("y";2),("z",4)])}}_{3} \to 3$$

$$\underbrace{\text{TermWert(var("y"),[("x",3);("y";2),("z",4)])}}_{2} \to \underbrace{\text{VariablenWert("y",[("y";2),("z",4)])}}_{2}$$

$$\underbrace{\text{TermWert(term(var("y"),"+",var("x")),[("x",3);("y";2)])}}_{5} \to$$

$$\underbrace{\underbrace{\text{TermWert(var("y"),[("x",3);("y";2)])}}_{2} + \underbrace{\text{TermWert(var("x")),[("x",3);("y";2)])}}_{3}}_{5}$$

Man beachte die unterschiedliche Verwendung des Pluszeichens. Die Zeichenkette „+“ auf der linken Regelseite ist Bestandteil der vom Interpreter auszuführenden Sprache. Das +-Symbol auf der rechten Regelseite stellt die CAML-Operation dar, die die Auswertung der auszuführenden Addition tatsächlich übernimmt. Die Zeichenkette „+“ gehört demnach zur Objektsprache, das Rechenzeichen + zur Metasprache CAML, mit der der Interpreter implementiert wird.

Implementierung:

```
(* Definition eines neuen Typs für Termobjekte *)

type Term =
   num of int |
   var of string |
   term of Term * string * Term;;
Type Term defined.

(* Definition der Operation TermWert - dient der Auswertung
   arithmetischer Terme *)

let rec TermWert = function
   (num(wert), zustand)
      -> wert |
   (var(bez), zustand)
      -> VariablenWert(bez, zustand) |
   (term(term1, "+", term2), zustand)
      -> TermWert(term1, zustand) + TermWert(term2, zustand) |
   (term(term1, "-", term2), zustand)
      -> TermWert(term1, zustand) - TermWert(term2, zustand) |
   (term(term1, "*", term2), zustand)
      -> TermWert(term1, zustand) * TermWert(term2, zustand) |
   (term(term1, "div", term2), zustand)
      -> TermWert(term1, zustand) / TermWert(term2, zustand) |
   (term(term1, "mod", term2), zustand)
      -> TermWert(term1, zustand) mod TermWert(term2, zustand) |
   (_, zustand)
      -> raise(Failure "kein korrekter Term");;
TermWert : Term * (string * int) list -> int = <fun>

(* Beispiel eines Termobjektes - entspricht y+(4*2) *)

let HilfsTerm = term(var("y"), "+", term(num(4), "div", num(2)));;
HilfsTerm : Term = term (var "y", "+", term (num 4, "div", num 2))
```

Test der Operation TermWert :

```
TermWert(HilfsTerm, VarZustand);;
- : int = 9
```

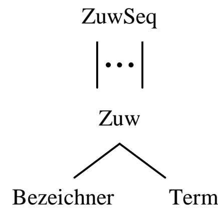
5.5 Ein Interpreter für Zuweisungssequenzen

Der oben beschriebenen Interpreter für primitive Programme wird jetzt so erweitert, dass statt primitiver Zuweisungen beliebige Zuweisungen ausgeführt werden können.

Hierzu führen wir zwei neue Datentypen „Zuw(eisung)“ und „Zuweisungssequenz“ (kurz: „ZuwSeq“) ein.

Objekte:

Ein Objekt vom Typ „Zuw“ ist eine Zuweisung der Gestalt „bez := term” wie x := y+(3-y). Ein Objekt vom Typ „ZuwSeq“ ist eine Sequenz bestehend aus beliebigen Zuweisungen wie BEGIN x:=3; y:=x; y:=y+1 END.



Operationen:

Modellierung:

(y:= 2*(x-z), {x: 6; y: 3; z: 4}) → [ZuwAusführen: Zuw * Zustand → Zustand] → {x: 6; y: 4; z: 4}

Modellierung:

(BEGIN x:=3; y:=x; y:=y+1 END, {x: 4; y: 7}) → [SeqAusführen: ZuwSeq * Zustand → Zustand] → {x: 3; y: 4}

Die Definitionen erhält man direkt, indem man die bereits erstellten Definitionen geringfügig abändert.

Definition / Reduktionsregeln:

ZuwAusführen((bez, t), zustand) →
    NeuerZustand((bez, TermWert(t, zustand)), zustand)

Definition / Reduktionsregeln:

SeqAusführen([], zustand) → zustand

SeqAusführen(zuw :: restZuweisungen, zustand) →
    SeqAusführen(restZuweisungen, ZuwAusführen(zuw, zustand))

Implementierung:

```
let ZuwAusführen = function
  ((bez, t), zustand) ->
    NeuerZustand((bez, TermWert(t, zustand)), zustand);;
ZuwAusführen : (string * Term) * (string * int) list -> (string * int)
list = <fun>

let rec SeqAusführen = function
  ([], zustand) -> zustand |
  (zuw::restZuweisungen, zustand) ->
   SeqAusführen(restZuweisungen, ZuwAusführen(zuw, zustand));;
SeqAusführen :
 (string * Term) list * (string * int) list -> (string * int) list = <fun>
```

Testaufrufe:

```
let VarZustand = [ ("x", 3); ("y", 7); ("z", 4) ];;
VarZustand : (string * int) list = ["x", 3; "y", 7; "z", 4]

let Progr  = [ ("x", term(var("y"),"-",var("x")));
               ("y", term(var("y"),"-",var("x")));
               ("x", term(var("x"),"+",var("y"))) ];;
Progr : (string * Term) list =
 ["x", term (var "y", "-", var "x"); "y", term (var "y", "-", var "x");
  "x", term (var "x", "+", var "y")]

SeqAusführen(Progr, VarZustand);;
- : (string * int) list = ["x", 7; "y", 3; "z", 4]
```

5.6 Anweisungen – Kontrollstrukturen

Ziel ist es, den Interpreter so zu erweitern, dass auch Fallunterscheidungen und Wiederholungen ausgeführt werden können. Insbesondere soll es möglich sein, Programme der folgenden Gestalt auszuführen.

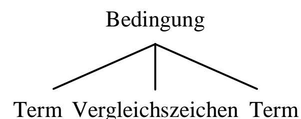
```
BEGIN
p := 1;
WHILE u > 0 DO
    BEGIN
    IF u mod 2 = 1 THEN
         BEGIN
         u := u - 1;
         p := p * b
         END;
    u := u div 2;
    b := b * b
    END
END
```

Es ergibt sich zunächst die Schwierigkeit, wie man solch komplexe Programme als Objekte eines bestimmten Typs in CAML darstellen kann. Innerhalb einer Anweisungssequenz können verschiedene Anweisungstypen vorkommen. Will man Anweisungssequenzen mit Hilfe von Listen darstellen, so muss ein neuer (einheitlicher) Typ für Anweisungen definiert werden.

Entscheidungs- und Wiederholungsanweisungen enthalten Bedingungen. Diese sollen zunächst betrachtet werden. Wir führen zu diesem Zweck den Datentyp „Bedingung“ ein.

Objekte:

Ein Objekt vom Typ „Bedingung“ hat die Struktur: Term, Vergleichszeichen, Term.



Beispiel: x > 0; x = y; x+2 <> 3*x

Zur Auswertung von Bedingungen muss der Wahrheitswert einer Bedingung ermittelt werden.

Operationen:

Modellierung:

(x>z-4, {x:4; y:2;z:8}) → BedWert [Bedingung * Zustand → boole] → false

Definition / Reduktionsregeln:

BedWert((t1,“>“,t2),zustand) → (TermWert(t1,zustand) > TermWert(t2,zustand))

...

Korrektheitsüberlegung:

$$\underbrace{\text{BedWert}(x > z-4, \{x:4; y:2; z:8\})}_{\text{false}} \rightarrow$$

$$\underbrace{\underbrace{\text{TermWert}(x, \{x:4; y:2; z:8\})}_{4} > \underbrace{\text{TermWert}(z-4, \{x:4; y:2; z:8\})}_{4}}_{\text{false}}$$

Implementierung:

```
(* Definition der Operation BooleWert - dient der Auswertung boolescher
   Terme *)

let rec BooleWert = function
   (term(term1, "=", term2), zustand)
      -> TermWert(term1, zustand) = TermWert(term2, zustand) |
   (term(term1, ">", term2), zustand)
      -> TermWert(term1, zustand) > TermWert(term2, zustand) |
   (term(term1, "<", term2), zustand)
      -> TermWert(term1, zustand) < TermWert(term2, zustand) |
   (term(term1, "<>", term2), zustand)
      -> TermWert(term1, zustand) <> TermWert(term2, zustand) |
   (term(term1, ">=", term2), zustand)
      -> TermWert(term1, zustand) >= TermWert(term2, zustand) |
   (term(term1, "<=", term2), zustand)
      -> TermWert(term1, zustand) <= TermWert(term2, zustand) |
   (_, zustand)
      -> raise(Failure "kein korrekter Term");;
BooleWert : Term * (string * int) list -> bool = <fun>
```

Test der Operation BooleWert:

```
let BooleTerm = term(var("y"), ">", term(num(4), "div", num(2)));;
BooleTerm : Term = term (var "y", ">", term (num 4, "div", num 2))

let VarZustand = [ ("x", 3); ("y", 7); ("z", 4) ];;
VarZustand : (string * int) list = ["x", 3; "y", 7; "z", 4]

BooleWert(BooleTerm, VarZustand);;
- : bool = true
```

Als Nächstes wird der Typ „Anweisung“ konzipiert. Man erkennt schnell, dass es sich bei der Struktur der Objekte, ähnlich wie beim Typ „Term“ , um eine rekursive Struktur handelt.

Objekte:

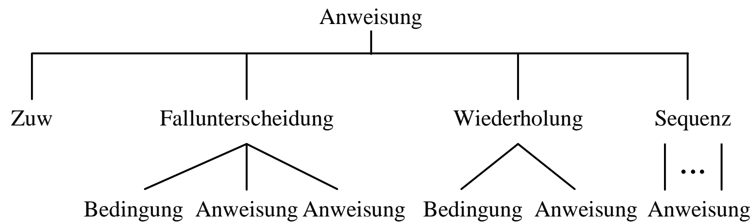
Ein Objekt vom Typ „Anweisung” ist entweder

- eine Zuweisung der Gestalt „bez := rechenterm“, oder
- eine Fallunterscheidungsanweisung der Gestalt „IF bed THEN anw_1 ELSE anw_2“, wobei „anw_1“ und „anw_2“ selbst wieder Anweisungen sind, oder
- eine Wiederholungsanweisung der Gestalt „WHILE bed DO anw“, wobei „anw“ selbst wieder eine Anweisungen ist, oder
- eine Sequenzanweisung der Gestalt „BEGIN anw_1; ...; anw_k END“, wobei „anw_1“, ... , „anw_k“ selbst wieder Anweisungen sind.

Übersicht:

Anweisung	Darstellung (konzeptionell)	Struktur
Zuweisung	bez := t	Bezeichner * Term
Fallunterscheidung	IF bed THEN anw1 ELSE anw2	Bedingung * Anweisung * Anweisung
Wiederholung	WHILE bed DO anw	Bedingung * Anweisung
Sequenz	BEGIN anw1; anw2; anw3 END	Anweisung list

Struktur:



Der Datentyp „Anweisung“ soll jetzt in CAML implementiert werden. Hierzu müssen in CAML neue Anweisungs-Objekte konstruiert werden. Wir führen hierzu die Konstruktoren „zw“, „fu“, „wh“ und „sq“ ein.

Anweisung	Darstellung (konzeptionell)	Darstellung als CAML-Objekt
Zuweisung	bez := t	zw(bez,t)

Fallunterscheidung	IF bed THEN
	anw1
	ELSE
	anw 2
		fu(bed, anw1, anw2)

Wiederholung	WHILE bed DO anw	wh(bed,anw)
Sequenz	BEGIN anw1; anw2; anw3 END	sq([anw1, anw3, anw3])

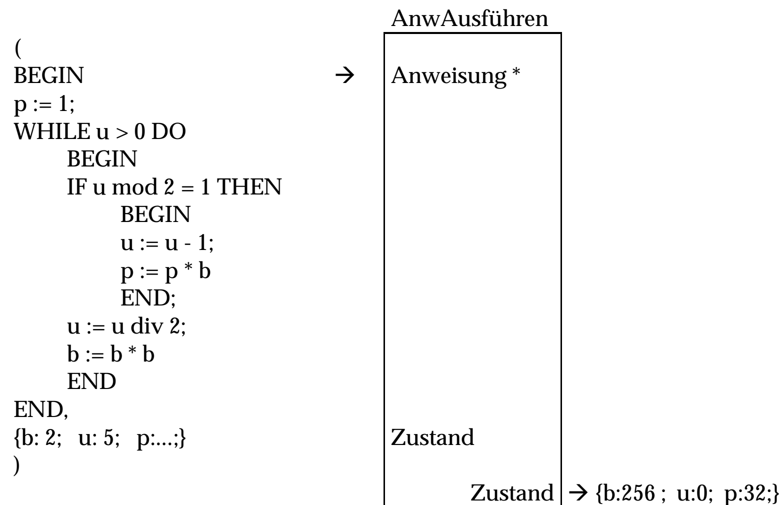
Operationen:

Zur Auswertung von Anweisungen wird eine Operation „AnwAusführen“ definiert. Um das Verhalten dieser Operation besser verstehen zu können, wird das oben vorgegebene Programm zunächst per Hand ausgewertet. Wir betrachten hierzu den Ausgangszustand {b: 2; u: 5; p: ...}.

Kontrollbedingung	Wert	Zuweisung	Zustand b	u	p
			2	5	
		p := 1			1
u > 0	true				
u mod 2 = 1	true				
		u := u - 1		4	
		p := p * b			2
		u := u div 2		2	
		b := b * b	4		
u > 0	true				
u mod 2 = 1	false				
		u := u div 2		1	
		b := b * b	16		
u > 0	true				
u mod 2 = 1	true				
		u := u - 1		0	
		p := p * b			32
		u := u div 2		0	
		b := b * b	256		
u > 0	false				

Die Operation „AnwAusführen“ kann somit wie folgt beschrieben werden:

Modellierung:



Verhaltensbeschreibung: Bei Eingabe einer Anweisung und eines Zustandes wird der neue Zustand ausgegeben, den man erhält, wenn man die eingegebene Anweisung bezüglich des eingegebenen Zustands ausführt.

Definition / Reduktionsregeln:

AnwAusführen(zw(bez, ausdruck), zustand)
  -> NeuerZustand((bez, TermWert(ausdruck, zustand)), zustand)

AnwAusführen(sq([ ]), zustand)
  -> zustand

AnwAusführen(sq(anw :: anwListe), zustand)
  -> AnwAusführen(sq(anwListe), AnwAusführen(anw,zustand))

AnwAusführen(fu(bed, anw1, anw2), zustand)
  -> if BooleWert(bed, zustand)
    then AnwAusführen(anw1, zustand)
    else AnwAusführen(anw2, zustand)

AnwAusführen(wh(bed, anw), zustand)
  -> if BooleWert(bed, zustand)
    then AnwAusführen(wh(bed, anw), AnwAusführen(anw, zustand))
    else zustand

Implementierung:

```
(* Definition eines neuen Typs für Anweisungen *)

type Bezeichner == string;;
Type Bezeichner defined.

type Bedingung == Term;;
```



```
Type Bedingung defined.

type Anweisung =
   zw of Bezeichner * Term |
   sq of Anweisung list |
   fu of  Bedingung * Anweisung * Anweisung |
   wh of  Bedingung * Anweisung;;
Type Anweisung defined.

(* Definition der Operation AnwAuswerten *)

let rec AnwAusführen = function
   (zw(bez, ausdruck), zustand)
      -> NeuerZustand((bez, TermWert(ausdruck, zustand)), zustand) |
   (sq([ ]), zustand)
      -> zustand |
   (sq(anw :: anwListe), zustand)
      -> AnwAusführen(sq(anwListe), AnwAusführen(anw,zustand)) |
   (fu(bed, anw1, anw2), zustand)
      -> if BooleWert(bed, zustand)
              then AnwAusführen(anw1, zustand)
              else AnwAusführen(anw2, zustand) |
   (wh(bed, anw), zustand)
      -> if BooleWert(bed, zustand)
              then AnwAusführen(wh(bed, anw), AnwAusführen(anw, zustand))
              else zustand ;;
AnwAusführen : Anweisung * (string * int) list -> (string * int) list = <fun>
```

1. Test des Interpreters:

```
let Programm1 =

sq( [
      zw("y", num(5));
      wh( term(var("y"), ">", num(0)),
            zw("y", term(var("y"), "-", num(1))))
    ] );;

Programm1 : Anweisung =
 sq
  [zw ("y", num 5);
   wh (term (var "y", ">", num 0), zw ("y", term (var "y", "-", num 1)))]

let VarZustand1 = [ ("x", 3); ("y", 3); ("z", 0)];;
VarZustand1 : (string * int) list = ["x", 3; "y", 3; "z", 0]

AnwAusführen(Programm1, VarZustand1);;
- : (string * int) list = ["x", 3; "y", 0; "z", 0]
```

2. Test des Interpreters

```
let Programm2 =

sq( [
      zw("p", num(1));
      wh( term(var("u"),">",num(0)),
            sq( [
                  fu( term(term(var("u"),"mod",num(2)),"=",num(1)),
                        sq( [
                             zw("u", term(var("u"),"-",num(1)));
                             zw("p", term(var("p"),"*",var("b")))
                            ] ),
                        sq ( [ ] )
                     );
                  zw("u", term(var("u"),"div",num(2)));
                  zw("b", term(var("b"),"*",var("b")))
                ] )
          )
     ] );;

Programm2 : Anweisung =
 sq
  [zw ("p", num 1);
   wh
    (term (var "u", ">", num 0),
     sq
      [fu
        (term (term (var "u", "mod", num 2), "=", num 1),
         sq
          [zw ("u", term (var "u", "-", num 1));
           zw ("p", term (var "p", "*", var "b"))], sq []);
       zw ("u", term (var "u", "div", num 2));
       zw ("b", term (var "b", "*", var "b"))])]

let VarZustand2 = [ ("b", 2); ("u", 5) ];;
VarZustand2 : (string * int) list = ["b", 2; "u", 5]
- : (string * int) list = ["b", 256; "u", 0; "p", 32]
```


5.7 Prototyping

Bisher wurden ausschließlich Konstrukte betrachtet, die die Programmiersprache Pascal zur Verfügung stellt. Ziel soll es jetzt sein, einen Interpreter-Prototyp zu erstellen, der „neue“, nicht in Pascal vorkommende Konstrukte erlaubt. Wir beschränken uns hier darauf, zwei weitere Kontrollstrukturen einzuführen. Eine Vielzahl von zusätzlichen Erweiterungsmöglichkeiten ist denkbar (z. B.: Einführung eines Prozedurkonzeps; Einführung eines Typkonzeps), dies wird hier aber nicht weiterverfolgt.

Erweiterung: Multizuweisungen

Eine Multizuweisung soll aus mehreren, gleichzeitig auszuführenden Zuweisungen bestehen.

Beispiel 1:

```
{x: 4; y: 2; ...}
(x, y) := (y, x)
{x: 2; y: 4; ...}
```

Hier soll der Variablen x der Wert der Variablen y und der Variablen y der Wert der Variablen x zugeordnet werden.

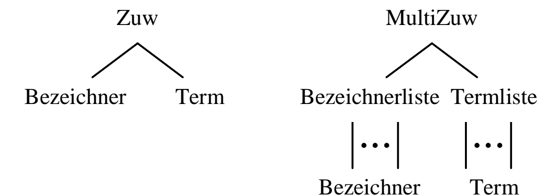
Beispiel 2:

```
{x: 3; y: 7; z: 2; ... }
(x, y, z) := (2, x, x+y)
{x: 2; y: 3; z: 10; ... }
```

Man beachte, dass zunächst alle Terme der rechten Seite ausgewertet werden. Die ermittelten Werte werden dann den (verschiedenen) Variablen der linken Seite zugewiesen.

Objekte:

Eine Multizuweisung besteht aus einer Liste von Bezeichnern und einer Liste von Termen. Die folgende Darstellung macht die Verallgemeinerung der Zuweisung deutlich.



Implementierung:

```
type Anweisung =
   zw of Bezeichner * Term |
   ...
   mzw of (Bezeichner list) * (Term list) ;;
```

Operationen:

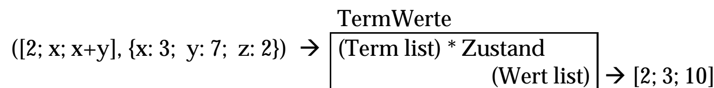
Bei der Festlegung der Abarbeitung von Multizuweisungen orientieren wir uns an der Abarbeitung „einfacher“ Anweisungen.

Definition / Reduktionsregeln:

ZuwAusführen((bez, t), zustand) →
  NeuerZustand((bez, TermWert(t, zustand)), zustand)

Die Definition zeigt, dass der Termwert bestimmt und der Variablen zugewiesen wird. In der verallgemeinerten Situation müssen jetzt die Termwerte einer Liste von Termen bestimmt und den entsprechenden Variablen zugewiesen werden. Wir modellieren hierzu neue Operationen:

Modellierung:

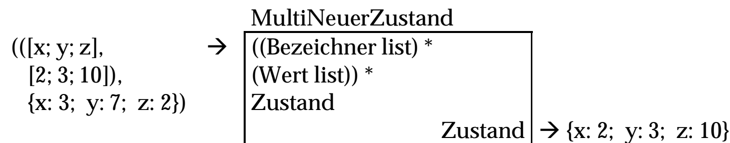


Verhaltensbeschreibung: Bei Eingabe einer Liste von Termen und eines Zustandes wird eine Liste der Termwerte ausgegeben.

Definition / Reduktionsregeln:

TermWerte(([], zustand)) -> []
TermWerte(t::tRestListe, zustand) ->
  TermWert(t, zustand) :: TermWerte(tRestListe, zustand);;

Modellierung:



Verhaltensbeschreibung: Bei Eingabe eines Paares bestehend aus einer Bezeichnerliste und einer Werteliste sowie eines Zustandes wird der Zustand ausgegeben, der entsteht, wenn den eingegebenen Bezeichnern die eingegebenen Werte neu zugeordnet werden.

Definition / Reduktionsregeln:

MultiNeuerZustand(([], []), zustand) -> zustand
MultiNeuerZustand(([], w::ws), zustand) -> ?
MultiNeuerZustand((b::bs, []), zustand) -> ?
MultiNeuerZustand((b::bs, w::ws), zustand) ->
  MultiNeuerZustand((bs,ws),NeuerZustand((b,w),zustand));;

Implementierung:

```
let rec TermWerte = function
  ([], zustand) -> [] |
  (t::tRestListe, zustand) -> TermWert(t, zustand) ::
                              TermWerte(tRestListe, zustand);;

let rec MultiNeuerZustand = function
  (([], []), zustand) -> zustand |
  (([], w::ws), zustand) -> raise(Failure "!") |
  ((b::bs, []), zustand) -> raise(Failure "!") |
  ((b::bs, w::ws), zustand) ->
     MultiNeuerZustand((bs,ws),NeuerZustand((b,w), zustand));;
```

Ergänzung der Implementierung der Operation „AnwAusführen”:

```
let rec AnwAusführen = function
   (zw(bez, ausdruck), zustand)
      -> NeuerZustand((bez, TermWert(ausdruck, zustand)), zustand) |
         ...
   (mzw(bListe, tListe), zustand)
      -> MultiNeuerZustand((bListe, TermWerte(tListe, zustand)),
                           zustand);;
```

Test des erweiterten Interpreters:.

```
let Programm3 =
mzw(["x"; "y"],[var("y"); var("x")]);;
Programm3 : Anweisung = mzw (["x"; "y"], [var "y"; var "x"])

let VarZustand3 = [ ("x", 2); ("y", 4) ];;
VarZustand3 : (string * int) list = ["x", 2; "y", 4]

AnwAusführen(Programm3, VarZustand3);;
- : (string * int) list = ["x", 4; "y", 2]
```

Erweiterung: Mehrfachfallunterscheidungen

Eine Mehrfachfallunterscheidung stellt eine Verallgemeinerung der in Pascal zur Verfügung stehenden CASE-Anweisung dar.

Beispiel 1:

```
COND
  x>0: y:=x;
  x=0: y:=0;
  x<0: y:=-x
END
```

Es sollen mehrere Fälle der Reihe nach abgetestet werden. Es wird nicht verlangt, dass die vorkommenden Bedingungen disjunkt sind. Die Anweisung der ersten zutreffenden Bedingung soll ausgeführt werden.

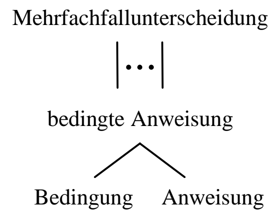
Beispiel 2:

```
COND
  x>=0: y:=x;
  x<=0: y:=-x
END
```

Wir skizzieren eine mögliche Realisierung:

Objekte:

Eine Mehrfachfallunterscheidung hat die folgende Struktur:



Implementierung:

```
type Anweisung =
  ...
  cnd of (Bedingung * Anweisung) list;;
```

Operationen:

Ergänzung der Implementierung der Operation AnwAusführen:

```
let rec AnwAusführen = function
   ...
   (cnd([]), zustand)
      -> zustand |
   (cnd((bed,anw)::rest), zustand)
      -> if BooleWert(bed, zustand)
              then AnwAusführen(anw, zustand)
              else AnwAusführen(cnd(rest), zustand);;
```

Test des erweiterten Interpreters:

```
let Programm4 =
sq( [
  zw("x", num(-3));
  cnd( [
       (term(var("x"), ">", num(0)), zw("y", var("x")));
       (term(var("x"), "=", num(0)), zw("y", num(0)));
       (term(var("x"), "<", num(0)), zw("y", term(num(0), "-",
                                                  var("x"))));
       ] )
    ] );;
```



```
Programm4 : Anweisung =
 sq
  [zw ("x", num -3);
   cnd
    [term (var "x", ">", num 0), zw ("y", var "x");
     term (var "x", "=", num 0), zw ("y", num 0);
     term (var "x", "<", num 0), zw ("y", term (num 0, "-",
                                                  var "x"))]]

let VarZustand4 = [ ("x", 2); ("y", 0) ];;
VarZustand4 : (string * int) list = ["x", 2; "y", 0]

AnwAusführen(Programm4, VarZustand4);;
-: (string * int) list = ["x", -3; "y", 3]
```

Was ist Prototyping?

Prototyping bedeutet (nach [Kieback u. a. 92]), bei der Systementwicklung frühzeitig ablauffähige Modelle (sog. „Prototypen“) des künftigen Anwendunggssystems zu erstellen und mit diesen zu experimentieren.

Prototyping ist aus zwei Gründen von besonderen Interesse: Zum einen schafft es eine Kommunikationsbasis für die Diskussion zwischen allen am Entwicklungsprozess beteiligten Gruppen (z. B.: Auftraggeber, Programmierer, ...). Anhand des lauffähigen Modells können frühzeitig Missverständnisse hinsichtlich des geplanten Produktes erkannt und beseitigt werden. Zum anderen ermöglicht Prototyping eine auf Experiment und Erfahrung gegründete Vorgehensweise bei der Software-Konstruktion.

5.8 Funktionale und imperative Programmierung

Ein Problem – zwei Lösungen:

Problem: Vertauschen

	Lösung 1:	**Lösung 2:**
Modellierung:	{x: 3; y:5; ...} Vertauschen {x: 5; y: 3; ...}	Vertauschen (3, 5) → int * int int * int → (5, 3)
Realisierung:	z := x x := y y := z	Vertauschen(x,y) → (y,x)
Anwendung:	{x: 3; y:5; z:0} z := x {x: 5; y: 3; z: 5} x := y {x: 3; y: 3; z: 5} y := z {x: 3; y: 5; z: 5}	Vertauschen(3,5) → (5,3)

Worin bestehen die Unterschiede zwischen den beiden Lösungen?

Zunächst unterscheiden sich die *Modellierungskonzepte*. Lösung 1 benutzt ein *Zustandskonzept*. Ein Ausgangszustand soll in einen Endzustand überführt werden. Lösung 2 benutzt ein *Zuordnungskonzept*. Einem (hier komplexen) Ausgangswert wird ein (hier komplexer) Endwert zugeordnet. Das Modellierungskonzept „Zustand" stellt eine Abstraktion eines Speichers dar. Die Zustände dienen dazu, Werte von Variablen zwischenzuspeichern.

Die *Realisierungen* weisen bereits größere Unterschiede auf. Lösung 1 benutzt ein *Aktionskonzept*. Es werden Aktionen mittels Anweisungen festgelegt, die beschreiben, wie die Vertauschung vorgenommen werden soll. Lösung 2 benutzt das *Funktionskonzept*. Es wird eine Funktion festgelegt, die die gewünschte Operation beschreibt.

Man beachte, dass im ersten Fall beschrieben wird, wie man unter den gegebenen Randbedingungen vertauschen kann, während im zweiten Fall beschrieben wird, was Vertauschen bedeutet. Man kann also hier unterscheiden zwischen „Wie-Programmierung" (*imperative Programmierung*) und „Was-Programmierung" (*deklarative Programmierung*). Wie-Programmierung ist sehr stark an einer Ausführungsmaschine orientiert. Mittels Anweisungen wird genau festgelegt, wie die hypothetische Maschine die Aktionen ausführen soll. Was-Programmierung ist dagegen direkt am Problem orientiert. Mittels Deklarationen (hier: Funktionsdeklarationen) wird die modellierte Operation genau festgelegt.

Auch die der *Anwendung* zu Grunde liegenden *Berechnungskonzepte* zeigen fundamentale Unterschiede auf. Lösung 1 benutzt das Konzept „*Zustandstransformation*". Die Aktionen verändern nach und nach den jeweiligen Variablenzustand. Dabei kann es vorkommen, dass Spuren hinterlassen werden, die direkt nichts mit der gewünschten Aktion zu tun haben. Im vorliegenden Fall erhält die Variable z einen Wert, den sie vorher nicht hatte und der mit der Vertauschung nichts direkt zu tun hat. Will man solche, nicht sehr transparenten, Veränderungen vermeiden, muss man spezielle Schutzmaßnahmen vorsehen. Dies wird aber in die Verantwortung des Programmierers gelegt. Lösung 2 benutzt das Konzept „*zielgerichtete Äquivalenzumformung*". Ein Ausgangsterm wird in einen äquivalenten Endterm überführt. Hierbei werden nur die beteiligten Terme bearbeitet. Seiteneffekte wie bei Lösung 1 können hierbei nicht auftreten.

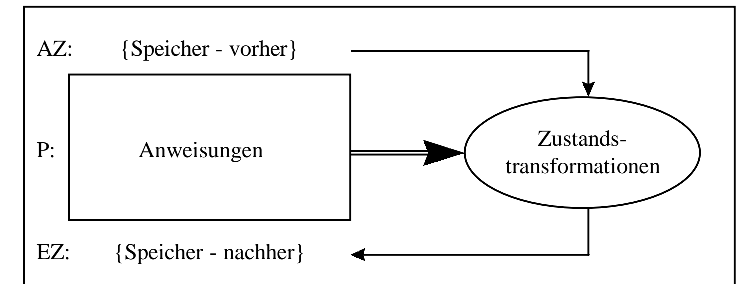
Wir fassen noch einmal den wesentlichen Unterschied zwischen imperativer und deklarativer Programmierung zusammen.

Bei der imperativen Programmierung wird beschrieben, wie berechnet werden soll.
Bei der deklarativen Programmierung wird beschrieben, was berechnet werden soll.
Imperative Programmierung ist eher maschinenorientiert.
Deklarative Programmierung ist eher problemorientiert.

Die folgenden Übersichten sollen die genannten Unterschiede verdeutlichen.

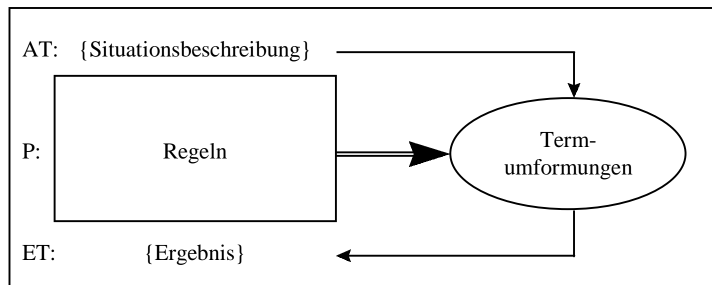
Imperative Programmierung:

Der imperativen Programmierung liegt eine maschinenorientierte Sichtweise zu Grunde.



Deklarative (hier funktionale) Programmierung:

Der deklarativen Programmierung liegt ein eher problemorientierte Sichtweise zu Grunde.



Bedeutung der deklarative Programmierung

Abschließend wird die Bedeutung der deklarativen Programmierung kurz skizziert.

Erster wesentlicher Faktor für die Relevanz deklarativer Konzepte ist die Tatsache, dass man bei der deklarativen Programmierung direkt auf der Problemebene arbeiten kann und nicht auf eine wie auch immer vorgegebene Maschinenebene wechseln muss. Dies ermöglicht oft eine adäquatere Behandlung der zu lösenden Probleme.

Als zweiter Faktor kann die Abstraktheit der deklarativen Programmierung angesehen werden. Moderne deklarative Programmiersprachen erzwingen einen strukturierten, abstrahierenden Programmierstil. Imperative Konstrukte, wie eine Seiteneffekte hervorrufende Wertzuweisung, stehen nicht zur Verfügung. Hieraus ergeben sich einige wesentliche Konsequenzen: Deklarative Programme sind meist kürzer, verständlicher und strukturierter als imperative Programme. Als Folge ergibt sich, dass Fehler leichter zu vermeiden bzw. leichter zu lokalisieren sind. Man macht sich diese Eigenschaften beim sog. „Rapid Prototyping“ zu Nutze. Hier geht es darum, schnell ein lauffähiges Produkt zu erstellen, wobei es mehr auf die Korrektheit des erstellten Produktes und weniger auf das Laufzeitverhalten ankommt.

Der dritte wesentliche Faktor liegt in der referentiellen Transparenz deklarativer Programme begründet. Hierunter versteht man die Tatsache, dass der Wert eines Programmteils nur von seiner Umgebung (der momentanen Variablenbelegung) und nicht vom Zeitpunkt seiner Auswertung abhängt. Teilberechnungen, d. h. die Auswertung unabhängiger Programmteile, können somit unabhängig voneinander – also insbesondere parallel – durchgeführt werden. Dies ist insofern von Bedeutung, als Fortschritte bei der technischen Realisierung von Parallelrechnern erzielt worden sind.

Den genannten Vorteilen eines deklarativen Programmierstils stehen folgende Nachteile gegenüber: Es gibt eine Reihe von Anwendungen, die adäquat mit zustandsbasierten Modellen bearbeitet werden, etwa interaktive Systeme. Solche Anwendungen lassen

sich oft besser mit imperativen Mitteln behandeln. Der zweite Nachteil liegt in einer vielfach mangelnden Effizienz begründet. Deklarative Programme laufen vergleichsweise langsam und benötigen viel Speicherkapazität. Der Grund hierfür kann wie folgt grob umrissen werden: Deklarative Programmiersprachen sind so konzipiert, dass der Benutzer durch direktes Arbeiten in der Problemebene entlastet wird. Dies hat natürlich zur Folge, dass das System durch erhöhten Platz- und Zeitbedarf zur Ausführung der Programme stärker belastet wird.

Raum für Notizen

6 Literatur

[Avenhaus 95] J. Avenhaus: Reduktionssysteme. Rechnen und Schließen in gleichungsdefinierten Strukturen. Berlin [u. a.]: Springer, 1995.

[Bird & Walder 92] R. Bird und P. Walder: Einführung in die funktionale Programmierung. München [u. a.]: Hanser [u. a.], 1992.

[Drumm & Stimm 95] H. Drumm und H. Stimm: Wissensverarbeitung mit PROLOG. Ein Einstieg in die Algorithmik. Landesmedienzentrum Rheinland-Pfalz (Hrsg.), 1995.

[Euteneuer 99] A. Euteneuer: Beitrag in „Standortplanung", PZ-Information Mathematik, Bad Kreuznach, Pädagogisches Zentrum Rheinland-Pfalz, i. Vorb.

[Fischbacher 97] T. Fischbacher: Funktionale Programmierung. In: LOG IN 17 (1997) Heft 3 / 4, S. 24-26.

[Hubwieser & Broy 95] P. Hubwieser und M. Broy: Der informationszentrierte Ansatz. Ein Vorschlag für eine zeitgemäße Form des Informatikunterrichts am Gymnasium. TUM-INFO-05-I9624-350/1.-FI. München, 1995.

[ISB 97] Staatliches Institut für Schulpädagogik und Bildungsforschung München (Hrsg.): Funktionales Programmieren in Gofer. Baustein zur Didaktik der Informatik. München, 1997.

[Kieback u. a. 92] A. Kieback, H. Lichter, M. Schneider-Hufschmidt und H. Züllinghoven: Prototyping in industriellen Software-Projekten. Informatik-Spektrum (1992) 15 : 65-77.

[Lehrplan 93] Ministerium für Bildung und Kultur (Hrsg.): Lehrplanentwurf Informatik. Mainz: Ministerium für Bildung und Kultur, 1993.

[Müller-Ewertz 99] G. Müller-Ewertz: Beitrag in „Standortplanung", PZ-Information Mathematik, Bad Kreuznach, Pädagogisches Zentrum Rheinland-Pfalz, i. Vorb.

[Puhlmann 98] H. Puhlmann: Funktionales Programmieren - Eine organische Verbindung von Informatikunterricht und Mathematik. In: LOG IN 18 (1998) Heft 2, S. 46-50.

[Schwill 93] A. Schwill: Funktionale Programmierung mit CAML. In: LOG IN 13 (1993) Heft 4, S. 20-30.

[Thiemann 94] P. Thiemann: Grundlagen der funktionalen Programmierung. Stuttgart: Teubner 1994.

[Wagenknecht 94] Christian Wagenknecht: Rekursion. Ein didaktischer Zugang mit Funktionen. Bonn: Dümmlers Verlag 1994.

[Wolff von Gudenberg 96] J. Wolff. von Gudenberg: Algorithmen, Datenstrukturen, Funktionale Programmierung. Eine praktische Einführung mit Caml Light. Bonn: Addison-Wesley 1996.

Hinweise zur Programmiersprache CAML
Bezugsquelle

CAML-light ist eine leicht portierbare, typisierte funktionale Sprache. CAML ist eine Abkürzung für „Categorical Abstract Machine Language“. Die Categorical Abstract Machine dient dazu, Funktionen zu definieren und Funktionsaufrufe auszuwerten.

Das komplette CAML-light-System kann für den privaten Gebrauch kostenlos vom Server der INRIA (`ftp://ftp.inria.fr/lang/caml-light/`) geladen werden. Es gibt unter anderem Versionen für LINUX, DOS, MS-Windows und Macintosh-Systeme. Zusatzinformationen über CAML-light findet man im WWW unter: `http://pauillac.inria.fr`