LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Informatik I

Kurzskriptum zur Vorlesung

Wintersemester 2002/03

F. Kröger

INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für Programmierung und Softwaretechnik

Inhaltsverzeichnis

Einleitung

Informatik (computer science)

- [DUDEN Informatik]:
  Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Computern.

- [Gesellschaft für Informatik: Studien- und Forschungsführer Informatik; Springer-Verlag]:
  Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.

- [Association for Computing Machinery]:
  Systematic study of algorithms and data structures, specifically
  (1) their formal properties,
  (2) their mathematical and linguistic realizations, and
  (3) their applications.

Teilbereiche der Informatik

(1) ***Technische Informatik***
Themen: Aufbau und Wirkungsweise von Computern und deren Komponenten sowie alle damit zusammenhängenden Fragen.

(2) ***Praktische Informatik***
Themen: Prinzipien und Techniken der Konstruktion von Informationsverarbeitungs-Systemen sowie deren Realisierung (Implementierung) auf Computern.

(3) ***Theoretische Informatik***
Themen: Theoretische Grundlegung und Durchdringung von Fragen und Konzepten der Informatik.

(4) ***Angewandte Informatik***
Themen: Anwendbarkeit von Informatik-Erkenntnissen in anderen Wissenschaften.

Neuerdings auch:

(5) ***Informatik und Gesellschaft***
Themen: Auswirkungen der Informationstechnik auf die Gesellschaft.

Inhalt dieser Vorlesung

- Einführung in einen zentralen Bereich von (2): Grundlegende Konzepte, Methoden und Techniken der systematischen Informationsverarbeitung,

- abgestützt auf Methoden aus (3).

Historische Entwicklung

- Altertum - Mittelalter: Verwendung des Abakus (lat.: abax, Tafel) als Hilfsmittel für die vier Grundrechenarten.
- 9. Jh.: Der persische Mathematiker Ibn Musa Al-Chwarismi schreibt das Lehrbuch *Kitab al jabr w´almuqubala* (*Regeln der Wiedereinsetzung und Reduktion*). Das Wort „Algorithmus“ geht auf seinen Namen zurück.
- 1623: Wilhelm Schickard konstruiert für Kepler eine Maschine, die addieren, subtrahieren, multiplizieren und dividieren kann. (Sie bleibt unbeachtet.)
- 1641: Blaise Pascal konstruiert eine Maschine, mit der man sechsstellige Zahlen addieren kann.
- 1647: Gottfried Wilhelm Leibniz konstruiert eine Rechenmaschine mit Staffelwalzen für die vier Grundrechenarten.
- 1774: Philipp Matthäus Hahn entwickelt eine mechanische Rechenmaschine, die erstmals zuverlässig arbeitet.
- 1838: Charles Babbage plant eine Maschine, bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene Lochkarten gesteuert wird.
- 1886: Hermann Hollerith entwickelt elektrisch arbeitende Zählmaschinen für Lochkarten, mit denen die statistischen Auswertungen von Volkszählungen in den USA vorgenommen werden.
- 1941: Konrad Zuse stellt mit der elektromechanischen Rechenanlage Z3 den ersten funktionsfähigen programmgesteuerten Rechenautomaten fertig. Die Maschine verwendet das binäre Zahlsystem.
- 1944: Howard H. Aiken erstellt die teilweise programmgesteuerte Rechenanlage MARK.
- 1946: J.P. Eckert und J.W. Mauchly stellen den ersten voll elektronischen Rechner ENIAC fertig.
- 1946-52: Auf der Grundlage der Ideen John v. Neumanns werden weitere Computer in Universitätslabors entwickelt („Von-Neumann-Rechner-Architektur“).
- 1949: M.V. Wilke stellt mit der EDSAC den ersten universellen Digitalrechner fertig.
- Ab 1950: Industrielle Rechnerentwicklung und -produktion.

Literaturhinweise

M.Broy: *Informatik. Eine grundlegende Einführung. Teil 1.*
Springer-Verlag 1992.

M.R. Hansen, H. Rischel: *Introduction to Programming using SML.*
Addison-Wesley 1999.

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

F. Kröger: *Einführung in die Informatik - Algorithmenentwicklung.*
Springer-Verlag 1991.

L.C. Paulson: *ML for the Working Programmer.*
Cambridge University Press 1991 (2. Auflage 1996).

J.D. Ullman: *Elements of ML Programming, ML97 Edition.*
Prentice-Hall 1998.

A. Wikström: *Functional Programming Using Standard ML.*
Prentice-Hall 1987.

Kapitel 1
Informationsverarbeitung durch Programme

1.1 Informationen und Daten

Grundlage jeglicher maschineller Informationsverarbeitung (und zentraler Begriff der Informatik):

Algorithmus: Systematische, „schematisch“ („automatisch“, „mechanisch“) ausführbare Verarbeitungsvorschrift.

Alltags-Algorithmen

(1) Bedienungsanleitungen, Kochrezepte, Spielregeln, ...
Beispiel: „Verrühren Sie 100g Mehl, 100ml Milch, 2 Eier, 1g Salz, 10g Zucker, lassen Sie den entstandenen Teig 10 Minuten quellen. Zerlassen Sie Butter in einer Pfanne, geben Sie eine Portion Teig hinein, ...“

(2) Mathematische Berechnungsverfahren
Beispiel: Verfahren zur Berechnung der Summe $1 + 2 + 3 + \ldots + n$ für gegebenes $n \in \mathbb{N}_0$

Bei (1) zu verarbeiten: (materielle) Dinge der „realen Welt“, nicht „mathematisch abstrahierbar“.

Bei (2) zu verarbeiten: mathematische, abstrakte, immaterielle Objekte (hier: Zahlen).

Verallgemeinerung von (2)

Beispiel: Zu gegebener Abfahrts- und Ankunftszeit eines Zuges ist seine Fahrtzeit zu berechnen.

- Abfahrts-/Ankunftszeit, Fahrtzeit: Dinge der realen Welt,
  z.B.: „Dreizehn Uhr zehn“.
- Abstraktion liefert: Mathematische Modelle der realen Dinge,
  z.B.: Die Zahlen „dreizehn“ und „zehn“.
- Verarbeitung der abstrakten „Eingabe“-Objekte liefert ein abstraktes „Ausgabe“-Objekt,
  z.B.: Die Zahl „einhundertunddreiundsechzig“.
- Rückinterpretation liefert: Antwort auf die gestellte Frage der realen Welt,
  z.B.: „Die Fahrtzeit beträgt einhundertdreiundsechzig Minuten“.

Mischformen

Beispiel: Zu gegebener Abfahrtszeit und gegebenem Zielort eines Zuges soll eine entsprechende Fahrkarte ausgegeben werden.

- Abfahrtszeit: modellierbar wie oben.
- Zielort: modellierbar in obigem Sinn (siehe später).
- Fahrkarte: materielles Objekt, nicht modellierbar in obigem Sinn.
- Abtrennung einer Informatik-Aufgabe:
  Ergebnis der Verarbeitung ist kein (abstraktes) Objekt, sondern ein „Steuersignal“ an einen mechanischen Apparateteil (der die physikalische Erstellung und Ausgabe der Fahrkarte erledigt).
- Steuersignale: Spezielle Art von Algorithmus-Ausgaben.

Darstellung mathematischer Objekte (im Fahrtzeit-Beispiel)

- Zu verarbeiten (nach Abstraktion): Zahlen, z.B. „dreizehn“. Zur Verarbeitung notwendig: ***Darstellung*** (***Repräsentation***) der Zahlen (vgl. Algorithmusbeispiel (2)).
- Typische Darstellung von „dreizehn“:

  13 (***Dezimaldarstellung***).

  Andere Darstellungsmöglichkeiten:

  ||||||||||||
  1101 (***Binärdarstellung***)
  XIII
  DREIZEHN
  (usw.).

Begriffsbestimmungen

- ***Information***: Ein Bedeutungsinhalt.
  Beispiele: Abfahrtszeit („Reale-Welt“-Information),
  Eine Zahl (***abstrakte Information***).
- ***Datum*** (***Datenelement***): Abstrakte Information in einer konkreten Darstellung (Beispiel: eine Zahl in Dezimaldarstellung).
- ***Datendarstellung***: Art der Darstellung von Daten (Beispiel: Dezimaldarstellung von Zahlen).
- ***Informationsverarbeitung*** (***Datenverarbeitung***): Verarbeitung von Darstellungen von abstrakten Informationen; kurz: Verarbeitung von Daten.

1.2 Algorithmen

Algorithmus im Fahrtzeit-Beispiel

(Vereinfachende Annahme: Keine Fahrt geht über Mitternacht hinaus.)

Eingabe: natürliche Zahlen; ***Parameter*** hierfür: $s_{ab}, m_{ab}, s_{an}, m_{an}$;
Ergebnis: natürliche Zahl;
Berechnung:

$$\text{Ergebnis} = (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}.$$

Algorithmus(idee) beinhaltet:

- Daten ($s_{ab}, \ldots, m_{an}, 60$) und mathematische Operationen (Addition, Subtraktion, Multiplikation) auf diesen („elementare Verarbeitungsschritte“),
- „Zusammensetzung“ des Berechnungsvorgangs.

Ausführungsbeispiel:

Die Eingabe von $13, 10, 15, 53$ für $s_{ab}, m_{ab}, s_{an}, m_{an}$ liefert das Ergebnis 163.

Der Algorithmusbegriff

- Ein Algorithmus löst (typischerweise) eine Klasse von Aufgaben, die durch seine ***Parameter*** bestimmt ist. Eine ***Eingabe*** besteht aus konkreten (aktuell zu verarbeitenden) Daten für die Parameter.
- Eingaben erzeugen ***Ausführungen*** eines Algorithmus, die in der Regel ***Resultate*** (***Ergebnisse***) liefern. Diese können Daten oder Steuersignale sein.
- Ein Algorithmus verwendet ***elementare Verarbeitungsschritte*** und beschreibt, in welcher Weise diese auszuführen sind.

Grundanforderungen an einen Algorithmus

- Präzise Aufschreibung: Die Verarbeitungsvorschrift muss (ebenso wie die zu verarbeitenden Daten) unmissverständlich aufgeschrieben sein (***Darstellung, Repräsentation*** des Algorithmus).
- Endliche Aufschreibung: Die Darstellung muss in endlicher Form gegeben sein.
- ***Effektivität*** der elementaren Verarbeitungsschritte: Jeder elementare Verarbeitungsschritt muss von der zugrunde liegenden „Verarbeitungseinheit“ (***Prozessor***) tatsächlich ausführbar sein.

Weitere Merkmale von Algorithmen

Ein Algorithmus heißt

- ***terminierend für eine Eingabe***, wenn jede mögliche Ausführung für diese Eingabe nach endlich vielen Schritten endet;
- ***terminierend***, wenn er für jede mögliche Eingabe terminierend ist (***terminiert***);
- ***deterministisch***, wenn die Reihenfolge der auszuführenden elementaren Verarbeitungsschritte für jede Eingabe eindeutig bestimmt ist, andernfalls heißt er ***nicht-deterministisch***;
- ***determiniert***, wenn das Resultat für jede Eingabe eindeutig bestimmt ist;
- ***sequenziell***, wenn in allen Ausführungen die Verarbeitungsschritte stets hintereinander ausgeführt werden;
- ***parallel*** (***nebenläufig***), wenn gewisse Verarbeitungsschritte nebeneinander ausgeführt werden.

Verarbeitungsziele von Algorithmen

- Algorithmus ist terminierend und liefert Datenelement(e) als Resultat.
- Algorithmus ist terminierend und liefert Steuersignale (und eventuell zusätzlich Daten) als Resultat (im Folgenden nicht behandelt).
- Algorithmus ist nicht terminierend und liefert während seiner Ausführung fortlaufend Steuersignale und/oder Daten (typische Anwendungen: Prozesssteuerung, Betriebssysteme, Datenübertragung in Netzen usw.). Von solchen Algorithmen erzeugte Systeme heißen ***reaktive Systeme*** (im Folgenden nicht behandelt).

Arten von Algorithmen

- ***Grundlegende Algorithmen***
  Algorithmen für immer wieder verwendbare Verarbeitungsschritte (***Grundaufgaben***) bei typischen, insbesondere komplexen Daten, ohne direkten Bezug auf konkret gegebene Anwendung in der realen Welt („auf Vorrat“).
- ***Anwendungs-Algorithmen***
  Algorithmen zur Lösung konkreter Aufgaben der realen Welt; deren Kenntnis (d.h. die Kenntnis, was die Daten modellieren) ist i.a. für die Entwicklung notwendig.

1.3 Programmierung

Mögliche Darstellungen des Fahrtzeit-Algorithmus

(1) Vollkommen verbale Darstellung (unbrauchbar):
Subtrahiere die durch s_{ab} gegebene Zahl von der durch s_{an} gegebenen Zahl, multipliziere das Ergebnis mit sechzig, addiere die durch m_{an} gegebene Zahl und subtrahiere schließlich die durch m_{ab} gegebene Zahl. Die sich ergebende Zahl ist das Ergebnis.

(2) Mathematisch orientierte Darstellung (i.w. wie bei Formulierung der Lösungsidee):
Berechne $(s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$;
Die berechnete Zahl ist das Ergebnis.

(3) Vollkommen formale Darstellung in einer Programmiersprache (notwendig zur Ausführung des Algorithmus auf einem Computer):

z.B.

in SML (in dieser Vorlesung verwendete Programmiersprache):

```
fun fahrtzeit(s_ab,m_ab,s_an,m_an) =
    (s_an - s_ab) * 60 + m_an - m_ab
```

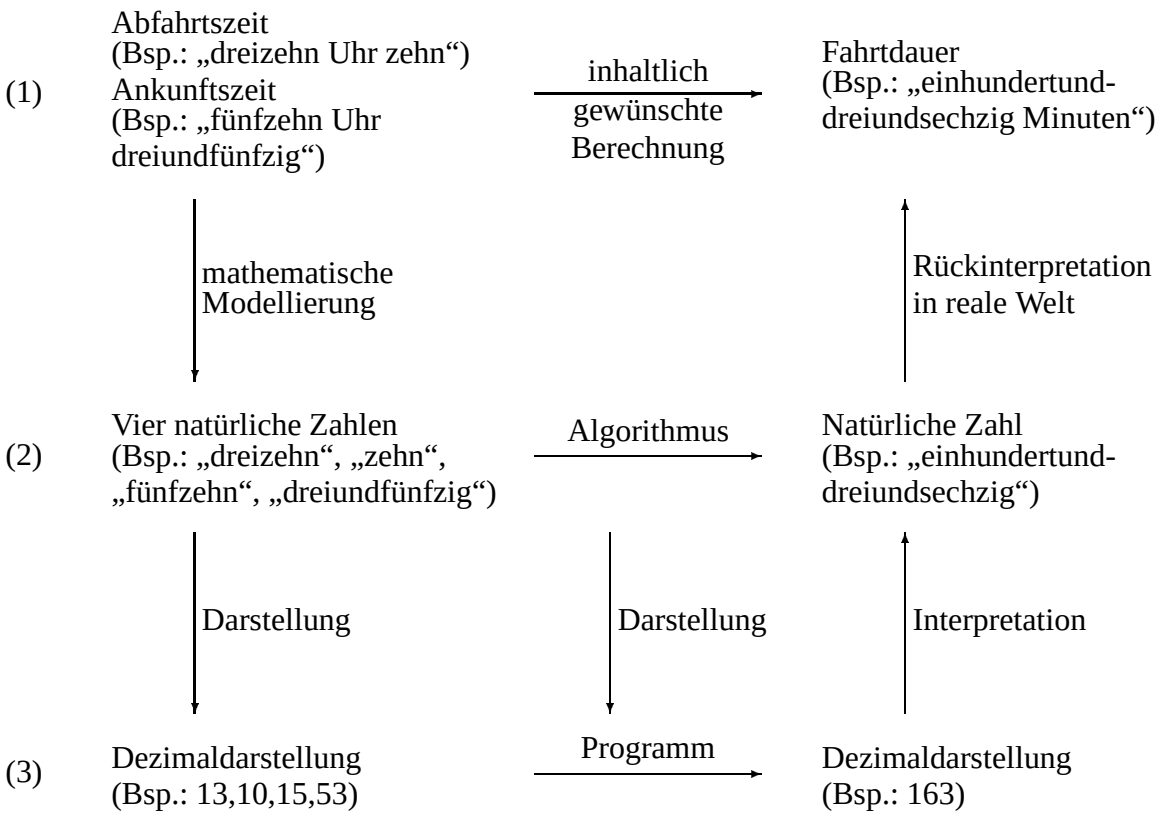
in Java (in Informatik II verwendete Programmiersprache):

```
public int fahrtzeit(int sab,int mab,int san,int man)
{
 return((san - sab) * 60 + man - mab);
}
```

Begriffsbestimmungen

- ***Algorithmische Sprache***: Eine „formal beschriebene Sprache“ (siehe später) zur Darstellung von Algorithmen (einschließlich der zu verarbeitenden Daten).
- ***Programmiersprache***: Eine algorithmische Sprache, die auf die Bedürfnisse der Ausführung von Algorithmen auf Computern zugeschnitten ist.
- ***Programm***: Ein in einer Programmiersprache dargestellter Algorithmus.

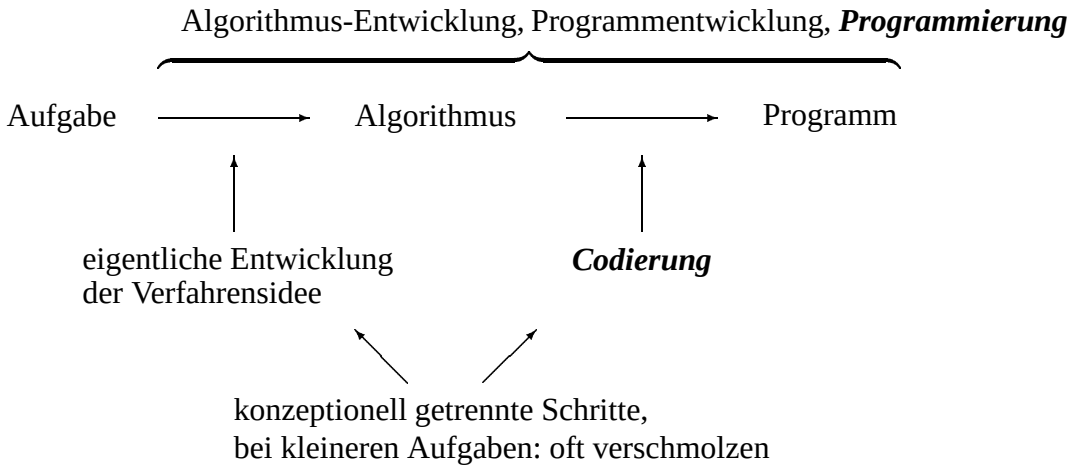
Gesamtbild (im Fahrtzeit-Beispiel)



(1): Reale-Welt-Ebene.
(2): Ebene der abstrakten Informationen (abstrakten Bedeutungen, ***semantische*** Ebene).
(3): Darstellungs-Ebene (***syntaktische*** Ebene).

Entwicklung von Algorithmen

Zentrale Aufgabe des Informatikers; Grundmuster:



Praxis der Programmierung

- „Können“ des Informatikers misst sich nicht daran, wieviele Programmiersprachen er kennt (insgesamt gibt es mehrere tausend), sondern daran, ob er „gute“ Algorithmen entwickeln kann.
- Dazu notwendig: Die Kenntnis der in den Programmiersprachen immer wiederkehrenden grundlegenden „algorithmischen Konzepte“.
- Statt einen Algorithmus sofort in einer Programmiersprache niederzuschreiben („loszuprogrammieren“), ist es (insbesondere bei komplexen Aufgaben) methodisch besser, den Algorithmus (genauer: seine einzelnen Entwicklungsstufen, siehe später) zunächst in einer „halbformalen“ Form aufzuschreiben (wie z.B. die Darstellung (2) im Fahrtzeit-Beispiel).
- Typische Form solcher Aufschreibungen: Darstellung in ***Pseudocode*** (programmiersprachenähnlich, unter Verwendung typischer algorithmischer Konzepte, mathematischer Schreibweisen und eventuell auch verbaler Zusätze).

Beschreibung des Fahrtzeit-Algorithmus in Pseudocode

algorithm Fahrtzeit
input $s_{ab}, m_{ab}, s_{an}, m_{an}$: **nat**
output : **nat**
pre Keine Fahrt geht über Mitternacht hinaus
result Berechnung der Fahrtzeit bei Abfahrt um „s_{ab} Uhr m_{ab}“
und Ankunft um „s_{an} Uhr m_{an}“
begin
$(s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$
end

Bemerkungen

- Auch Programmiersprachen haben sich geschichtlich entwickelt.

  In den frühen Jahren der Programmierung: „Maschinen-“ oder „maschinennahe“ Sprachen. Programme in solchen Sprachen für den Menschen sehr unübersichtlich und nur schwer verständlich.

  Die Bereitstellung von „problemorientierten“ („höheren“) Programmiersprachen (und ihre automatische Übersetzung in „Maschinencode“), in denen (sinngemäß) so formuliert werden kann wie oben, ist eine wichtige Errungenschaft der Informatik-Entwicklung.
- Die algorithmischen Konzepte von verschiedenen problemorientierten Programmiersprachen sind zum Teil gleichartig, zum Teil charakterisieren sie unterschiedliche ***Programmierstile***.
- In dieser Vorlesung behandelte Programmierstile: ***Funktionale*** (***applikative***) und ***imperative*** (***prozedurale***) Programmierung.

- Funktionale Programmierung: Algorithmus als Formulierung des funktionalen Zusammenhangs zwischen Eingabe- und Resultatdaten.
  Beispiel: Fahrtzeit-Algorithmus.
- Imperative Programmierung: Algorithmus „verändert Größen“ durch „Anweisungen“.
  Beispiel einer Anweisung: „Erhöhe x um 1“.

Programmierstile und Programmiersprachen

(wesentliche) Programmierstile	Programmiersprachen
Funktionale Programmierung	Lisp, ML, Miranda, Haskell
Imperative Programmierung	Fortran, Algol, Pascal, Modula, C
Objektorientierte Programmierung	Eiffel, Oberon, C++, Java
Logische Programmierung	Prolog (und Varianten)

Die Programmiersprache ML (genauer: SML)

- Entwickelt 1980 (R. Milner).
- Erweitert 1985 (D. MacQueen).
- Standardisiert Ende 80er Jahre (Standard ML, SML).
- Revidierte (aktuelle) Version 1997.

Kapitel 2
Konzepte funktionaler Programmierung

2.1 Funktionen und Terme

(Mathematische) Terminologie und Schreibweisen

- Es seien A und B Mengen. Eine ***Funktion*** (***Abbildung***) ***von*** A ***in*** B ist eine Zuordnung von genau einem Element $y \in B$ zu jedem Element x einer gewissen Teilmenge A' von A, geschrieben:

  $$x \mapsto y.$$

  A heißt ***Quelle***, B heißt ***Ziel***, und A' heißt ***Definitionsbereich*** der Funktion. Ist $A' = A$, so heißt die Funktion ***total***, andernfalls (d.h.: $A' \subsetneq A$) ***partiell***.

- $A \to B$ bezeichnet die Menge aller Funktionen von A in B. Ist $f \in A \to B$, so schreiben wir auch

  $$f : A \to B \qquad \text{und} \qquad f : x \mapsto y$$

  und $D(f)$ für den Definitionsbereich A' von f.

- Ist $f : A \to B$ und $a \in D(f)$, so liefert die ***Funktionsanwendung*** von f auf das ***Argument*** a das a vermöge f zugeordnete Element aus B, genannt ***Wert*** der Funktionsanwendung (von f für a).

- Schreibweisen für die Funktionsanwendung (von f auf a):

$f(a)$	(***Funktionsschreibweise***)
fa	(***Präfixschreibweise***)
af	(***Postfixschreibweise***)
$a_1 f\ a_2$	(falls $A = A_1 \times A_2$, $a = (a_1, a_2)$, ***Infixschreibweise***)

Typen

Daten, die von Algorithmen verarbeitet werden, werden üblicherweise in ***Typen*** (Mengen „gleichartiger“ Daten) eingeteilt. Typen (und ebenso die betreffenden Daten) können ***elementar*** oder ***zusammengesetzt*** sein. Beispiele von elementaren Typen (***Basistypen***) sind

$\mathbb{N}_0$:	Natürliche Zahlen, z.B. $13, 0, 8732$	in Pseudocode bezeichnet durch:	**nat**
$\mathbb{Z}$:	Ganze Zahlen, z.B. $13, -13, 0$		**int** („integer“)
$\mathbb{R}$:	Reelle Zahlen, z.B. $3.14, -82.0, 0.0$		**real**

Kartesische Produkte (Menge von ***Tupeln***) der Art

$\mathbb{R} \times \mathbb{R}$ (**real** $\times$ **real**),
$\mathbb{N}_0 \times \mathbb{Z} \times \mathbb{Z}$ (**nat** $\times$ **int** $\times$ **int**)
usw.

sind erste Beispiele von zusammengesetzten Typen.

Sprechweise: Ein Datenelement „ist vom Typ ...“ oder „hat den Typ ...“.

Fahrtzeitberechnung als funktionaler Algorithmus (vgl. Abschnitt 1.2)

- Einzelberechnungen für (jeweils durch vier natürliche Zahlen) gegebene Uhrzeiten:
  Ab: 13 Uhr 10, An: 15 Uhr 53: $(15 - 13) \cdot 60 + 53 - 10$
  Ab: 13 Uhr 24, An: 15 Uhr 12: $(15 - 13) \cdot 60 + 12 - 24$
  usw.

- Der ***Term***

  $$(x_1 - x_2) \cdot 60 + x_3 - x_4$$

  mit den (***freien***) Parametern („Platzhaltern“ für konkrete Daten) x_1, x_2, x_3, x_4 beschreibt das den Einzelberechnungen gleichermaßen zugrunde liegende allgemeine „Rechenmuster“.

- Der Term definiert eine Funktion

  $$\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0,$$
  $$(x_1, x_2, x_3, x_4) \mapsto (x_1 - x_2) \cdot 60 + x_3 - x_4,$$

  (mathematisch) geschrieben auch:

  $$\lambda(x_1, x_2, x_3, x_4) \,.\, (x_1 - x_2) \cdot 60 + x_3 - x_4.$$

  Die Bildung einer Funktion aus einem Term in dieser Weise heißt ***Funktionsabstraktion***. Die Parameter werden dabei ***gebunden***.

- Die Bezeichnungen der Parameter (und ihre Reihenfolge in $(x_1, \ldots, x_4)$) ist für die Bedeutung der durch Funktionsabstraktion gewonnenen Funktion irrelevant. Sie können durch beliebige andere ersetzt werden. Eine Notation der Funktion mit „mnemotechnisch günstigeren“ Bezeichnungen wäre etwa

  $$\lambda(s_{ab}, m_{ab}, s_{an}, m_{an}) \,.\, (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$$

- Die Funktion ist die Rechenvorschrift, d.h. der Algorithmus zur Fahrtzeitberechnung; Notation in Pseudocode (z.B.):

  function$(s_{ab}, m_{ab}, s_{an}, m_{an}) : (s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$

  oder (unter Einschluss der Quellen- und Zielangabe, ***getypte*** Beschreibung):

  function$(s_{ab}$: **nat**, m_{ab} : **nat**, s_{an} : **nat**, m_{an} : **nat**) **nat**:
  $(s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$

- Einzelberechnungen sind dann Ausführungen des Algorithmus für eine konkrete Eingabe und lassen sich als Anwendungen der Funktion (***Funktionsaufrufe***) mit der Eingabe als Argument formulieren, z.B. (jeweils in Präfixschreibweise):

  in mathematischer Notation:

  $$(\lambda(x_1, x_2, x_3, x_4)\,.\,(x_1 - x_2) * 60 + x_3 - x_4)\;(15, 13, 53, 10),$$
  $$(\lambda(s_{ab}, m_{ab}, s_{an}, m_{an})\,.\,(s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab})\;(13, 10, 15, 53),$$

  in Pseudocode:

  $$(\textbf{function}(s_{ab}, m_{ab}, s_{an}, m_{an}) : (s_{an} - s_{ab}) * 60 + m_{an} - m_{ab})\;(13, 10, 15, 53).$$

  Der Wert eines solchen Aufrufs ist das Resultat der betreffenden Algorithmusausführung.

  Beachte: Das Argument $(13, 10, 15, 53)$ ist in diesem Kontext zu verstehen als zusammengesetztes Datenelement (4-Tupel aus $\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$).

Namen für Funktionen

- Zur „einfacheren“ Verwendung von Funktionen: Bezeichnung durch (***Bindung*** an) ***Namen***, z.B.:

  in mathematischer Notation:

  $$Fahrtzeit : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0,$$
  $$Fahrtzeit : (s_{ab}, m_{ab}, s_{an}, m_{an}) \mapsto (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$$
  (oder: $Fahrtzeit = \lambda(s_{ab}, m_{ab}, s_{an}, m_{an})\,.\,(s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$),

  in Pseudocode:

  $$Fahrtzeit = \textbf{function}(s_{ab} : \textbf{nat}, m_{ab} : \textbf{nat}, s_{an} : \textbf{nat}, m_{an} : \textbf{nat})\;\textbf{nat} :$$
  $$(s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$$

  Eine derartige „Funktionsdefinition und -benennung“ heißt ***Funktionsdeklaration*** (***Funktionsvereinbarung***). (Eine Funktion ohne Namen heißt auch ***anonym***.)

- Vereinfachte Notation für Aufruf der (***benannten***) Funktion $Fahrtzeit$:

  $$Fahrtzeit(13, 10, 15, 53).$$

Zusammensetzung von Funktionen

Eine Funktion kann sich auf (eine oder mehrere) andere Funktionen ***abstützen***.
Beispiel ($Fahrtzeit'$: Berechnung der Fahrtzeit in Stunden und Minuten):

$$MinStd = \textbf{function}(m : \textbf{nat})\;\textbf{nat} \times \textbf{nat} :$$
$$\textbf{result}\ \text{Umrechnung von } m \text{ Minuten in } (\text{Stunden}, \text{Minuten})$$
$$(m\ div\ 60, m\ mod\ 60)$$

$$Fahrtzeit = \textbf{function}(s_{ab} : \textbf{nat}, m_{ab} : \textbf{nat}, s_{an} : \textbf{nat}, m_{an} : \textbf{nat})\ \textbf{nat} :$$
$$\textbf{result}\ \text{Fahrtzeit in Minuten}$$
$$(s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$$
$$Fahrtzeit' = \textbf{function}(s_{ab} : \textbf{nat}, m_{ab} : \textbf{nat}, s_{an} : \textbf{nat}, m_{an} : \textbf{nat})\ \textbf{nat} \times \textbf{nat} :$$
$$\textbf{result}\ \text{Fahrtzeit in (Stunden, Minuten)}$$
$$MinStd(Fahrtzeit(s_{ab}, m_{ab}, s_{an}, m_{an}))$$

Funktionale Algorithmen

- Ein Algorithmus im Sinne der funktionalen Programmierung ist eine in einer Funktionsdeklaration benannte, durch Funktionsabstraktion aus einem Term gewonnene Funktion der Art

  $$\textbf{function}(x_1 : typ_1, \ldots, x_n : typ_n)\ typ : t$$

  (die sich auf andere solche Funktionen abstützen kann).
  Dabei bezeichnen $typ_1, \ldots, typ_n, typ$ Typen, $x_1, \ldots, x_n$ Parameter (in diesem Kontext auch ***formale*** Parameter genannt; die Argumente von Funktionsaufrufen heißen dann auch ***aktuelle*** Parameter). Der Term t heißt ***Rumpf*** der Funktion.

- Terme, aus denen Funktionen gewonnen werden, enthalten Anwendungen von ***Basisfunktionen*** und eventuell anderen Funktionen auf Parameter und Daten.

- Basisfunktionen sind vorgegebene Funktionen (z.B. $+, -, *, /, div, mod, \ldots$), die die elementaren Verarbeitungsschritte (vgl. Abschnitt 1.2) repräsentieren.

Konstanten

Zur Systematisierung: Terme ohne Parameter können als (***nullstellige***) Funktionen angesehen werden. Namen solcher Funktionen heißen ***Konstanten***, die entsprechenden Deklarationen ***Elementdeklarationen***. Schreibweise (z.B.):

$$betrag = 543 - 3028.$$

Aufruf:

$$betrag.$$

Lokale Konstanten

In Funktionsrümpfen können auch innerhalb der Funktion deklarierte (***lokale***) Konstanten verwendet werden. Dies geschieht in der Form eines Terms der Art

let ... **in** t **end**,

wobei t ein Term ist und in **let** ... **in** eine endliche Anzahl von Elementdeklarationen stehen.

Beispiel:

$g =$ **function**$(x$: **real**, y_1 : **real**, y_2 : **real**$)$ **real** :
let
$z_1 = y_1 * y_1$
$z_2 = y_2/y_1$
in
$(z_1 + z_2)/x$
end

2.2 Bedingte Terme

Beispiel

Erweiterte Fahrtzeitberechnung: Ohne die bisherige Einschränkung, dass keine Fahrt über Mitternacht hinausgeht. (Verbleibende Einschränkung: Fahrten dauern weniger als 24 Stunden.)

Algorithmus:

$Fahrtzeit'' =$ **function**$(s_{ab}$: **nat**, m_{ab} : **nat**, s_{an} : **nat**, m_{an} : **nat**$)$ **nat** :
pre Fahrtdauer weniger als 24 Stunden
let
$z = (s_{an} - s_{ab}) * 60 + m_{an} - m_{ab}$
in
if $z < 0$ **then** $z + 1440$ **else** z
end

Bedingte Terme

Zur Formulierung von ***Fallunterscheidungen*** werden ***bedingte Terme*** der Form

if b **then** t_1 **else** t_2

als weitere Art von Termen zugelassen. Dabei ist b eine ***Bedingung***, t_1 und t_2 sind Terme. Eine Bedingung kann „wahr“ oder „falsch“ sein. Diese beiden ***Wahrheitswerte*** – dargestellt durch $true$ und $false$ – bilden einen eigenen (Basis-) Typ – bezeichnet durch **bool** – der genau diese beiden Datenelemente enthält.

Vergleichsoperationen

Funktionen wie $<, >, =, \ldots$ (***Vergleichsoperationen***, verwendet in Infixschreibweise) sind typische Basisfunktionen zur Formulierung von (elementaren) Bedingungen.

Beispiel: $< : \textbf{nat} \times \textbf{nat} \to \textbf{bool}$

(ebenso für $\textbf{int} \times \textbf{int} \to \textbf{bool}$ und $\textbf{real} \times \textbf{real} \to \textbf{bool}$)

$$x < y = \begin{cases} true, & \text{falls } x \text{ kleiner als } y \text{ ist} \\ false & \text{sonst.} \end{cases}$$

Zusammengesetzte Bedingungen

- Aus einzelnen Bedingungen können mit folgenden weiteren Basisfunktionen neue komplexere Bedingungen zusammengesetzt werden:

  $\neg : \textbf{bool} \to \textbf{bool}$ (***Negation***, „nicht“),
  $\wedge : \textbf{bool} \times \textbf{bool} \to \textbf{bool}$ (***Konjunktion***, „und“),
  $\vee : \textbf{bool} \times \textbf{bool} \to \textbf{bool}$ (***Disjunktion***, „oder“).

  Diese Funktionen sind durch folgende ***Wahrheitstafeln*** definiert:

x	y	$\neg x$	$x \wedge y$	$x \vee y$
$true$	$true$	$false$	$true$	$true$
$true$	$false$		$false$	$true$
$false$	$true$	$true$	$false$	$true$
$false$	$false$		$false$	$false$

  Bemerkung: Für $\neg, \wedge, \vee$ gelten eine ganze Reihe von Rechengesetzen („Boolesche Algebra“), z.B.:

  $\neg\neg x = x$,
  $x \wedge y = y \wedge x$,
  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$,
  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$,
  $x \wedge true = x$
  usw.

- Beispiel:

  $$zwischen10und20 = \textbf{function}(x : \textbf{int})\,\textbf{bool} : \\ 10 < x \wedge x < 20$$

  $$beispiel = \textbf{function}(x : \textbf{int}, y : \textbf{int})\,\textbf{int} : \\ \textbf{if } zwischen10und20(x) \vee y = 0 \textbf{ then } x \\ \textbf{else if } zwischen10und20(y) \textbf{ then } y \\ \textbf{else } x - y$$

Bemerkung

Bedingungen sind Terme. Sie heißen auch ***Boolesche Ausdrücke***. (Terme, die (ausschließlich) mit $+, -, *, /$ usw. gebildet sind und Zahlen repräsentieren, heißen auch ***arithmetische Ausdrücke***.)

2.3 Rekursive Funktionen

(Mathematische) Konzepte und Begriffe

- ***Induktive Definition*** der Menge $\mathbb{N}_0$
  a) $0 \in \mathbb{N}_0$.
  b) Ist $n \in \mathbb{N}_0$, so ist auch $n + 1 \in \mathbb{N}_0$.
  [c) Außer den Elementen gemäß a) und b) enthält $\mathbb{N}_0$ keine weiteren Elemente.]

  (Die „Regel“ c) ist analog bei allen induktiven Definitionen notwendig und wird meist nicht explizit angegeben.)

  Konstruktives Verständnis der Definition:
  Jedes Element von $\mathbb{N}_0$ (und nur Elemente von $\mathbb{N}_0$) können durch endlich-oftmalige Anwendung der Definitionsregeln „konstruiert“ werden (und die Konstruktion ist zudem eindeutig bestimmt).

- Induktive Definition einer Menge M: allgemeines Schema
  a) Explizite Angabe von Elementen von M.
  b) Regeln zur Erzeugung weiterer Elemente $y \in M$ aus schon vorhandenen $x_1, \ldots, x_k \in M$.

- Beispiel
  Sei A eine Menge. Ein n-Tupel

  $$a = (a_1, \ldots, a_n) \in A^n \quad (= \underbrace{A \times A \times \ldots \times A}_{n})$$

  heißt ***Folge*** (der Länge n, über A). A^* bezeichnet die Menge $\bigcup_{n=0}^{\infty} A^n$, d.h. die Menge aller Folgen beliebiger endlicher Länge (genannt ***endliche Folgen***) über A (einschließlich der ***leeren Folge*** $\varepsilon \in A^0$ mit der Länge 0).

  Die Funktion $:: : A \times A^* \to A^*$ ist definiert durch (in Infixschreibweise)

  $$b :: (a_1, \ldots, a_n) = (b, a_1, \ldots, a_n).$$

  Induktive Definition von A^* (für gegebenes A)
  a) $\varepsilon \in A^*$.
  b) Ist $b \in A$ und $a \in A^*$, so ist $b :: a \in A^*$.

- Induktion und Rekursion

  Induktiv definierte Mengen M ermöglichen:

1. ***Induktionsbeweise*** von Aussagen der Art

   „Für alle $x \in M$ gilt die Eigenschaft $p(x)$“

   mit dem allgemeinen ***Induktionsprinzip*** (Grundform):

   Falls:
   a) Für alle explizit angegebenen Elemente $x \in M$ gilt $p(x)$.
   b) Für beliebige $x_1, \ldots, x_k, y \in M$, y erzeugbar aus $x_1, \ldots, x_k$, gilt: Falls $p(x_1), \ldots, p(x_k)$ gilt, so gilt auch $p(y)$.

   Dann:
   Für alle $x \in M$ gilt $p(x)$.

   (Im Fall der Menge $\mathbb{N}_0$ ist dieses Prinzip die bekannte ***vollständige Induktion***.)

2. (falls „induktive Erzeugung“ eindeutig:) ***rekursive Definitionen*** von Funktionen

   $$f : M \to N$$

   mit dem ***rekursiven Definitionsprinzip*** (Grundform):

   a) Definiere $f(x)$ explizit für alle explizit angegebenen Elemente $x \in M$.
   b) Für jede Regel, die y aus $x_1, \ldots, x_k$ erzeugt, definiere $f(y)$ unter Verwendung von $f(x_1), \ldots, f(x_k)$ (Rückführung, ***Rekursion***).

(Bemerkung: Neben diesen Grundformen gibt es viele weitere Varianten von Induktionsbeweisen und rekursiven Funktionsdefinitionen.)

- Beispiele für rekursive Funktionsdefinitionen

  – Fakultät: $! : \mathbb{N}_0 \to \mathbb{N}_0$ (Informell: $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n$)

  $$0! = 1.$$
  $$(n+1)! = (n+1) \cdot n!\,.$$

  Andere Schreibweisen hierfür:

  $$0! = 1$$
  $$n! = n \cdot (n-1)! \qquad \text{für } n > 0$$

  $$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

  – Fibonacci-Funktion: $\mathit{fib} : \mathbb{N}_0 \to \mathbb{N}_0$

  $$\mathit{fib}(n) = \begin{cases} 1, & \text{falls } n = 0 \text{ oder } n = 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2) & \text{sonst.} \end{cases}$$

  (Variante der Grundform!)

  – Länge einer endlichen Folge über einer Menge A: $l : A^* \to \mathbb{N}_0$

  $$l(x) = \begin{cases} 0, & \text{falls } x = \varepsilon \\ 1 + l(a), & \text{falls } x = b :: a. \end{cases}$$

Algorithmische Bedeutung rekursiv definierter Funktionen

Rekursiv definierte Funktionen können als Algorithmen aufgefasst werden: Die Berechnung von $f(z)$ für ein Argument z geschieht schrittweise gemäß den Rückführungen in der Funktionsdefinition, bis man eine explizite Definition anwenden kann. (Diese Schritte spiegeln (in der Grundform) den induktiven Aufbau von z gemäß der induktiven Definition der Menge wider.)

Beispiel:
$$\begin{aligned} 4! &= 4 \cdot 3! \\ &= 4 \cdot (3 \cdot 2!) \\ &= 4 \cdot (3 \cdot (2 \cdot 1!)) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24. \end{aligned}$$

Rekursive Funktionen

Weitere Art von Algorithmen: Funktionen, die in ihrem Rumpf (mindestens) einen Aufruf von sich selbst enthalten (***rekursive Funktionen***).

Beispiele:

- Fakultät:
$$\begin{aligned} fak = \textbf{function}&(n : \textbf{nat})\ \textbf{nat} : \\ &\textbf{if } n = 0 \textbf{ then } 1 \\ &\textbf{else } n * fak(n-1) \end{aligned}$$

- Fibonacci-Funktion:
$$\begin{aligned} fib = \textbf{function}&(n : \textbf{nat})\ \textbf{nat} : \\ &\textbf{if } n = 0 \vee n = 1 \textbf{ then } 1 \\ &\textbf{else } fib(n-1) + fib(n-2) \end{aligned}$$

- Exponentiation: $exp : \textbf{int} \times \textbf{nat} \to \textbf{int}$, $exp(x, n) = x^n$
$$\begin{aligned} exp = \textbf{function}&(x : \textbf{int}, n : \textbf{nat})\ \textbf{int} : \\ &\textbf{if } n = 0 \textbf{ then } 1 \\ &\textbf{else } x * exp(x, n-1) \end{aligned}$$

- Ackermann-Funktion:
$$\begin{aligned} ack = \textbf{function}&(m : \textbf{nat}, n : \textbf{nat})\ \textbf{nat} : \\ &\textbf{if } m = 0 \textbf{ then } n + 1 \\ &\textbf{else if } n = 0 \textbf{ then } ack(m-1, 1) \\ &\textbf{else } ack(m-1, ack(m, n-1)) \end{aligned}$$

Verschränkt rekursive Funktionen

Funktionen, die sich in ihrem Rumpf gegenseitig aufrufen, heißen ***verschränkt rekursiv***.

Beispiel: Test, ob eine gegebene natürliche Zahl gerade ist:

$$gerade(n) = \begin{cases} true, & \text{falls } n \text{ gerade} \\ false & \text{sonst.} \end{cases}$$

Algorithmus:

$$
\begin{aligned}
gerade = \ & \textbf{function}(n : \textbf{nat})\ \textbf{bool}: \\
& \quad \textbf{if}\ n = 0\ \textbf{then}\ true \\
& \quad \textbf{else}\ ungerade(n-1) \\
ungerade = \ & \textbf{function}(n : \textbf{nat})\ \textbf{bool}: \\
& \quad \textbf{if}\ n = 0\ \textbf{then}\ false \\
& \quad \textbf{else}\ gerade(n-1)
\end{aligned}
$$

Die Technik der Einbettung

Für viele konkrete Aufgaben: Keine „direkte“ Lösung in Form einer rekursiven Funktion angebbar; mögliche Lösung durch ***Einbettung*** in eine allgemeinere Aufgabe.

Beispiel: Für $n \in \mathbb{N}_0$ soll bestimmt werden, ob n Primzahl ist, d.h. ob gilt:

$n > 1$ und
n ist durch keine Zahl $i \in \mathbb{N}_0$ mit $2 \leq i < n$ teilbar. (1)

Kein direkter rekursiver Ansatz für (1), Verallgemeinerung:

(2) Bestimme, ob n durch keine Zahl $i \in \mathbb{N}_0$ mit $k \leq i < n$ teilbar ist (wobei $2 \leq k \leq n$).

(2) lässt sich leicht rekursiv lösen, und (1) wird durch (2) für $k = 2$ gelöst:

$$
\begin{aligned}
keineteiler = \ & \textbf{function}(n : \textbf{nat}, k : \textbf{nat})\ \textbf{bool}: \\
& \quad \textbf{if}\ k = n\ \textbf{then}\ true \\
& \quad \textbf{else}\ n\ mod\ k \neq 0 \wedge keineteiler(n, k+1) \\
istprim = \ & \textbf{function}(n : \textbf{nat})\ \textbf{bool}: \\
& \quad n > 1 \wedge keineteiler(n, 2)
\end{aligned}
$$

Bemerkung

Zu einer Aufgabe kann es „grundsätzlich verschiedene“ rekursive Lösungsideen geben.

Beispiel Exponentiation:

$$
\begin{aligned}
exp' = \ & \textbf{function}(x : \textbf{int}, n : \textbf{nat})\ \textbf{int}: \\
& \quad \textbf{if}\ n = 0\ \textbf{then}\ 1 \\
& \quad \textbf{else if}\ n\ mod\ 2 = 0\ \textbf{then}\ exp'(x * x, n\ div\ 2) \\
& \qquad\qquad \textbf{else}\ x * exp'(x * x, , n\ div\ 2)
\end{aligned}
$$

Terminierung und Korrektheit rekursiver Funktionen

- Konzept der Rekursion: Sehr mächtig, aber auch mit „Gefahren“.
- ***Terminierungsproblem***
  Terminiert eine rekursive Funktion wie gewünscht (d.h. für alle Argumente aus dem beabsichtigten Definitionsbereich)?

- ***Korrektheitsproblem*** (allgemeiner)
  Löst die rekursive Funktion die gestellte Aufgabe?
- Klärung solcher Fragen: Durch formale Beweise möglich und eventuell angebracht (***Programmverifikation***). Grundlegendes Beweismittel: Induktion. (Beispiele: siehe später.)

2.4 Polymorphe Funktionen

Polymorphe Typen

In Funktionen zugelassen: Typbezeichnungen mit ***Typvariablen*** (stehen für beliebige Typen).
Pseudocode-Bezeichnungen: $\alpha, \beta, \gamma, \ldots$.

Eine Typbezeichnung (kurz: ein Typ) heißt ***polymorph*** (***Polytyp***), wenn er Typvariablen enthält.

Beispiele: α,
$\beta \times \gamma \times$ **real**.

Ein nicht polymorpher Typ heißt ***monomorph*** (***Monotyp***).

Polymorphe Funktionen

Eine Funktion $f : typ \to typ'$ mit polymorphen Typen typ und/oder typ' heißt ***polymorph***. f kann aufgerufen werden mit Argumenten von allen Monotypen, die aus typ entstehen, wenn die in typ enthaltenen Typvariablen (konsistent) durch monomorphe Typen ersetzt werden. Das entsprechende Resultat ist von dem Typ, der durch die gleiche Ersetzung aus typ' entsteht.

Beispiel:

$tausch =$ **function**$(x : \alpha, y : \beta)\ \beta \times \alpha : (y, x)$

Aufrufe:	$tausch(7, 13)$	(liefert $(13, 7)$)
	$tausch(true, -13)$	(liefert $(-13, true)$)
	$tausch(3.14, false)$	(liefert $(false, 3.14)$)
	usw.	

2.5 Funktionen höherer Ordnung

Begriffsbestimmungen

- Zusammensetzung von Typen: Sind typ und typ' Typen, so ist $typ \to typ'$ (Menge der Funktionen von typ in typ') ein (zusammengesetzter) Typ.
- Typen der Art $typ \to typ'$ heißen ***Funktionstypen***. Typen, die nicht Funktionstypen sind, nennen wir ***Datentypen***.
- Eine Funktion vom Typ $typ \to typ'$, wo typ und/oder typ' selbst Funktionstypen oder unter Verwendung von Funktionstypen zusammengesetzt sind, heißt ***Funktion höherer Ordnung*** (***Funktional***).

Funktionen als Resultate

- Schreibweise: $typ \to typ' \to typ''$ für $typ \to (typ' \to typ'')$
  (und analog für mehr als 3 Typen).
- Jede Funktion

  $$f_1 \in typ_1 \times typ_2 \times \ldots \times typ_n \to typ$$

  lässt sich auch als Funktion

  $$f_2 \in typ_1 \to typ_2 \to \ldots \to typ_n \to typ$$

  darstellen (***Currying***) und umgekehrt (***Uncurrying***). f_2 heißt ***curried***, genauer: die ***curried Version von*** f_1. Umgekehrt heißt f_1 die ***uncurried Version von*** f_2.
- Beispiel: Addition zweier ganzer Zahlen $(x, y \mapsto x + y)$.
  Uncurried (**int** $\times$ **int** $\to$ **int**):

  $$plus = \textbf{function}(x : \textbf{int}, y : \textbf{int})\ \textbf{int} : x + y$$

  Aufruf (z.B.): $plus(3, 5)$.
  Curried (**int** $\to$ **int** $\to$ **int**):

  $$add = \textbf{function}(x : \textbf{int})\ \textbf{int} \to \textbf{int} : \textbf{function}(y : \textbf{int})\ \textbf{int} : x + y$$

  Aufruf (z.B.): $add(3)(5)$.

Funktionen als Argumente

- Anwendungsbeispiel:
  Ist f eine differenzierbare mathematische Funktion $\mathbb{R} \to \mathbb{R}$, so ist der Differenzenquotient

  $$\frac{f(x + \Delta) - f(x)}{\Delta}$$

für kleines Δ ein guter Näherungswert für den Wert $f'(x)$ der Ableitung f' von f für das Argument x.

Berechnung des Differenzenquotienten für beliebiges f, x und Δ:

$$\mathit{diffquot} = \textbf{function}(f : \textbf{real} \to \textbf{real}, x : \textbf{real}, \mathit{delta} : \textbf{real})\ \textbf{real} : \\ (f(x + \mathit{delta}) - f(x))/\mathit{delta}$$

Möglicher Aufruf:

$$\mathit{diffquot}(\textbf{function}(x : \textbf{real})\ \textbf{real} : x * x, 5.0, 0.001)$$

Alternativ:

$$\mathit{quadrat} = \textbf{function}(x : \textbf{real})\ \textbf{real} : x * x \\ \mathit{diffquot}(\mathit{quadrat}, 5.0, 001)$$

- Allgemein: Mit der Möglichkeit, Funktionen als Eingaben zuzulassen, werden Funktionen höherer Ordnung zu einem sehr mächtigen Konzept.

Beispiele (Grundalgorithmen)

1. Funktionskomposition

$$f : A \to B,\ g : C \to A \quad \mapsto \quad f \circ g : C \to B, \\ f \circ g : x \mapsto f(g(x)).$$

   $\circ$ als Funktion höherer Ordnung:

$$\circ : (A \to B) \times (C \to A) \to C \to B.$$

   Algorithmus für $\circ$:

$$\mathit{comp} = \textbf{function}(f : \boldsymbol{\alpha} \to \boldsymbol{\beta}, g : \boldsymbol{\gamma} \to \boldsymbol{\alpha})\ \boldsymbol{\gamma} \to \boldsymbol{\beta} : \\ \textbf{function}(x : \boldsymbol{\gamma})\ \boldsymbol{\beta} : f(g(x))$$

   Anwendungsbeispiel (vgl. Abschnitt 2.1):

$$\mathit{Fahrtzeit}' = \mathit{comp}(\mathit{MinStd}, \mathit{Fahrtzeit})$$

2. n-fache Iteration einer Funktion

$$f : A \to A,\ n \in \mathbb{N}_0 \quad \mapsto \quad f^n : A \to A, \\ f^n : x \mapsto \underbrace{f(f(\ldots(f}_{n}(x))\ldots)).$$

   Algorithmus:

$$\mathit{iteriert} = \textbf{function}(f : \boldsymbol{\alpha} \to \boldsymbol{\alpha}, n : \textbf{nat})\ \boldsymbol{\alpha} \to \boldsymbol{\alpha} : \\ \textbf{function}(x : \boldsymbol{\alpha})\ \boldsymbol{\alpha} : \\ \textbf{if } n = 0 \textbf{ then } x \\ \textbf{else } f(\mathit{iteriert}(f, n - 1)(x))$$

Anwendungsbeispiel:

$$zweihochx = \textbf{function}(x : \textbf{nat})\ \textbf{nat} :\\ \quad \textbf{result}\ 2^x\\ \quad iteriert(\textbf{function}(y : \textbf{nat})\ \textbf{nat} : 2 * y, x)(1)$$

Funktionen, Konstanten, Terme, Werte

- Algorithmus ist gegeben durch Deklarationen der Art

  $$f = \textbf{function}(x_1 : typ_1, \ldots, x_n : typ_n)\ typ : t, \qquad (1)$$
  $$a = t'. \qquad (2)$$

  Bisherige Sichtweise: (2) aufgefasst als spezielle Funktionsdeklaration.

- Im Lichte der Funktionen höherer Ordnung:

  $$\textbf{function}(x_1 : typ_1, \ldots, x_n : typ_n)\ typ : t$$

  auch auffassbar als Term d.h.: (1) auffassbar als Elementdeklaration im Stile von (2).

- Vereinheitlichende Sprechweise (angelehnt an SML): (1) und (2) sind ***Wertdeklarationen***; an die betreffenden Namen sind die von den Termen gelieferten Werte gebunden.

  (Der Wert des Terms $\textbf{function}(x_1 : typ_1, \ldots, x_n : typ_n)\ typ : t$ ist die (ebenso bezeichnete) Funktion $(x_1, \ldots, x_n) \mapsto t$ als Datenelement von $typ_1 \times \ldots \times typ_n \to typ$.)

Kapitel 3
Funktionale Programmierung in SML

3.1 SML-Programme

SML-Sitzungen

- SML ist in einem ***interaktiven*** Programmiersystem realisiert: In einer ***Sitzung*** können – nach „Aufforderung“ durch das (***Prompt***)- Zeichen „-“ – sowohl (beliebig viele) Algorithmen (als Wertdeklarationen eingeleitet mit dem Schlüsselwort **val**) als auch deren Ausführungen (Funktionsaufrufe) formuliert werden.

- Wertdeklarationen werden von Aufrufen (ebenso wie mehrere Aufrufe voneinander) durch die Eingabe eines Strichpunkts getrennt. Nach jeder solchen Eingabe bestimmt das System die betreffenden Werte (in der Reihenfolge der Aufschreibung) und zeigt diese an. Danach erscheint wieder „-“, und die nächste Eingabe kann getätigt werden usw. (Mehrere Wertdeklarationen können selbst auch durch Strichpunkte voneinander getrennt werden.)

- Beispiel-Sitzung:

```
- val a = 18.375
  val aquadrat = a*a
  val b = ~0.31/a
  val f = fn(x:real):real => (aquadrat + b)/x;
val a = 18.375 : real
val aquadrat = 337.640625 : real
val b = ~0.0168707482993 : real
val f = fn : real -> real
- f 2.0           (* Berechnung von f(2) *);
val it = 168.811877126 : real
- f(4.0)      (* Berechnung von f(4) *);
val it = 84.4059385629 : real
```

  (SML-Sitzungen und -Programme notieren wir in `Schreibmaschinenschrift`, die „Antworten“ schreiben wir zur besseren Lesbarkeit *kursiv*.)

- Die SML-Notation ist im wesentlichen so wie der Pseudocode von Kapitel 2, mit nachfolgend beschriebenen Ausnahmen und Besonderheiten.

Besonderheiten von SML-Darstellungen

- „~“ bezeichnet das „einstellige Minus“. Die Booleschen Basisfunktionen $\neg$, $\wedge$ und $\vee$ werden mit **not**, **andalso** und **orelse** bezeichnet.

- Die Deklaration einer Funktion geschieht in der im Beispiel angegebenen Form (insbesondere mit dem Schlüsselwort **fn** statt **function**), bei rekursiven Funktionen in der Form

  val rec f = **fn** $x \Rightarrow \ldots$

  Die Typangaben können (meistens) auch weggelassen werden (Ausnahmen: s.u.). Das System erkennt die Typen „automatisch“ (aus den Datendarstellungen) und gibt sie mit der Wertangabe aus. Außerdem kann als andere Schreibweise auch

  fun $f(x) = \ldots$

  gewählt werden (auch bei rekursiven Funktionen), z.B.:

  ```
  - fun f(x) = (aquadrat + b)/x;
  val f = fn : real -> real
  ```

  (Wir nennen jetzt diese im Folgenden bevorzugte Schreibweise ***Funktionsdeklaration***.)

  Den Wert liefert das System in jedem Fall in der im Beispiel illustrierten Form (mit dem Typ der Funktion nach dem Doppelpunkt).

- Zeichen wie $+$ und $*$ (und andere) bezeichnen Basisfunktionen für Daten verschiedener Typen, etwa **int** und **real**. In Deklarationen von Funktionen, deren Typ wegen dieser ***Überladung*** der Zeichen (mit jeweils mehr als einer Bedeutung) nicht eindeutig bestimmt werden könnte, ist der jeweilige Anwendungsfall durch eine (zumindest teilweise beibehaltene) Typangabe (***Typ-Einschränkung***) zu bestimmen. Beispiel:

  ```
  - fun quadrat(x) = x*x;
  ?
  - fun quadrat(x:real) = x*x;
  val quadrat = fn : real -> real
  ```

- Funktionsaufrufe können in Funktions- oder Präfixschreibweise geschrieben werden. Gleiches gilt auch für die Parameterangabe bei der Deklaration einer Funktion, z.B.:

  ```
  fun f x = (aquadrat + b)/x
  ```

- Der Wert eines Funktionsaufrufs wird ebenfalls mit seinem Typ angegeben und außerdem mit einem ausgezeichneten Namen it versehen.

- Außer einzelnen Funktionsaufrufen können auch allgemeine Terme eingegeben werden. Ihre Werte werden ebenfalls mit it gekennzeichnet, z.B. (in obigem Kontext):

  ```
  - if a > 20.0 then f(4.0)+b else 1.0;
  val it = 1.0 : real
  ```

- Verschränkt rekursive Funktionen werden ***simultan*** deklariert in der Form

  fun $f1 \ldots$ **and** $f2 \ldots$ **and** $f3 \ldots$

- Namen (***Identifikatoren***, in SML-Sprechweise auch: ***(funktionale) Variablen***) bestehen aus Buchstaben, Ziffern, Hochkommas ($'$) und Unterstrichen (_) und beginnen mit einem Buchstaben (***alphanumerische*** Identifikatoren), oder sie sind aus den Zeichen

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

  zusammengesetzt (***symbolische*** Identifikatoren). Ausgeschlossen sind die folgenden als ***Schlüsselwörter*** reservierten Zusammensetzungen:

  ```
  abstype and andalso as case datatype do else end eqtype
  exception fn fun functor handle if in include infix infixr
  let local nonfix of op open orelse raise rec sharing sig
  signature struct structure then type val while with withtype
  ```

  sowie die Zeichen(kombinationen)

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

- Typvariablen werden durch $'a, 'b$ usw. bezeichnet. Kartesische Produkte werden mit $*$ (für $\times$) gebildet. Beispiel:

  ```
  - fun iteriert (f,n) = fn x => if n=0 then x
                                 else f(iteriert(f,n-1)(x));
  val iteriert = fn : ('a -> 'a) * int -> 'a -> 'a
  ```

  (Beachte: Polymorphie (gleicher Algorithmus für Argumente verschiedener Typen) und Überladung (gleiche Bezeichnung für verschiedene Algorithmen) sind verschiedene Konzepte!)

- Curried Funktionen der Form **fun** $f\ x = \textbf{fn}\ y \Rightarrow \textbf{fn}\ z \Rightarrow \ldots$ können auch noch in der kompakteren Schreibweise **fun** $f\ x\ y\ z\ \ldots = \ldots$ deklariert werden. Beispiel:

  ```
  - fun iteriert (f,n) x = if n=0 then x
                           else f(iteriert(f,n-1)(x));
  val iteriert = fn : ('a -> 'a) * int -> 'a -> 'a
  ```

- Es gibt keine speziellen Schlüsselwörter wie **pre** und **result**. Statt dessen können jedoch allgemeiner an beliebiger Stelle in $(*\ldots*)$ eingeschlossene ***Kommentare*** angegeben werden. Sie werden vom System „nicht beachtet“.

Bemerkung

Wir beschränken uns im Folgenden immer darauf, dass Namen in Wertdeklarationen nicht mehrfach deklariert werden. Dies wäre (in SML zwar möglich, aber) kein rein funktionales Konzept. Ebenso verwenden wir den Namen it nicht in Termen.

3.2 Basistypen und Basisfunktionen

Basistypen von SML

Typ(bezeichnung)	Datenmenge	Datendarstellung
int	Ganze Zahlen	...,~3,~2,~1,0,1,2,3, ...
real	***Gleitpunktzahlen***	(z.B.:) 17.82, 2.3E~1, ~0.5
bool	Wahrheitswerte	$true$, $false$
char	Zeichen (Characters)	(z.B:) #"a", #"b", #"!"
string	Texte (Strings)	(z.B:) "Hallo", "ich bin", ""

Bemerkungen

- Es gibt keinen Datentyp **nat**. Natürliche Zahlen werden als Datenelemente des Typs **int** verstanden. Beispiel:

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

- Bedeutung von 2.3E~1: $2.3 \cdot 10^{-1}$.
- **real** umfasst eine Teilmenge der Menge $\mathbb{R}$ (endliche Dezimalbrüche).
  In der Praxis weitere Einschränkung (bei **int** und **real**): Nur Darstellungen gewisser maximaler Stellenzahlen möglich (durch jeweiligen Computer bestimmt).
- Zeichen sind Elemente einer (für Computer-Anwendungen normierten) Zeichenmenge (dem ***ASCII-Zeichensatz***) und werden in #" und " eingeschlossen dargestellt.
- Texte sind aus beliebigen Zeichen zusammengesetzt und werden in " eingeschlossen dargestellt.
- Die angegebenen Datendarstellungen können als „Namen“ für Werte und damit als (vorgegebene, ***spezielle***) Konstanten im Sinne der Abschnitte 2.1 und 2.5 angesehen werden.

Der ASCII-Zeichensatz

Der ASCII-Zeichensatz enthält folgende 128 Zeichen (mit zugehörigen ***Codenummern***):

Zeichen	Code		Zeichen	Code		Zeichen	Code		Zeichen	Code
NUL	0		␣	32		@	64		‘	96
SOH	1		!	33		A	65		a	97
STX	2		"	34		B	66		b	98
ETX	3		#	35		C	67		c	99
EOT	4		$	36		D	68		d	100
ENQ	5		%	37		E	69		e	101
ACK	6		&	38		F	70		f	102
BEL	7		’	39		G	71		g	103
BS	8		(	40		H	72		h	104
HT	9		)	41		I	73		i	105
LF	10		*	42		J	74		j	106
VT	11		+	43		K	75		k	107
FF	12		,	44		L	76		l	108
CR	13		-	45		M	77		m	109
SO	14		.	46		N	78		n	110
SI	15		/	47		O	79		o	111
DLE	16		0	48		P	80		p	112
DC1	17		1	49		Q	81		q	113
DC2	18		2	50		R	82		r	114
DC3	19		3	51		S	83		s	115
DC4	20		4	52		T	84		t	116
NAK	21		5	53		U	85		u	117
SYN	22		6	54		V	86		v	118
ETB	23		7	55		W	87		w	119
CAN	24		8	56		X	88		x	120
EM	25		9	57		Y	89		y	121
SUB	26		:	58		Z	90		z	122
ESC	27		;	59		[	91		{	123
FS	28		<	60		\	92		|	124
GS	29		=	61		]	93		}	125
RS	30		>	62		^	94		~	126
US	31		?	63		_	95		DEL	127

Die Zeichen mit den Codenummern 0 - 32 und 127 sind ***Steuerzeichen*** (z.B. für „backspace“, „return“, „Zwischenraum“ usw.). Die übrigen Zeichen heißen ***druckbar***.

Basisfunktionen (für Daten von Basistypen)

Basisfunktionen	Typen	Bemerkungen
Arithmetische Operationen:		
$+, -, *$	**int** $*$ **int** $\rightarrow$ **int**	
	real $*$ **real** $\rightarrow$ **real**	
~	**int** $\rightarrow$ **int**	
	real $\rightarrow$ **real**	
$/$	**real** $*$ **real** $\rightarrow$ **real**	
div	**int** $*$ **int** $\rightarrow$ **int**	
mod	**int** $*$ **int** $\rightarrow$ **int**	
Vergleichsoperationen:		
$=, <>$	**int** $*$ **int** $\rightarrow$ **bool**	$<>$: „ungleich“;
	bool $*$ **bool** $\rightarrow$ **bool**	$=, <>$: nicht für **real**!
	char $*$ **char** $\rightarrow$ **bool**	
	string $*$ **string** $\rightarrow$ **bool**	
$<, <=, >, >=$	**int** $*$ **int** $\rightarrow$ **bool**	$<=$: „kleiner-gleich“;
	real $*$ **real** $\rightarrow$ **bool**	$>=$: „größer-gleich“;
	char $*$ **char** $\rightarrow$ **bool**	
	string $*$ **string** $\rightarrow$ **bool**	
Boolesche Operationen:		
not	**bool** $\rightarrow$ **bool**	
andalso, **orelse**	**bool** $*$ **bool** $\rightarrow$ **bool**	
Textkonkatenation:		
^	**string** $*$ **string** $\rightarrow$ **string**	

Lexikografische Ordnung

- Die Vergleichsoperationen auf **char** beziehen sich auf die durch die Codezahlen des ASCII-Zeichensatzes gegebene Ordnung der Zeichen.
- Die Vergleichsoperationen auf **string** beziehen sich auf die ***lexikografische Ordnung*** von Texten (basierend auf der Ordnung der Zeichen). Diese ist für zwei Texte "$x_1 \ldots x_n$" und "$y_1 \ldots y_m$" (mit Zeichen $x_1, \ldots, x_n, y_1, \ldots, y_m$ des ASCII-Zeichensatzes, $n, m \geq 0$) wie folgt definiert:

  "$x_1 \ldots x_n$" $<=$ "$y_1 \ldots y_m$" genau dann, wenn
  - entweder: $n \leq m$ und $x_i = y_i$ für alle $i = 1, \ldots, n$
  - oder: es gibt ein $k \geq 1$ mit $k \leq n$ und $k \leq m$, so dass $x_i = y_i$ für alle $i = 1, \ldots, k-1$ gilt und die Codezahl von x_k kleiner als die Codezahl von y_k ist.

Standardfunktionen

Die meisten der angegebenen Basisfunktionen sind tatsächlich durch die Sprachdefinition vorgegeben. Darüber hinaus sind in jedem SML-System noch eine Reihe weiterer Funktionen als ***Standardfunktionen*** „vordefiniert“. Einige davon sind obligatorisch, bei anderen

kann es von System zu System Unterschiede geben. Einige der genannten Basisfunktionen sind solche Standardfunktionen. Außerdem gehören dazu eine Reihe von ***Typkonvertierungs***-Funktionen, z.B.:

Funktionen	Typ	Bemerkungen
$real$	**int** $\rightarrow$ **real**	ganze als reelle Zahl
$floor$	**real** $\rightarrow$ **int**	reelle als ganze Zahl abrunden
$ceil$	**real** $\rightarrow$ **int**	reelle als ganze Zahl aufrunden
ord	**char** $\rightarrow$ **int**	ASCII-Codezahl des Zeichens
chr	**int** $\rightarrow$ **char**	Zeichen gemäß ASCII-Codezahl
str	**char** $\rightarrow$ **string**	Zeichen als Text

Bemerkung

- Alle angegebenen Basisfunktionen sind total (wenn man von Größenbeschränkungen absieht) mit Ausnahme von $/, div, mod$.
- Alle Basisfunktionen sind von einem Typ $typ \rightarrow typ$ oder $typ \times typ \rightarrow typ'$. Letztere werden in Infixschreibweise angewendet. Ihre Bezeichnungen ($+, -$ usw.) heißen auch ***Infix-Operatoren***.

3.3 Syntaxdefinitionen

Syntax und Semantik

Programmiersprache: „Formal beschriebene Sprache“ zur Darstellung von Algorithmen (einschließlich der auftretenden Daten). Das erfordert:

- Präzise Festlegung der „textuellen Gestalt“ eines Programms (***Syntax*** der Sprache),
- präzise Festlegung der Bedeutung und Wirkungsweise eines („syntaktisch korrekten“) Programms (***Semantik*** der Sprache).

Definitionen

- Ein ***Alphabet*** ist eine endliche Menge, deren Elemente (in diesem Kontext) ***Zeichen*** (***Symbole***) genannt werden.
- Eine ***Zeichenreihe*** (***Zeichenkette***) über einem Alphabet Σ ist eine endliche Folge von Elementen $\sigma_1, \ldots, \sigma_n$ von Σ, $n \in \mathbb{N}_0$. (Schreibweise hier: $\sigma_1\sigma_2 \ldots \sigma_n$.) ε heißt (in diesem Kontext) ***leere Zeichenreihe***.
- Eine ***formale Sprache*** über einem Alphabet Σ ist eine Teilmenge der Menge Σ^* aller Zeichenreihen über Σ.

Grundprinzipien für Programmiersprachen

- Eine Programmiersprache ist (hinsichtlich ihrer Syntax) eine formale Sprache (über einem geeigneten Alphabet).
- Zur Festlegung der Syntax (***Syntaxdefinition***) einer Programmiersprache muss das betreffende Alphabet festgelegt werden, und es muss angegeben werden, welche Zeichenreihen ***syntaktisch korrekte*** Programme sind (d.h. zur Sprache gehören). Letzteres geschieht (größtenteils) durch formal formulierte Regeln.

Beispiel: Alphanumerische Identifikatoren in SML

Bildungsregeln:

(a) Jeder alphanumerische Identifikator (*anId*) ist ein Buchstabe (*Buchst*), gefolgt von einer (eventuell leeren) endlichen Folge von Zeichen, die jeweils ein Buchstabe, eine Ziffer (*Ziffer*), ein Zeichen `'` oder ein Zeichen `_` sein können.

(b) Ein Buchstabe ist ein Zeichen `A` oder `B` oder `C` oder ... oder `y` oder `z`.

(c) Eine Ziffer ist ein Zeichen `0` oder `1` oder ... oder `9`.

Formalere (***BNF-***) Notation:

(a′) $\langle \textit{anId} \rangle ::= \langle \textit{Buchst} \rangle \left\{ \langle \textit{Buchst} \rangle \mid \langle \textit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\}^*$

(b′) $\langle \textit{Buchst} \rangle ::= \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \mid \texttt{N} \mid \texttt{O} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid$
$\texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid$
$\texttt{k} \mid \texttt{l} \mid \texttt{m} \mid \texttt{n} \mid \texttt{o} \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z}$

(c′) $\langle \textit{Ziffer} \rangle ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$

BNF (Backus-Naur-Form)

- Häufig verwendete Notation zur Formulierung der „Syntaxregeln“ für Programmiersprachen.
- Neben dem zugrunde liegenden Alphabet Σ der Sprache wird ein weiteres Alphabet V verwendet (Menge der ***Nicht-Terminalzeichen***; die Elemente von Σ heißen in diesem Zusammenhang auch ***Terminalzeichen***).
- V enthält ein ausgezeichnetes Element S (***Startzeichen***).
- Die Syntaxregeln werden gegeben durch ***Produktionsregeln*** der Form

  $$A ::= \beta$$

  wobei $A \in V$ und β eine ***BNF-Satzform*** ist.
- BNF-Satzformen sind (induktiv) definiert wie folgt:

(1) Jede BNF-Satzform hat die Gestalt $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ $(n \geq 1)$.

(2) Jedes β_i $(i = 1, \ldots, n)$ hat die Gestalt $\gamma_{i_1}\gamma_{i_2}\ldots\gamma_{i_{m_i}}$ $(m_i \geq 1)$.

(3) Jedes γ_{i_j} $(i = 1, \ldots, n, j = 1, \ldots, m_i)$ ist entweder ein Element aus $\Sigma \cup V$ oder hat eine der vier Gestalten

$$\Big\{\delta\Big\},\ \Big\{\delta\Big\}^*,\ \Big\{\delta\Big\}^+,\ \Big[\delta\Big]$$

mit einer BNF-Satzform δ.

- Σ, V, S und die Menge der Produktionsregeln bilden eine ***BNF-Grammatik***.

Ableitungen in einer BNF-Grammatik

Ist $B \in V$ und $\sigma \in \Sigma^*$ (für eine gegebene BNF-Grammatik), so heißt σ aus B ***ableitbar***, wenn σ aus B erzeugbar ist durch endlich-oftmalige Nacheinander-Ausführung jeweils einer der folgenden (textuellen) Ersetzungen (dabei sei β eine BNF-Satzform $\beta_1 \mid \ldots \mid \beta_n$):

(1) Ein $A \in V$, für das die Produktionsregel $A ::= \beta$ in der BNF-Grammatik enthalten ist, wird ersetzt durch eines der β_i $(i = 1, \ldots, n)$.

(2) $\Big\{\beta\Big\}$ wird ersetzt durch eines der β_i $(i = 1, \ldots, n)$.

(3) $\Big\{\beta\Big\}^*$ wird ersetzt durch $\Big\{\beta\Big\}\Big\{\beta\Big\}\ldots\Big\{\beta\Big\}$ (mit einer beliebigen Anzahl (auch null) von $\Big\{\beta\Big\}$).

(4) $\Big\{\beta\Big\}^+$ wird ersetzt durch $\Big\{\beta\Big\}\Big\{\beta\Big\}\ldots\Big\{\beta\Big\}$ (mit einer beliebigen Anzahl (mindestens eins) von $\Big\{\beta\Big\}$).

(5) $\Big[\beta\Big]$ wird ersetzt durch ε (d.h. „ersatzlos gestrichen“) oder durch eines der β_i $(i = 1, \ldots, n)$.

Sei $\mathcal{L}(B) = \{\sigma \in \Sigma^* \mid \sigma \text{ aus } B \text{ ableitbar}\}$. Die durch die BNF-Grammatik (mit dem Startzeichen S) definierte Sprache ist dann $\mathcal{L}(S)$. (Allgemeiner kann auch $\mathcal{L}(B)$ für $B \neq S$ als die „durch B repräsentierte formale Sprache“ angesehen werden.)

Beispiel einer Ableitung

Regeln: (a′), (b′), (c′) von oben.
Menge aller syntaktisch korrekten alphanumerischen Identifikatoren: $\mathcal{L}(\langle anId\rangle)$.
Z.B.: `a_1` $\in \mathcal{L}(\langle anId\rangle)$ gemäß folgender Ableitung von `a_1` aus $\langle anId\rangle$:

$$\begin{aligned}
\langle anId\rangle &\rightsquigarrow \langle Buchst\rangle\Big\{\langle Buchst\rangle \mid \langle Ziffer\rangle \mid \texttt{'} \mid \texttt{_}\Big\}^* && \text{mit (1)}\\
&\rightsquigarrow \texttt{a}\Big\{\langle Buchst\rangle \mid \langle Ziffer\rangle \mid \texttt{'} \mid \texttt{_}\Big\}^* && \text{mit (1)}\\
&\rightsquigarrow \texttt{a}\Big\{\langle Buchst\rangle \mid \langle Ziffer\rangle \mid \texttt{'} \mid \texttt{_}\Big\}\Big\{\langle Buchst\rangle \mid \langle Ziffer\rangle \mid \texttt{'} \mid \texttt{_}\Big\} && \text{mit (3)}\\
&\rightsquigarrow \texttt{a_}\Big\{\langle Buchst\rangle \mid \langle Ziffer\rangle \mid \texttt{'} \mid \texttt{_}\Big\} && \text{mit (2)}\\
&\rightsquigarrow \texttt{a_}\langle Ziffer\rangle && \text{mit (2)}\\
&\rightsquigarrow \texttt{a_1} && \text{mit (1)}
\end{aligned}$$

Zusätzliche syntaktische Einschränkungen

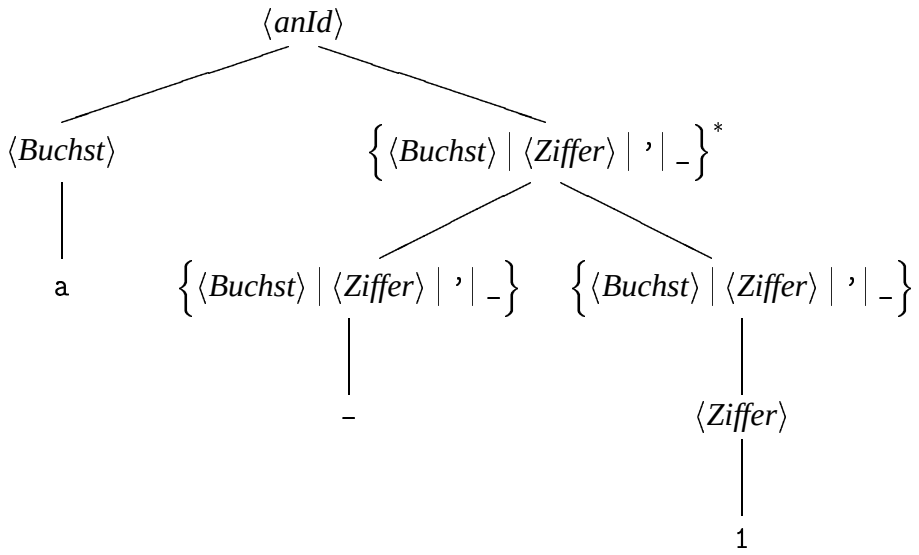
Manche syntaktischen Festlegungen lassen sich nicht (oder nur sehr schwer) innerhalb einer BNF-Grammatik beschreiben. Diese werden durch zusätzliche, verbal formulierte ***Kontextbedingungen*** angegeben.

Komplette Syntaxdefinition im Beispiel:
Alphanumerische Identifikatoren sind definiert durch die BNF-Grammatik für ⟨*anId*⟩ mit der Kontextbedingung:

- Die Schlüsselwörter `abstype` ... `withtype` sind ausgeschlossen.

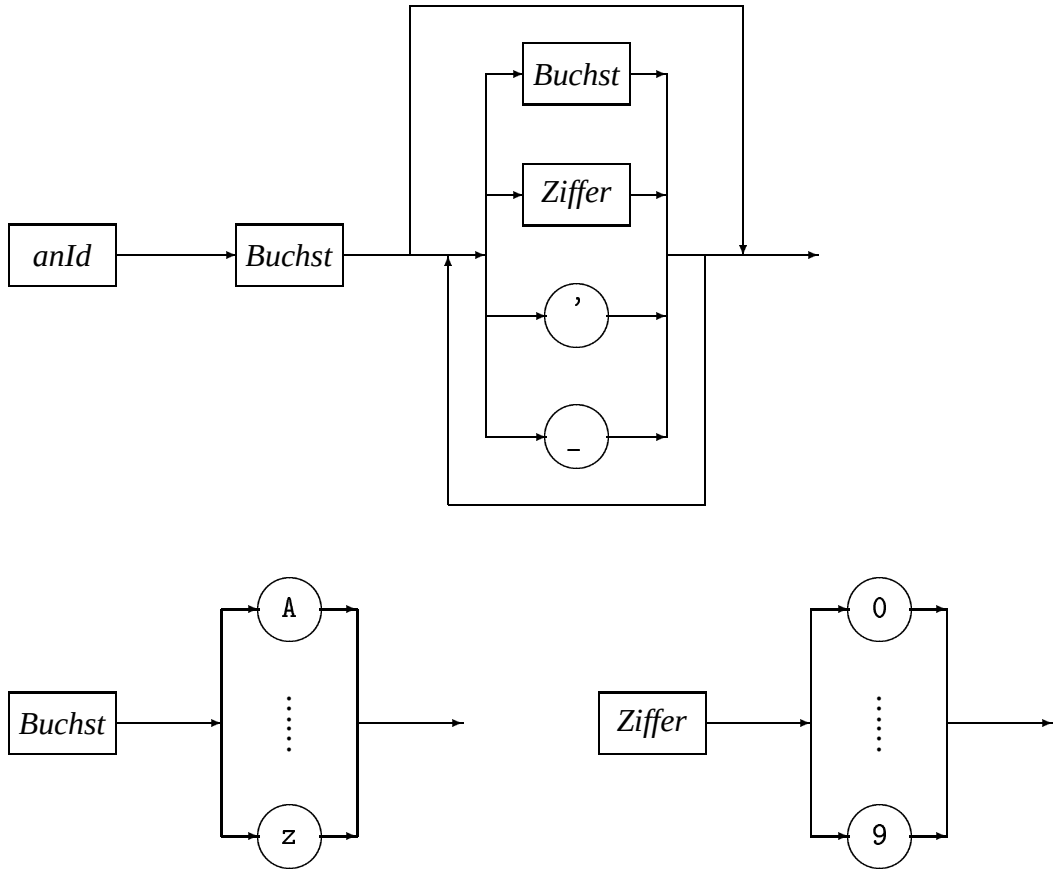
Bemerkung

Ableitungen in BNF-Grammatiken lassen sich grafisch durch ***Ableitungsbäume*** veranschaulichen, z.B. die Ableitung von `a_1` aus ⟨*anId*⟩ durch:



Syntaxdiagramme

BNF-Syntaxdefinitionen werden oft grafisch durch ***Syntaxdiagramme*** dargestellt, die Regeln für alphanumerische Identifikatoren z.B. durch:



Allgemein:

- Terminalzeichen in ◯,
- Für jedes Nicht-Terminalzeichen A (in ▭) ein Diagramm; eine Zeichenreihe aus $\mathcal{L}(A)$ erhält man, indem man in dem Diagramm für A einen beliebigen Weg von ▭ A ▭ entlang der Pfeile bis zum „Ausgang“ wählt, dabei jedes auftretende Terminalzeichen „aufsammelt“ und jedes auftretende Nicht-Terminalzeichen B durch eine entsprechend gefundene Zeichenreihe aus $\mathcal{L}(B)$ ersetzt.

Syntaxdefinition für SML

Im Folgenden: BNF-Grammatik für (Eingaben bei) SML-Sitzungen (gemäß den bisher eingeführten Sprachelementen, ohne die zusätzliche Schreibweise für Funktionsdeklarationen).

Zugrundeliegendes Alphabet: Menge der druckbaren Zeichen des ASCII-Zeichensatzes.

Startzeichen: ⟨*Programm*⟩.

Kontextbedingungen: Nicht angegeben (ergeben sich aus den informellen Angaben in den Abschnitten 3.1 und 3.2).

Produktionsregeln:

⟨*Programm*⟩ ::= $\left\{\left\{\langle \textit{WertDekl}\rangle\right\}^{+}\texttt{;} \mid \langle \textit{Term}\rangle\,\texttt{;}\right\}^{+}$

⟨*WertDekl*⟩ ::= `val` $\left[\texttt{rec}\right]$ ⟨*Name*⟩ `=` ⟨*Term*⟩

⟨*Term*⟩ ::= ⟨*atomTerm*⟩ | ⟨*FunktAbstr*⟩ | ⟨*FunktAnw*⟩ | ⟨*bedTerm*⟩ | ⟨*letTerm*⟩

⟨*atomTerm*⟩ ::= ⟨*spezKonst*⟩ | ⟨*Name*⟩ | `(`⟨*Term*⟩$\left\{\texttt{,}\langle \textit{Term}\rangle\right\}^{*}$`)`

⟨*FunktAbstr*⟩ ::= `fn` $\left\{\langle \textit{forPar}\rangle \mid \texttt{(}\langle \textit{forPar}\rangle\left\{\texttt{,}\langle \textit{forPar}\rangle\right\}^{*}\texttt{)}\right\}\left[\texttt{:}\langle \textit{Typ}\rangle\right]$ `=>` ⟨*Term*⟩

⟨*forPar*⟩ ::= ⟨*Name*⟩$\left[\texttt{:}\langle \textit{Typ}\rangle\right]$

⟨*Typ*⟩ ::= `int` | `real` | `bool` | `string` | ⟨*Typ*⟩`*`⟨*Typ*⟩ | ⟨*Typ*⟩`->`⟨*Typ*⟩ | `(`⟨*Typ*⟩`)`

⟨*FunktAnw*⟩ ::= ⟨*Term*⟩⟨*atomTerm*⟩ | ⟨*Term*⟩⟨*InfixOp*⟩⟨*Term*⟩

⟨*InfixOp*⟩ ::= `+` | `-` | `*` | `/` | `div` | `mod` | `=` | `<>` | `<` | `<=` | `>` | `>=` | `andalso` | `orelse` | `^`

⟨*bedTerm*⟩ ::= `if` ⟨*Term*⟩ `then` ⟨*Term*⟩ `else` ⟨*Term*⟩

⟨*letTerm*⟩ ::= `let` $\left\{\langle \textit{WertDekl}\rangle\right\}^{+}$ `in` ⟨*Term*⟩ `end`

⟨*spezKonst*⟩ ::= ⟨*intKonst*⟩ | ⟨*realKonst*⟩ | ⟨*charKonst*⟩ | ⟨*stringKonst*⟩ | `true` | `false`

⟨*intKonst*⟩ ::= $\left[\texttt{~}\right]\left\{\langle \textit{Ziffer}\rangle\right\}^{+}$

⟨*realKonst*⟩ ::= ⟨*intKonst*⟩`.`$\left\{\langle \textit{Ziffer}\rangle\right\}^{+}$ | ⟨*intKonst*⟩$\left[\texttt{.}\left\{\langle \textit{Ziffer}\rangle\right\}^{+}\right]$`E`⟨*intKonst*⟩

⟨*charKonst*⟩ ::= `#"`⟨*Zeichen*⟩`"`

⟨*stringKonst*⟩ ::= `"`$\left\{\langle \textit{Zeichen}\rangle\right\}^{*}$`"`

⟨*Zeichen*⟩ ::= ⟨*Buchst*⟩ | ⟨*Ziffer*⟩ | ⟨*Symbol*⟩ | ⟨*sonstZeich*⟩ |

⟨*Name*⟩ ::= ⟨*anId*⟩ | ⟨*symbId*⟩

⟨*anId*⟩ ::= ⟨*Buchst*⟩$\left\{\langle \textit{Buchst}\rangle \mid \langle \textit{Ziffer}\rangle \mid \texttt{'} \mid \texttt{\_}\right\}^{*}$

⟨*symbId*⟩ ::= $\left\{\langle \textit{Symbol}\rangle\right\}^{+}$

⟨*Buchst*⟩ ::= `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z` | `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z`

⟨*Ziffer*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

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

⟨*sonstZeich*⟩ ::= `'` | `(` | `)` | `,` | `.` | `;` | `[` | `]` | `_` | `{` | `}`

3.4 Termauswertung

Präzedenz- und Assoziierungsregeln für Infix-Operatoren

- Den Infix-Operatoren werden folgende ***Präzedenzen*** zugeordnet:

Präzedenz	Infix-Operatoren
6	$*$ / *div* *mod*
5	$+$ $-$ ^
4	
3	$=$ <> $<$ <= $>$ >=
2	**andalso**
1	**orelse**

- Operatoren mit höherer Präzedenz ***binden*** stärker als solche mit niedrigerer Präzedenz.
- Bei gleicher Präzedenz wird nach links gebunden (***assoziiert***).
- Beispiel:
```
- 2 * 7 - 3 + 4;
val it = 15 : int
```

Werte und Umgebungen

- Zu jedem Zeitpunkt einer SML-Sitzung ist an jeden bis dahin durch eine nicht-lokale (d.h. nicht innerhalb eines Terms **let** ... **in** auftretende) Wertdeklaration eingeführten Namen a ein Wert w gebunden (geschrieben: <a,w>). Die Menge aller dieser Bindungen heißt (aktuelle) ***Umgebung***.
- Zu Beginn der Sitzung ist die Umgebung leer. Jede nicht-lokale Wertdeklaration der Form **val** $a = t$ erweitert die Umgebung. Dazu wird zum Term t der Wert $W(t)$ in Abhängigkeit der bisherigen Umgebung bestimmt. Ergibt diese ***Auswertung*** keinen Wert ($W(t)$ ***undefiniert***), so liefert das System eine Fehlermeldung. Andernfalls wird der Umgebung die neue Bindung <$a, W(t)$> hinzugefügt.

Auswertung

Der Wert $W(t)$ eines Terms t in einer Umgebung U ist rekursiv gemäß der Syntax von t (einschließlich der Präzedenz- und Assoziierungsregeln) definiert wie folgt:

(1) Ist t eine spezielle Konstante, so ist $W(t)$ das durch t bezeichnete Datenelement.

(2) Ist t ein Name und <t, w> $\in U$, so ist $W(t) = w$.

(3) Ist t von der Form $(t_1, \ldots, t_n)$ und sind $W(t_1), \ldots, W(t_n)$ definiert, so ist $W(t) = (W(t_1), \ldots, W(t_n))$. Andernfalls ist $W(t)$ undefiniert.

(4) Ist t eine Funktionsabstraktion **fn** $(x_1, \ldots, x_n) \Rightarrow t'$, (im Fall $n = 1$ auch ohne Klammern), so ist $W(t)$ die durch t bezeichnete Funktion.

(5) Sei t eine Funktionsanwendung der Form $u(v_1, \ldots, v_n)$ (im Fall $n = 1$ auch ohne Klammern). $W(u)$ sei die Funktion **fn** $(x_1, \ldots, x_n) \Rightarrow t'$, und $W(v_1), \ldots, W(v_n)$ seien die Werte von $v_1, \ldots, v_n$ in U. $W'(t')$ sei der Wert von t' in der Umgebung $U \cup \{<x_1, W(v_1)>, \ldots, <x_n, W(v_n)>\}$. Sind alle diese Werte definiert, so ist $W(t) = W'(t')$. Andernfalls ist $W(t)$ undefiniert.

(6) Sei t eine Funktionsanwendung der Form $\tilde{}\, t_1$ oder **not** t_1 oder t_1 *op* t_2 mit einem von **andalso** und **orelse** verschiedenen Infix-Operator *op* (der eine Basisfunktion $\oplus$ bezeichnet). Sind $W(t_1)$ und $W(t_2)$ definiert, so ist $W(t) = -W(t_1)$ bzw. $W(t) = \neg W(t_1)$ bzw. $W(t) = W(t_1) \oplus W(t_2)$. Andernfalls ist $W(t)$ undefiniert.

(7) Ist t ein let-Term der Form **let val** $a_1 = t_1 \ldots$ **val** $a_n = t_n$ **in** u **end**, so sei:

$$
\begin{aligned}
&w_1 = W(t_1) \text{ in } U \text{ und } U_1 = U \cup \{<a_1, w_1>\},\\
&w_2 = W(t_2) \text{ in } U_1 \text{ und } U_2 = U_1 \cup \{<a_2, w_2>\},\\
&\vdots\\
&w_n = W(t_n) \text{ in } U_{n-1} \text{ und } U_n = U_{n-1} \cup \{<a_n, w_n>\},\\
&w = W(u) \text{ in } U_n.
\end{aligned}
$$

(falls $w_1, \ldots, w_n, w$ definiert). Dann ist $W(t) = w$. Ist einer der Werte undefiniert, so ist $W(t)$ undefiniert.

(8) Sei t ein bedingter Term der Form **if** b **then** t_1 **else** t_2. Ist $W(b) = true$ und $W(t_1)$ definiert, so ist $W(t) = W(t_1)$. Ist $W(b) = false$ und $W(t_2)$ definiert, so ist $W(t) = W(t_2)$. In allen anderen Fällen ist $W(t)$ undefiniert.

(9) $W(t_1$ **andalso** $t_2) = W($**if** t_1 **then** t_2 **else** $false)$;
$W(t_1$ **orelse** $t_2) = W($**if** t_1 **then** $true$ **else** $t_2)$.

Bemerkung

W stellt eine Vereinfachung (nur definiert für die bisher betrachteten Sprachelemente und mit den eingeführten Einschränkungen) des vom SML-System angewendeten Auswertungs-Algorithmus als (rekursive) Funktion dar. In der praktischen Realisierung der Auswertung wird die gegebene Aufschreibung (***konkrete Syntax***) eines Terms „syntaktisch analysiert“ und typischerweise in einer Form dargestellt, in der die syntaktische Struktur einschließlich Präzedenzen und Assoziierungen explizit sichtbar wird (***abstrakte Syntax***).

Beispiel: Konkrete Syntax: $10 + x * 85 - (134 + y)$;
Abstrakte Syntax (vereinfacht): $Diff(Sum(10, Prod(x, 85)), Sum(134, y))$

Strikte und nicht-strikte Auswertung

- Charakteristisch für die Auswertung gemäß W sind die Schritte (5) und (6): Bei einer Funktionsanwendung werden zunächst alle Argumente ausgewertet und die sich ergebenden Werte dann in die betreffende Funktion eingesetzt. (***Wertaufruf***, ***call-by-value***, ***strikte Auswertung***). Dieser Vorgang lässt sich veranschaulichen durch eine Folge von Ersetzungen (***Substitutionsmodell***), z.B. für einen Aufruf $dop(3 + 5)$ mit einer Funktion **fun** $dop\ x = 2 * x$:

$$dop(3+5) \rightsquigarrow dop(8) \rightsquigarrow 2*8 \rightsquigarrow 16.$$

- Eine andere Auswertungs-„Strategie“ ist der ***Namensaufruf*** (***call-by-name***). Dabei wird erst die Funktion auf die Argument-Terme angewendet und dann der sich ergebende Term weiter ausgewertet. Im Substitutionsmodell des Beispiels ergeben sich hierbei die Ersetzungsschritte

  $$dop(3+5) \rightsquigarrow 2*(3+5) \rightsquigarrow 2*8 \rightsquigarrow 16.$$

- Die ***verzögerte Auswertung*** (***lazy evaluation***, ***call-by-need***) arbeitet ebenso wie der Namensaufruf, vermeidet aber möglicherweise auftretende Mehrfach-Auswertungen von Argumenten.

- Ausgenommen vom Wertaufruf sind bedingte Terme und Terme mit den Booleschen Operationen **andalso** und **orelse**. Deren Auswertung gemäß den Regeln (8) und (9) ist ein Spezialfall der verzögerten Auswertung und heißt auch ***Kurzauswertung*** (***short-circuit*** evaluation). Sie bewirkt, dass z.B. der Wert von **if** b **then** t_1 **else** t_2 definiert sein kann, obwohl t_1 oder t_2 undefiniert sind. (Die drei Konstruktionen **if** . . . **then** . . . **else** . . ., **andalso** und **orelse** stellen ***nicht-strikte*** Funktionen dar: Das Ergebnis einer Anwendung kann definiert sein, obwohl einzelne Argumente undefiniert sind. Alle sonstigen Funktionen werden als ***strikt*** aufgefasst: Ist mindestens ein Argument undefiniert, so ist das Ergebnis der Anwendung undefiniert.)

Beispiel (Auswertung des Aufrufs einer rekursiven Funktion)

Sei

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

($W^{n=i}$ bezeichne die Auswertung mit der Bindung <n, i>.)

$$\begin{aligned}
W(fak(3)) &= W^{n=3}(\textbf{if } n=0 \textbf{ then } 1 \textbf{ else } n*fak(n-1)) \\
&= 3*W^{n=3}(fak(n-1)) \\
W^{n=3}(fak(n-1)) &= W^{n=2}(\textbf{if } n=0 \textbf{ then } 1 \textbf{ else } n*fak(n-1)) \\
&= 2*W^{n=2}(fak(n-1)) \\
W^{n=2}(fak(n-1)) &= W^{n=1}(\textbf{if } n=0 \textbf{ then } 1 \textbf{ else } n*fak(n-1)) \\
&= 1*W^{n=1}(fak(n-1)) \\
W^{n=1}(fak(n-1)) &= W^{n=0}(\textbf{if } n=0 \textbf{ then } 1 \textbf{ else } n*fak(n-1)) \\
&= 1 \\
W^{n=2}(fak(n-1)) &= 1*1=1 \\
W^{n=3}(fak(n-1)) &= 2*1=2 \\
W(3) &= 3*2=6
\end{aligned}$$

Terminierung rekursiver Funktionen

Sei f eine rekursive Funktion. Die Terminierung von f für eine Eingabe x bedeutet, dass die Auswertung $W(f(x))$ (in der durch den Kontext gegebenen Umgebung) nach endlich vielen Schritten einen definierten Wert hat.

Beispiele

1.
```
fun fak n = if n=0 then 1 else n*fak(n-1)
```
Behauptung: fak terminiert für jede beliebige Eingabe $k \in \mathbb{N}_0$.

2.
```
fun h(x,y,z) = if x=y+1 then y
               else z*h(x+y+1,2*(y+1),z)
```
Aufrufabfolge von h, z.B.:

$$h(8,1,2) \rightsquigarrow h(10,4,2) \rightsquigarrow h(15,10,2) \rightsquigarrow ?$$

Behauptung: h terminiert für beliebige Eingabe (i,j,k) mit $i,j,k \geq 0$ und $i > j$.

Korrektheit rekursiver Funktionen

Sei f eine rekursive Funktion. Die Korrektheit von f bedeutet, dass die Auswertung $W(f(x))$ für alle Eingaben x des beabsichtigten Definitionsbereichs von f (in der durch den Kontext gegebenen Umgebung) nach endlich vielen Schritten den gewünschten Wert hat.

(Beachte: In dieser Formulierung ist die Terminierung mit eingeschlossen (***totale Korrektheit***). Eine schwächere Eigenschaft ist die ***partielle Korrektheit***: Die Werte, für die die Auswertung nach endlich vielen Schritten zu Ende kommt, müssen wie gewünscht sein.)

Beispiel

```
fun ggt(m,n) = if m=n then m
               else if m>n then ggt(m-n,n)
                    else ggt(m,n-m)
```

Behauptung: Für beliebige Eingabe (i,j) mit $i,j \geq 1$ ist $W(ggt(i,j))$ (kurz: $ggt(i,j)$) der größte gemeinsame Teiler von i und j.

Das heißt (Schreibweise: $a|b$ für „a ist Teiler von b“):

$ggt(i,j)|i,\ ggt(i,j)|j,$ und es gibt kein $k > ggt(i,j)$ mit $k|i$ und $k|j$.

3.5 Typüberprüfung

Typaussagen

Zur ***Typüberprüfung*** (Bestimmung der Typen von Termen und Feststellung eventueller Typisierungsfehler) wird der Typ typ eines Terms t vom SML-System aus der syntaktischen Gestalt von t bestimmt (***Typinferenz***). Im Allgemeinen hängt typ von den Typen gewisser Identifikatoren ab; ***Typaussagen*** sind von der Form

„Unter der Annahme, dass die Identifikatoren $x_1, \ldots, x_n$ die Typen $typ_1, \ldots, typ_n$ haben, hat der Term t den Typ typ."

Formale Schreibweise hierfür:

$$\Gamma \rhd t : typ,$$

wobei $\Gamma = \{x_1 : typ_1, \ldots, x_n : typ_n\}$ die Menge der Typannahmen ist (und immer angenommen sei, dass $x_i \neq x_j$ für $i \neq j$ ist).

Beispiel: $\{a : \textbf{int}\} \rhd \textbf{fn}\ x \Rightarrow a + 2 * x : \textbf{int} \to \textbf{int}$.

Typinferenz

- Typaussagen werden formal ***hergeleitet*** aus ***Typaxiomen*** mit Hilfe von ***Typisierungsregeln***.
- Typaxiome sind „elementare" Typaussagen über Konstanten, Namen, Basisfunktionen (...). Beispiele:

  $\emptyset \rhd 7 : \textbf{int}$,
  $\emptyset \rhd 7.0 : \textbf{real}$,
  $\emptyset \rhd not : \textbf{bool} \to \textbf{bool}$,
  $\{x : typ\} \rhd x : typ$.

- Typisierungsregeln sind von der Gestalt

  $$P_1, \ldots, P_m \vdash K$$

  wobei $P_1, \ldots, P_m, K$ Typaussagen sind. Die $P_1, \ldots, P_m$ heißen ***Prämissen***, K heißt ***Konklusion*** der Regel.

  Informelle Bedeutung:

  „Falls die Typaussagen $P_1, \ldots, P_m$ gelten, so gilt auch die Typaussage K."

- Eine (formale) Herleitung einer Typaussage A ist eine endliche Folge $A_1, \ldots, A_{k-1}, A_k$ von Typaussagen mit:
  (1) $A = A_k$.
  (2) Jedes A_i $(i = 1, \ldots, k)$ ist entweder ein Typaxiom oder Konklusion einer Typisierungsregel mit Prämissen aus $\{A_1, \ldots, A_{i-1}\}$.

Typisierungsregeln: Einige Beispiele

(R1) $\Gamma_1 \rhd t_1 : typ_1,\ \Gamma_2 \rhd t_2 : typ_2 \vdash \Gamma_1 \cup \Gamma_2 \rhd (t_1, t_2) : typ_1 * typ_2$

(R2) $\Gamma_1 \rhd t_1 : \textbf{int},\ \Gamma_2 \rhd t_2 : \textbf{int},\ \Gamma_3 \rhd op : \textbf{int} * \textbf{int} \to \textbf{int}$
$\vdash \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \rhd t_1\ op\ t_2 : \textbf{int}$

(R3) $\Gamma_1 \rhd b : \textbf{bool},\ \Gamma_2 \rhd t_1 : typ,\ \Gamma_3 \rhd t_2 : typ$
$\vdash \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \rhd \textbf{if}\ b\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : typ$

(R4) $\quad \Gamma \cup \{x : typ_1\} \rhd t : typ_2 \;\vdash\; \Gamma \rhd \textbf{fn}\ x \Rightarrow t : typ_1 \to typ_2$

Beispiel einer Typinferenz

Herleitung von $\{a : \textbf{int}\} \;\rhd\; \textbf{fn}\ x \Rightarrow a + 2 * x : \textbf{int} \to \textbf{int}$:

(1)	$\emptyset \;\rhd\; 2 : \textbf{int}$	Typaxiom
(2)	$\emptyset \;\rhd\; + : \textbf{int} * \textbf{int} \to \textbf{int}$	Typaxiom
(3)	$\emptyset \;\rhd\; * : \textbf{int} * \textbf{int} \to \textbf{int}$	Typaxiom
(4)	$\{a : \textbf{int}\} \;\rhd\; a : \textbf{int}$	Typaxiom
(5)	$\{x : \textbf{int}\} \;\rhd\; x : \textbf{int}$	Typaxiom
(6)	$\{x : \textbf{int}\} \;\rhd\; 2 * x : \textbf{int}$	(R2) mit Prämissen (1),(3),(5)
(7)	$\{a : \textbf{int}, x : \textbf{int}\} \;\rhd\; a + 2 * x : \textbf{int}$	(R2) mit Prämissen (2),(4),(6)
(8)	$\{a : \textbf{int}\} \;\rhd\; \textbf{fn}\ x \Rightarrow a + 2 * x : \textbf{int} \to \textbf{int}$	(R4) mit Prämisse (7)

3.6 Ausnahmen und Mustervergleich

Ausnahmen

Mit einer ***Ausnahmedeklaration*** der Form

exception a

kann in SML ein Name a für eine ***Ausnahme*** eingeführt werden. Ein Term der Gestalt

raise a

bewirkt dann, dass die Berechnung (an dieser Stelle) abgebrochen wird. (Danach setzt eine ***Fehlerbehandlung*** ein, deren Wirkung ebenfalls vom Benutzer im Programm bestimmt werden kann.)

Beispiel

```
- exception fak_nicht_korrekt_aufgerufen
  fun fak n = if n<0 then
                raise fak_nicht_korrekt_aufgerufen
              else
              if n=0 then 1 else n*fak(n-1);
exception fak_nicht_korrekt_aufgerufen
val fak = fn : int -> int
- fak ~4;
uncaught exception fak_nicht_korrekt_aufgerufen
```

Mustervergleich

Für die Deklaration einer Funktion mit Fallunterscheidung wie

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

bevorzugt der SML-„Programmierstil“ eine andere Darstellung:

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

Allgemeine Schreibweise solcher Deklarationen (für eine Funktion f):

$$\begin{array}{l}\textbf{fun}\ f\ m_1 = t_1 \\ \quad \mid f\ m_2 = t_2 \\ \qquad \vdots \\ \quad \mid f\ m_k = t_k\end{array}$$

$m_1, \ldots, m_k$ sind (paarweise verschiedene) ***Muster***, $t_1, \ldots, t_k$ sind Terme (gleichen Typs). Bei einem Aufruf von f wird das betreffende Argument mit den Mustern in der angegebenen Reihenfolge verglichen; das erste, bei dem der Vergleich Erfolg hat, bestimmt die Auswahl des entsprechenden Terms t_i.

Muster

- Hängt der zum Muster m_i gehörige Term t_i nicht von m_i ab, so kann für m_i das spezielle Muster „ _“ („beliebig“, ***wildcard***) verwendet werden. Beispiel:

  ```
  fun nullodereins 0 = true
    | nullodereins 1 = true
    | nullodereins n = false
  ```

  mit wildcard:

  ```
  fun nullodereins 0 = true
    | nullodereins 1 = true
    | nullodereins _ = false
  ```

- Das Prinzip der Fallunterscheidung mit Mustern kann auch auf Parameter-Tupel übertragen werden. Beispiele:

  1.
  ```
  (* Ackermann-Funktion *)
  fun ack(0,n) = n+1
    | ack(m,0) = ack(m-1,1)
    | ack(m,n) = ack(m-1,ack(m,n-1))
  ```

  2.
  ```
  fun g(x,_,0) = x+1
    | g(_,y,_) = y+2
  ```

- Allgemein: Ein Muster ist (vorläufig) eine spezielle Konstante (nicht für Zahlen aus **real**), ein Name, eine wildcard oder ein Tupel aus solchen Bestandteilen (mit paarweise verschiedenen Namen).

  (Die Möglichkeit, Muster zu bilden, wird im folgenden Kapitel noch erweitert.)

- Das Ergebnis des Vergleichs eines Musters m mit einem Argument x ist wie folgt definiert:

  (1) Ist m eine spezielle Konstante, so ist der Vergleich erfolgreich, wenn x und m identisch sind. Andernfalls schlägt der Versuch fehl.

  (2) Ist m ein Name, dann ist der Vergleich erfolgreich (und der Wert von x wird an m gebunden).

  (3) Ist m eine wildcard, so ist der Versuch erfolgreich.

  (4) Ist m von der Form $(m_1, \ldots, m_l)$, so ist der Versuch erfolgreich, wenn x von der Form $(x_1, \ldots, x_l)$ ist und die Vergleiche von m_i mit x_i für $i = 1, \ldots, l$ erfolgreich sind. Andernfalls schlägt der Vergleich fehl.

Bemerkungen

- Man beachte: Nicht jede Funktionsdefinition durch Fallunterscheidung lässt sich (allein) durch Mustervergleich formulieren. Beispiel (Fakultäts-Funktion mit Fehlerbehandlung):

```
exception neg_Arg
fun fak 0 = 1
  | fak n = if n<0 then raise neg_Arg
            else n*fak(n-1)
```

- Die Muster in einer Funktionsdeklaration sollten „alle möglichen Fälle“ überdecken; andernfalls liefert das System eine „Warnung“. Beispiel:

```
- fun unvollstaendig 0 = true
    | unvollstaendig 1 = true;
Warning: match nonexhaustive
- unvollstaendig 1;
val it = true : bool
- unvollstaendig 7;
uncaught exception nonexhaustive match failure
```

- Die allgemeine Form von Funktionsdeklarationen mit Mustervergleich lässt sich leicht erweitern auf Funktionen höherer Ordnung, z.B.:

$$\begin{array}{l}\textbf{fun}\ f\ g\ m_1 = t_1 \\ \quad \mid f\ g\ m_2 = t_2 \\ \qquad \vdots \\ \quad \mid f\ g\ m_k = t_k\end{array}$$

  (Beispiele folgen in Abschnitt 4.4.)

- Die bisherige Form von Funktionsdeklarationen lässt sich dem Konzept des Mustervergleichs als Spezialfall mit nur einem Muster unterordnen.

Kapitel 4
Strukturierte Daten

4.1 Rechenstrukturen

Daten in komplexen Anwendungen

- Zur Modellierung und Darstellung von „komplexen Informationen“ (Adressenkarteien, Bankkonten, ...) sind zusammengesetzte, „strukturierte“ Daten angebracht. Die jeweilige Art einer solchen Strukturierung, d.h. der strukturelle Aufbau der betreffenden Daten wird als deren ***Datenstruktur*** bezeichnet.
- Die entsprechenden zusammengesetzten Datentypen sind bestimmt durch die jeweiligen Datenmengen und (wesentlich) die darauf definierten Basisfunktionen.

Rechenstrukturen

- Eine ***Rechenstruktur*** ist eine Menge von Datentypen zusammen mit einer Menge von Funktionen auf diesen Typen.
- Die Bezeichnungen der Datentypen einer Rechenstruktur heißen ***Sorten***. Sie bilden zusammen mit den Funktionsbezeichnungen (die auch Konstanten sein können) und zugehörigen Typ- (genauer: Sorten-) Angaben die ***Signatur*** der Rechenstruktur.
- Eine Rechenstruktur, in der einer der enthaltenen Datentypen typ ausgezeichnet ist, heißt auch ***Rechenstruktur von*** typ.

Rechenstrukturen in der Algorithmusentwicklung

- Die Funktionsbezeichnungen (einschließlich Konstanten) der Signatur einer Rechenstruktur eines Datentyps typ bestimmen die auf typ erlaubten Basisfunktionen (und Bezeichnungen von Datenelementen).
- Bei der Entwicklung von Algorithmen ist es methodisch vorteilhaft, dem Problem angepasste Rechenstrukturen mit „im mathematischen Sinne“ definierten Datenmengen und Funktionen zu verwenden. Rechenstrukturen in diesem Sinne heißen auch ***abstrakte Datentypen***. Konkrete Realisierungen (etwa in einer Programmiersprache) heißen auch ***konkrete Datentypen***.

Beispiele

1. Die Rechenstruktur von **bool**

   Abstrakt (gemäß Abschnitt 2.2):

Signatur	Bedeutung
bool	Menge der Wahrheitswerte
$true$: **bool** $false$: **bool**	} alle Wahrheitswerte
$\neg$: **bool** $\rightarrow$ **bool**	Negation
$\wedge$: **bool** $\times$ **bool** $\rightarrow$ **bool**	Konjunktion
$\vee$: **bool** $\times$ **bool** $\rightarrow$ **bool**	Disjunktion

Konkrete Realisierung in SML: direkt verfügbar (mit teilweise anderen Bezeichnungen).

2. Die Rechenstruktur „der natürlichen Zahlen“ (d.h.: von **nat**)

   Abstrakt:

Signatur	Bedeutung
nat	$\mathbb{N}_0$
bool	Menge der Wahrheitswerte
$0, 1, 2, \ldots$: **nat**	alle Elemente von $\mathbb{N}_0$
$+$: **nat** $\times$ **nat** $\rightarrow$ **nat**	Addition
$-$: **nat** $\times$ **nat** $\rightarrow$ **nat**	Subtraktion
$\vdots$	
$=$: **nat** $\times$ **nat** $\rightarrow$ **bool**	Gleichheit
$<$: **nat** $\times$ **nat** $\rightarrow$ **bool**	Kleiner-Relation
$\vdots$	

   (Die an erster Stelle aufgeführte Sorte bezeichne (auch im Folgenden) den ausgezeichneten Datentyp.)

   Konkrete Realisierung in SML: „über **int**“.

4.2 Tupel

Die Rechenstruktur der Tupel (über Datentypen $typ_1, \ldots, typ_n$)

- Informell: Aus Datentypen $typ_1, \ldots, typ_n$ ($n \geq 1$) wird der Datentyp

  $$typ_1 \times \ldots \times typ_n = \{(x_1, \ldots, x_n) \mid x_1 \in typ_1, \ldots, x_n \in typ_n\}$$

  gebildet.

- Signatur:

  $typ_1 * \ldots * typ_n$
  $typ_1, \ldots, typ_n$

$$
\begin{aligned}
&(\,,\ldots,\,) : typ_1 \to typ_2 \to \ldots \to typ_n \to typ_1 * typ_2 * \ldots * typ_n\\
&\#1 : typ_1 * \ldots * typ_n \to typ_1\\
&\vdots\\
&\#n : typ_1 * \ldots * typ_n \to typ_n
\end{aligned}
$$

- Bedeutung der Funktionszeichen:
  - $(\,,\ldots,\,)$: Bildet aus Elementen $x_1, \ldots, x_n$ von $typ_1, \ldots, typ_n$ das Tupel $(x_1, \ldots, x_n)$.

    Beispiel: $\underbrace{(\,,\ldots,\,)\ 7\ \text{"abc"}\ 3.14}_{} = (7, \text{"abc"}, 3.14)$.

    Schreibweise dieser Funktionsanwendung direkt: $(7, \text{"abc"}, 3.14)$.
  - $\#i$ $(1 \leq i \leq n)$: Liefert die $i-$te Komponente x_i eines Tupels $(x_1, \ldots, x_n)$.

Bemerkung

Bezeichnungen für Funktionen in einer Signatur für einen Datentyp, die Elemente des Typs als Werte liefern (hier: $(\,,\ldots,\,)$), heißen ***Konstruktoren*** für den Typ (ebenso eventuell vorhandene Konstanten des Typs). Bezeichnungen für Funktionen, die einzelne „Teil"-Daten von (zusammengesetzten) Daten liefern (hier: $\#1, \#2, \ldots$), heißen ***Selektoren***.

Tupel in SML

- Direkt verfügbar in der angegebenen Form mit folgender zusätzlicher Eigenschaft: Auf Tupeln, die aus Daten zusammengesetzt sind, für deren Typen $=$ und $<>$ als Basisfunktionen definiert sind, können $=$ und $<>$ (komponentenweise Gleichheit bzw. Ungleichheit) ebenfalls als Basisfunktionen verwendet werden.

  Bemerkung:
  SML-Typen, für die $=$ und $<>$ definiert sind (bei zusammengesetzten Typen mit analoger Einschränkung) heißen ***Gleichheitstypen***.

  Beispiel:

```
- val p = ("Meier",25);
val p = ("Meier",25) : string * int
- #1(p);
val it = "Meier" : string
- (#1(p),#2(p)+7);
val it = ("Meier",32) : string * int
- p = ("Huber",25);
val it = false : bool
- val q = ((3.0,7),fn x => 2*x);
val q = ((3.0,7),fn) : (real * int) * (int -> int)
```

- Weitere verfügbare Form (***Records***):

  Verwendung von explizit anzugebenden, frei wählbaren Namen $name_1, \ldots, name_n$ zur Kennzeichnung der Komponenten und Selektoren $\#name_i$ statt $\#i$; Reihenfolge der Komponenten unerheblich.

SML-Schreibweise: $\{name_1 = x_1, \ldots, name_n = x_n\}$ (statt $(x_1, \ldots, x_n)$)
SML-Typbezeichnung: $\{name_1 : typ_1, \ldots, name_n : typ_n\}$ (statt $typ_1 * \ldots * typ_n$)

Beispiel:

```
- val p = {Name="Meier",Alter=25};
val p = {Alter=25,Name="Meier"} : {Alter:int, Name:string}
-              (* Beachte Umordnung der Namen ! *)
  #Name(p);
val it = "Meier" : string
```

Typdeklarationen

Wie Funktionen und Datenelemente können in einem Programm auch Typen mit einem Namen versehen werden. Dieser wird in SML in einer ***Typdeklaration*** der Form

type ⟨*Name*⟩ = ⟨*Typ*⟩

eingeführt. Beispiel:

```
- type Student = {Name:string, Alter:int};
type Student = {Alter:int, Name:string}
- val p = {Name="Meier", Alter=25};
val p = {Alter=25,Name="Meier"} : {Alter:int, Name:string}
```

Beispiele

1.
```
type Uhrzeit = {std:int,min:int}
fun Fahrtzeit'''(an:Uhrzeit,ab:Uhrzeit) =
      let val z = (#std(an) - #std(ab))*60 + #min(an) - #min(ab)
      in if z<0 then z+1440 else z
      end
```

2.
```
- type Datum = int*int   (* Tag,Monat *)
  type Uhrzeit = {std:int,min:int}
  type Termin = {d:Datum,u:Uhrzeit,stichwort:string}
  val Eintrag = {d=(12,2),u={std=9,min=30},stichwort="Arzt"}
  fun Verlegung(t:Termin,neuezeit:Uhrzeit) =
      {d= #d(t),u=neuezeit,stichwort= #stichwort(t)}
  val Eintrag_neu = Verlegung(Eintrag,{std=10,min=15});
 ⋮
val Eintrag = {d=(12,2),stichwort="Arzt",u={min=30,std=9}}
  : {d:Datum, stichwort:string, u:Uhrzeit}
val Verlegung =
  fn : Termin * Uhrzeit -> {d:Datum, stichwort:string, u:Uhrzeit}
val Eintrag_neu = {d=(12,2),stichwort="Arzt",u={min=15,std=10}}
  : {d:Datum, stichwort:string, u:Uhrzeit}
```

Bemerkung

Die Basisfunktionen der Tupel- (und Record-) Typen sind im formalen Sinne von Abschnitt 2.4 polymorphe Funktionen, z.B.:

$$\#1 : 'a1 * \ldots * 'an \to 'a1$$

Dies gilt auch für die zusätzlich verfügbaren Funktionen $=$ und $<>$, z.B.:

$$= : (''a1 * \ldots * ''an) * (''a1 * \ldots * ''an) \to \textbf{bool}$$

(Beachte: Typvariablen, die für Gleichheitstypen stehen, werden mit $''a, ''b, \ldots$ bezeichnet.)

4.3 Variante Daten

Die Rechenstruktur der varianten Daten (aus Datentypen $typ_1, \ldots, typ_n$)

- Informell: Aus Datentypen $typ_1, \ldots, typ_n$ ($n \geq 1$) wird der Datentyp

  $$typ_1 \mid \ldots \mid typ_n$$

  gebildet. $typ_1, \ldots, typ_n$ müssen dabei nicht alle verschieden sein.

- Signatur:

  $typ_1 \mid \ldots \mid typ_n$
  $typ_1, \ldots, typ_n$ (nicht notwendig verschieden)
  $inj_1 : typ_1 \to typ_1 \mid \ldots \mid typ_n$
  $\vdots$
  $inj_n : typ_n \to typ_1 \mid \ldots \mid typ_n$

- Bedeutung der Funktionszeichen:

  inj_i bildet (für $i = 1, \ldots, n$) ein Element x von typ_i auf x als Element von $typ_1 \mid \ldots \mid typ_n$ ab (***Injektion*** von typ_i in $typ_1 \mid \ldots \mid typ_n$).

Variante Daten in SML

Nicht direkt verfügbar, aber: Variantentypen können – mit frei wählbaren Namen für die Injektionen – durch ***datatype-Deklarationen*** eingeführt und mit Namen versehen werden. Diese Art der Deklarationen ist von der Form

datatype ⟨*Name*⟩ = ⟨*InjName*⟩ **of** ⟨*Typ*⟩ | ... | ⟨*InjName*⟩ **of** ⟨*Typ*⟩

Mustervergleich mit varianten Daten

- Jedes Datenelement eines Variantentyps $typ_1 \mid \ldots \mid typ_n$ kann eindeutig dargestellt werden in der Form $inj_i(x)$ mit $x \in typ_i$.

- Neue Art von Mustern (in Erweiterung der Festlegungen in Abschnitt 3.6):

  $inj_i\ m,$

  wobei m wieder ein Muster ist.

  (Der Vergleich eines solchen Musters mit einem Argument y in einem Funktionsaufruf ist genau dann erfolgreich, wenn y von der Form $inj_i\ x$ (oder $inj_i(x)$) ist und der Vergleich von m mit x erfolgreich ist.)

Beispiel:

```
- datatype Figur = Kreis of real | Quadrat of real |
                   Dreieck of real*real*real;
datatype Figur
  = Dreieck of real * real * real | Kreis of real | Quadrat of real
- Kreis(1.2 + 3.1);
val it = Kreis 4.3 : Figur
-       (* Die folgende Funktion bestimmt zu einer Figur den Radius
           eines Kreises, der gleichen Umfang wie die Figur hat *)
  fun KglUm (Kreis r)         = r
    | KglUm (Quadrat a)       = 2.0*a/3.14
    | KglUm (Dreieck (a,b,c)) = (a+b+c)/(2.0*3.14);
val KglUm = fn : Figur -> real
- KglUm(Quadrat(7.1));
val it = 4.52229299363 : real
```

Sonderfälle

- Die Anzahl der Typen, aus denen ein Variantentyp gebildet wird, kann 1 sein, z.B.:

  ```
  datatype datum = Datum of int*int
  ```

  (Der Typ **int** $*$ **int** erhält auf diese Weise den Konstruktor $Datum$, der entsprechende Werte mnemotechnisch besser beschreiben hilft.)

  Beispielterm vom Typ $datum$:

  ```
  Datum (25,6)
  ```

- Beliebig viele der Konstruktoren eines Variantentyps können auch Konstanten sein (die dann ohne weitere Zusätze notiert werden). Falls dies für alle Konstruktoren zutrifft, heißt der betreffende Typ auch ***Aufzählungstyp***. Beispiel:

  ```
  datatype Farbe = Rot | Blau | Gelb | Gruen
  ```

  Beispielterm vom Typ $Farbe$:

  ```
  Blau
  ```

4.4 Listen

Die Rechenstruktur der Listen (über einem Datentyp typ)

- Wichtige, in der Praxis sehr häufig vorkommende Datentypen sind die Mengen typ^* aller (endlichen) Folgen über anderen Datentypen typ. Diese können mit verschiedenen Basisfunktionen zu ihrer Bearbeitung ausgestattet werden, im Zusammenhang mit den nachfolgend ausgewählten Funktionen heißen sie ***Listen*** (***Sequenzen***). Sie sind ***homogene*** Datenstrukturen (alle Komponenten haben gleichen Typ) im Gegensatz zu den (im Allgemeinen) ***inhomogenen*** Tupeln.

- Bezeichnung von Listentypen: typ **list** (für typ^*).

- Signatur:

  typ **list**
  typ
  bool
  $nil : typ$ **list**
  $:: : typ * typ$ **list** $\rightarrow typ$ **list**
  $@ : typ$ **list** $* typ$ **list** $\rightarrow typ$ **list**
  $hd : typ$ **list** $\rightarrow typ$
  $tl : typ$ **list** $\rightarrow typ$ **list**
  $null : typ$ **list** $\rightarrow$ **bool**

- Bedeutung der Konstanten und Funktionszeichen:

  – nil: leere Liste ε.

  – $::$ („Vornanhängen“ eines neuen Elements an eine Liste):

  $$y :: (x_1, \ldots, x_n) = (y, x_1, \ldots, x_n) \qquad \text{(Infixschreibweise).}$$

  – $@$ (Zusammenfügen zweier Listen):

  $$(y_1, \ldots, y_m) \mathbin{@} (x_1, \ldots, x_n) = (y_1, \ldots, y_m, x_1, \ldots, x_n)$$
  (Infixschreibweise).

  – hd („Kopf“ der Liste, nicht definiert für nil):

  $$hd(x_1, \ldots, x_n) = x_1.$$

  (Beachte: hd ist der einzige Listen-Selektor.)

  – tl (Rest der Liste ohne Kopf, nicht definiert für nil):

  $$tl(x_1, x_2, \ldots, x_n) = (x_2, \ldots, x_n).$$

  – $null$ (Test auf „Leersein“):

  $$null(x) = true \;\Leftrightarrow\; x = nil.$$

Bemerkung

Jede Liste kann (eindeutig) mit den Konstruktoren nil und :: erzeugt werden. Genauer: Jede Liste vom Typ typ **list** ist entweder nil oder von der Form $x :: xt$ mit x vom Typ typ und xt vom Typ typ **list** (vgl. induktive Definition in Abschnitt 2.3).

Listen in SML

- Direkt verfügbar in der angegebenen Form und als Gleichheitstyp mit $=$ und $<>$ vom Typ

  $''a$ **list** $*''a$ **list** $\rightarrow$ **bool**

- Zusätzlich mögliche Schreibweisen:
  $[\,]$ für nil,
  $[x_1, x_2, \ldots, x_n]$ für $x_1 :: x_2 :: \ldots :: x_n :: nil$.

- :: und @ sind Infix-Operatoren mit Präzedenz 4 und Assoziierung nach rechts.

- Beispiel:

```
- val x = 17::nil;
val x = [17] : int list
- val y = 9::2::x;
val y = [9,2,17] : int list
- hd y;
val it = 9 : int
- tl y;
val it = [2,17] : int list
- null(tl x);
val it = true : bool
- x = y;
val it = false : bool
- x @ y;
val it = [17,9,2,17] : int list
- [x,y];
val it = [[17],[9,2,17]] : int list list
- [(9,2),(3,5)];
val it = [(9,2),(3,5)] : (int * int) list
```

Mustervergleich mit Listen

Als Muster sind zugelassen die Konstante nil (oder $[\,]$) sowie (in zusätzlicher Erweiterung der bisherigen Festlegungen):

$m_1 :: m_2$,

wobei m_1 und m_2 Muster sind. Gemäß obiger Bemerkung überdecken diese Muster alle möglichen Listen.

(Der Vergleich eines Musters $m_1 :: m_2$ mit einem Argument x in einem Funktionsaufruf

ist genau dann erfolgreich, wenn x sich als $x_1 :: x_2$ darstellen lässt und die Vergleiche von m_1 mit x_1 und von m_2 mit x_2 erfolgreich sind.)

Einige Grundalgorithmen für Listen

1. Bestimmung der Länge einer Liste ($'a$ **list** $\to$ **int**)

```
fun length nil     = 0
  | length (_::xt) = 1+length(xt)
```

2. Suchen eines Elements in einer Liste ($''a$ **list** $*\,''a \to$ **bool**)

```
fun enthalten (nil,a)   = false
  | enthalten (x::xt,a) = x=a orelse enthalten(xt,a)
```

($enthalten(y, a) = true$ genau dann, wenn a eine Komponente von y ist.)

3. Spiegeln (Revertieren) einer Liste ($'a$ **list** $\to$ $'a$ **list**)

```
fun rev nil     = nil
  | rev (x::xt) = (rev xt)@[x]
```

($rev : (x_1, x_2, \ldots, x_n) \mapsto (x_n, \ldots, x_2, x_1)$.)

4. Bestimmung der letzten Komponente einer nicht-leeren Liste ($'a$ **list** $\to$ $'a$)

```
exception Last
fun last nil     = raise Last
  | last [x]     = x
  | last (_::xt) = last xt
```

Bemerkung: $last$ kann auch ausgedrückt werden durch $last(x) = hd(rev(x))$, d.h. durch Funktionskomposition: $last = hd \circ rev$. In SML ist $\circ$ als Standardfunktion verfügbar, also:

```
val last = hd o rev
```

(Beachte: nicht als Funktionsdeklaration!)

5. ***Sortieren*** einer Liste ganzer Zahlen (***durch Einfügen***) (**int list** $\to$ **int list**)

(Eine Folge $(x_1, x_2, \ldots, x_n)$ ganzer Zahlen heißt (aufsteigend) ***sortiert*** (***geordnet***), wenn $x_1 \leq x_2 \leq \ldots \leq x_n$ gilt.)

```
fun insertel (a:int,nil)   = [a]
  | insertel (a:int,x::xt) = if a <= x then a::x::xt
                             else x::insertel(a,xt)
fun inssort nil     = nil
  | inssort (x::xt) = insertel(x,inssort(xt))
```

6. Anwendung einer Funktion auf alle Komponenten einer Liste

$f : typ_1 \to typ_2 \quad \mapsto \quad map(f) : typ_1$ **list** $\to typ_2$ **list**,
$map(f)(x_1, \ldots, x_n) = (f(x_1), \ldots, f(x_n))$.

(D.h.: $map : ('a \to 'b) \to 'a$ **list** $\to 'b$ **list**.)

```
fun map f nil     = nil
  | map f (x::xt) = f(x)::map(f)(xt)
```

Beispielanwendung:

```
- map (fn x => x*x) [1,2,3,4];
val it = [1,4,9,16] : int list
- map(map(fn x => x*x)) [[1,2],[3],[4,5,6]];
val it = [[1,4],[9],[16,25,36]] : int list list
```

7. (Rechts-) ***Faltung*** einer Liste mit einer Funktion

   $f : typ_1 * typ_2 \to typ_2 \quad \mapsto \quad foldr(f) : typ_2 \to typ_1\ \textbf{list} \to typ_2,$
   $foldr(f)(z)(x_1, \ldots, x_n) = f(x_1, f(x_2, \ldots, f(x_n, z)\ldots)).$

   (D.h.: $foldr : ('a * 'b \to 'b) \to 'b \to 'a\ \textbf{list} \to 'b$.)

   ```
   fun foldr f z nil     = z
     | foldr f z (x::xt) = f(x,foldr f z xt)
   ```

   Beispielanwendung:

   ```
   - foldr (op +) 0 [2,7,3,8];  (* op + ist + als Funktion *);
   val it = 20 : int
   ```

8. ***Filtern*** einer Liste

   $p : typ \to \textbf{bool} \quad \mapsto \quad filter(p) : typ\ \textbf{list} \to typ\ \textbf{list},$
   $filter(p)(x_1, \ldots, x_n) =$ „Teilliste“ derjenigen x_i mit $p(x_i) = true$.

   (D.h.: $filter : ('a \to \textbf{bool}) \to 'a\ \textbf{list} \to 'a\ \textbf{list}$.)

   ```
   fun filter p nil     = nil
     | filter p (x::xt) = if p(x) then x :: filter p xt
                          else filter p xt
   ```

   Beispielanwendung:

   Dabei verwendet: Standardfunktion $explode$: **string** $\to$ **char list**, die eine Zeichenreihe x in die Liste der Zeichen von x umwandelt, z.B.:

   $explode(\text{"abc"}) = (\#\text{"a"}, \#\text{"b"}, \#\text{"c"})$.

   ```
   - val beginnt_mit_B = filter (fn x => hd(explode(x)) = #"B")
     (* Beachte: Wertdeklaration ! *);
   val beginnt_mit_B = fn : string list -> string list
   - beginnt_mit_B ["Udo","Bernd","Xaver","Karl","Boris","Leo"];
   val it = ["Bernd","Boris"] : string list
   ```

Ein Anwendungsbeispiel

Eine Bank berechnet für jede (Soll- und Haben-) Buchung eine Gebühr. Ist der Buchungsbetrag kleiner als 1000 EUR, beträgt die Gebühr 30 Cent, andernfalls 50 Cent. Am Monatsende ist zu den getätigten Buchungen die Gesamtgebühr zu berechnen.

Modellierung: Buchungen eines Monats als endliche Folge von EUR-Beträgen (**real list**), Gesamtgebühr als Cent-Betrag (**int**).

Algorithmus:

1. „direkt“:

```
fun Gebuehr nil     = 0
  | Gebuehr (b::bt) = if abs(b) < 1000.0 then 30+Gebuehr(bt)
                      else 50+Gebuehr(bt)
    (* Argument von Gebuehr ist die Liste der Buchungen
       Benutzt: Standardfunktion abs (Absolutbetrag) *)
```

2. Mit der (als Standardfunktion verfügbaren) Funktion $foldr$:

```
fun hinzuaddieren(x,y) =
      if abs(x) < 1000.0 then 30+y else 50+y
val Gebuehr' = foldr hinzuaddieren 0
```

Weitere listenartige Datenstrukturen

In der Praxis wichtig sind auch gewisse Varianten von Listen mit einer modifizierten Auswahl von Basisfunktionen:

- ***Stapel***: Endliche Folgen mit der Konstanten nil und den Basisfunktionen ::, hd, tl und $null$. (Die drei ersten Funktionen werden dann meist auch $push$, top und pop genannt.)
- ***Schlangen***: Endliche Folgen mit der Konstanten nil, den Basisfunktionen hd, tl und $null$ sowie einer weiteren Basisfunktion $enter$ mit der Bedeutung

  $$enter((x_1, \ldots, x_n), y) = (x_1, \ldots, x_n, y)$$

  („Hintenanfügen“ eines Elements).
- Stapel und Schlangen sind in SML nicht direkt in dieser Form vorhanden. Sie können als „Teil-“ Struktur von Listen verwendet werden (mit einer leicht zu definierenden Funktion $enter$).

  Möglichkeit zur „genaueren Definition“: In Abschnitt 5.4.

4.5 Reihungen

Die Rechenstruktur der Reihungen (über einem Datentyp typ)

- Endliche Folgen, die mit den nachfolgenden Basisfunktionen ausgestattet sind, heißen ***Reihungen*** (***Vektoren***).
- Bezeichnung von Reihungstypen: typ **vect** (für typ^*).

- Signatur:

  typ **vect**
  typ
  nat
  $dim : typ\ \mathbf{vect} \to \mathbf{nat}$
  $init : \mathbf{nat} * typ \to typ\ \mathbf{vect}$
  $get : typ\ \mathbf{vect} * \mathbf{nat} \to typ$
  $update : typ\ \mathbf{vect} * \mathbf{nat} * typ \to typ\ \mathbf{vect}$

- Bedeutung der Funktionszeichen:
  - dim (Länge der Reihung):
    $dim(x_1, \ldots, x_n) = n.$
  - $init$ (Erzeugen einer Reihung):
    $init(n, a) = (\underbrace{a, a, \ldots, a}_{n}).$
  - get (Zugriff auf die Komponenten einer Reihung,
    nur definiert für $1 \leq i \leq dim(x)$):
    $get((x_1, \ldots, x_n), i) = x_i.$
  - $update$ (Erzeugen einer Reihung mit veränderter Komponente,
    nur definiert für $1 \leq i \leq dim(x)$):
    $update((x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n), i, a) =$
    $(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n).$

Reihungen in SML

Die Rechenstruktur der Reihungen ist in SML (in der angegebenen Form) nicht direkt verfügbar.

Zur Realisierung: Siehe Abschnitt 5.4.

Zwei Grundalgorithmen für Reihungen

1. (Komponentenweise) Gleichheit zweier Reihungen ($''a$ **vect** $*\,''a$ **vect** $\to$ **bool**)

```
fun eqallg(x,y,k) =  (* Vorbedingung: k>0; dim(x)=dim(y).
                        Bestimmt, ob x und y ab k-ter
                        Komponente gleich sind *)
     if k>dim(x) then true
     else get(x,k)=get(y,k) andalso eqallg(x,y,k+1)
fun reiheq(x,y) = dim(x) = dim(y) andalso eqallg(x,y,1)
```

2. Suchen eines Datenelements in einer Reihung ($''a$ **vect** $*\,''a \to$ **bool**)

```
fun enthallg(x,k,a) =  (* Vorbedingung: k >= 1 *)
     if k > dim(x) then false
     else a=get(x,k) orelse enthallg(x,k+1,a)
fun enthalten1(x,a) = enthallg(x,1,a)
```

4.6 Binärbäume

Mathematische Definition

Es sei A eine Menge. Die Menge A^{Δ} der ***Binärbäume*** über A ist induktiv definiert wie folgt:

a) A^{Δ} enthält den ***leeren Binärbaum*** τ.

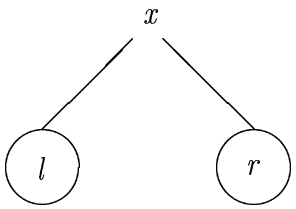
b) Sind $x \in A$ und $xl, xr \in A^{\Delta}$, so ist das 3-Tupel (x, xl, xr) ein Element von A^{Δ}.

x heißt ***Wurzel***, xl ***linker Unterbaum***, xr ***rechter Unterbaum*** eines Binärbaums (x, xl, xr). Ein Binärbaum (x, τ, τ) heißt ***Blatt***. Ein von τ verschiedener Binärbaum heißt ***nicht-leer***.

Die ***Knoten*** und ***Teilbäume*** eines Binärbaums sind induktiv gegeben wie folgt: τ hat keine Knoten und nur τ als Teilbaum. Die Knoten von (x, xl, xr) sind x und alle Knoten von xl und alle Knoten von xr. Die Teilbäume von (x, xl, xr) sind (x, xl, xr) und alle Teilbäume von xl und alle Teilbäume von xr.

Grafische Darstellung von Binärbäumen

(Nicht-leere) Binärbäume (x, xl, xr) werden oft grafisch dargestellt durch:

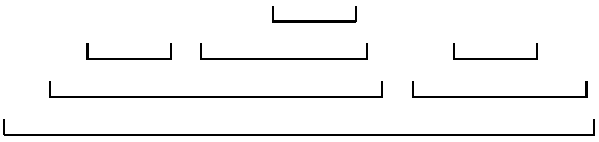


wobei an den Stellen l und r die entsprechenden Darstellungen von xl und xr angesetzt sind. „Zweige“, die auf τ führen, werden weggelassen.

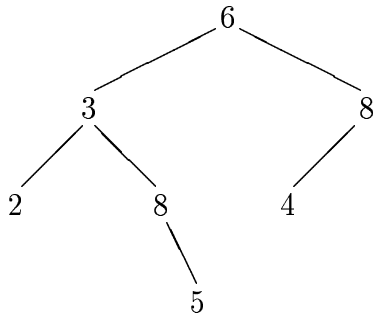
Beispiel

Binärbaum: $(6, (3, (2, \tau, \tau), (8, \tau, (5, \tau, \tau))), (8, (4, \tau, \tau), \tau))$

Induktiver Aufbau:



Grafische Darstellung:



Knoten: $6, 3, 8, 2, 8, 4, 5$

Blätter, die als Teilbäume in dem Binärbaum vorkommen: $(2, \tau, \tau), (5, \tau, \tau), (4, \tau, \tau)$
(auch kurz: $2, 5, 4$)

Die Rechenstruktur der Binärbäume (über einem Datentyp typ)

- Bezeichnung von Binärbaumtypen: typ **bintree** (für $typ^{\triangle}$).
- Signatur:

  typ **bintree**
  typ
  bool
  $empty : typ$ **bintree**
  $build : typ * typ$ **bintree** $* typ$ **bintree** $\rightarrow typ$ **bintree**
  $root : typ$ **bintree** $\rightarrow typ$
  $left : typ$ **bintree** $\rightarrow typ$ **bintree**
  $right : typ$ **bintree** $\rightarrow typ$ **bintree**
  $isempty : typ$ **bintree** $\rightarrow$ **bool**

- Bedeutung der Konstanten und Funktionszeichen:
  - $empty$: leerer Binärbaum τ.
  - $build$ („Zusammensetzen“ eines Binärbaums):
    $build(x, xl, xr) = (x, xl, xr)$.
  - $root$ (Wurzel des Binärbaums, nicht definiert für $empty$):
    $root(x, xl, xr) = x$.
  - $left$ (linker Unterbaum des Binärbaums, nicht definiert für $empty$):
    $left(x, xl, xr) = xl$.
  - $right$ (rechter Unterbaum des Binärbaums, nicht definiert für $empty$):
    $right(x, xl, xr) = xr$.
  - $isempty$ (Test auf „Leersein“):
    $isempty(x) = true \Leftrightarrow x = empty$.

Binärbäume in SML

Die Rechenstruktur der Binärbäume ist in SML nicht direkt verfügbar.
Zur Realisierung: Siehe Abschnitt 5.4.

Einige Grundalgorithmen für Binärbäume

1. Bestimmung der Anzahl der Knoten eines Binärbaums ($'a$ **bintree** $\rightarrow$ **int**)

```
fun knotanz z =
      if isempty z then 0
      else 1 + knotanz(left(z)) + knotanz(right(z))
```

2. Gleichheit zweier Binärbäume ($''a$ **bintree** $*''a$ **bintree** $\rightarrow$ **bool**)

```
fun bbeq(z1,z2) =
      if   isempty(z1) orelse isempty(z2)
      then isempty(z1) andalso isempty(z2)
      else root(z1) = root(z2) andalso
           bbeq(left(z1),left(z2)) andalso
           bbeq(right(z1),right(z2))
```

3. Suchen eines Datenelements in einem Binärbaum ($''a$ **bintree** $*''a \rightarrow$ **bool**)

```
fun enthalten2(z,a) =
      if isempty z then false
      else a = root(z) orelse
           enthalten2(left(z),a) orelse
           enthalten2(right(z),a)
```

4. ***Linearisierung*** eines Binärbaums in ***Vorordnung*** ($'a$ **bintree** $\rightarrow$ $'a$ **list**)

```
fun linvor z =
      if isempty z then nil
      else [root(z)] @ linvor(left(z)) @ linvor(right(z))
```

5. ***Linearisierung*** eines Binärbaums in ***symmetrischer Ordnung***
($'a$ **bintree** $\rightarrow$ $'a$ **list**)

```
fun linsym z =
      if isempty z then nil
      else linsym(left(z)) @ [root(z)] @ linsym(right(z))
```

6. ***Linearisierung*** eines Binärbaums in ***Nachordnung*** ($'a$ **bintree** $\rightarrow$ $'a$ **list**)

```
fun linnach z =
      if isempty z then nil
      else linnach(left(z)) @ linnach(right(z)) @ [root(z)]
```

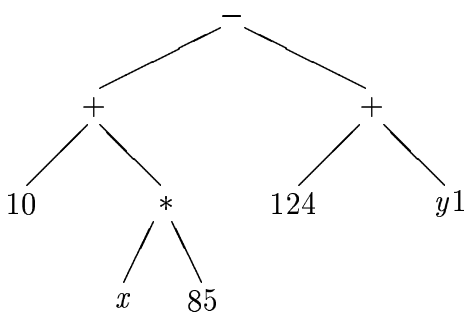
Anwendungsbeispiel

- Die abstrakte Syntaxdarstellung (siehe Abschnitt 3.4) von Termen kann in Form einer „Baumstruktur“ geschehen. Für den einfachen Fall von Termen, die nur mit Basisfunktionen gebildet sind, ergeben sich hierbei Binärbäume.

- Beispiel

  Term: $10 + x * 85 - (124 + y1)$

  als Baum:



- Für die eigentliche Auswertung ist noch eine andere Darstellung besser geeignet, in der alle Operatoren in Postfixschreibweise verwendet werden (und dadurch keine Klammern mehr benötigt werden); im Beispiel:

  $10\ x\ 85\ *\ +\ 124\ y1\ +\ -$

  Die Umwandlung der Binärbaumdarstellung in diese Darstellung ist die Linearisierung in Nachordnung.

- Konkreter Algorithmus für diesen Übergang (Baumeinträge vom Typ **string**):

```
fun pfumw(b : string bintree) =
      if isempty b then nil
      else pfumw(left(b)) @ pfumw(right(b)) @ [root(b)]
```

  (Typ von $pfumw$: **string bintree** $\rightarrow$ **string list**.)

Bemerkung

Es gibt noch eine ganze Reihe anderer Baumstrukturen,z.B.:

- Bäume mit p ($p \geq 2$) Unterbäumen: ***p-adische Bäume***. (Binärbäume sind 2-adische Bäume.)

- Datenelemente (des zugrunde liegenden Typs) sind nur „in den Blättern“ vorhanden: ***beblätterte Binärbäume***.

Kapitel 5
Methodisches Programmieren

5.1 Modularisierung

Beispiel einer komplexen Anwendung

In einer Firma sollen durch ein Programm Quittungen erstellt werden:

Fma. L. Kaufgut *12.11.2002*
Ladenstr. 17
77777 Handelstadt

QUITTUNG

Wir bestätigen, von Herrn P. Schulze

EUR 218.37 (zweihundertachtzehn)

erhalten zu haben.

Diese Quittung wurde maschinell erstellt und trägt keine Unterschriften.

- Aufgabe: Bei Eingabe von Datum, Kundenname und Zahlungsbetrag soll eine Quittung in dieser Form erstellt werden.
- Modellierung:

Kundenname:	**string**,
allgemeiner Quittungstext:	**string**,
Datum:	**string**,
Zahlungsbetrag:	**real**.

- Erste Grobgliederung der Aufgabe in Teilaufgaben:
  - Erzeugung der Wortdarstellung zum Zahlungsbetrag,
  - Einsetzen der Eingaben und der erzeugten Wortdarstellung in den festen Quittungstext,
  - Erstellung der endgültigen Fassung der Quittung gemäß gewünschtem Layout (bei langen Wortdarstellungen ist eventuell ein Zeilenumbruch erforderlich),
  - (– Eingabe der Parameter, Drucken der Quittung.)

Das Modulkonzept

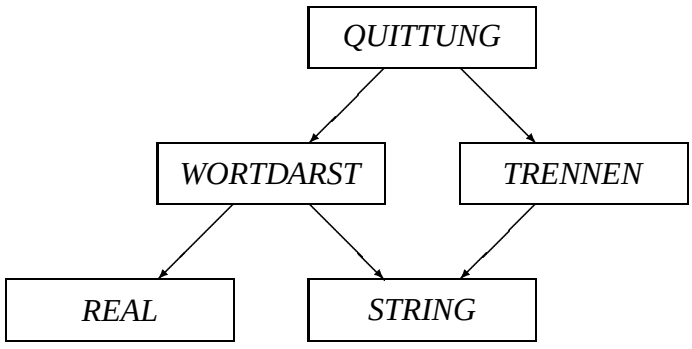
- Ein ***Modul*** ist eine Menge von Datentypen zusammen mit einer Menge von Funktionen auf diesen Typen und eventuell noch weiteren Bestandteilen (die im Folgenden noch besprochen werden).
- Insbesondere ist jede Rechenstruktur ein Modul.
- Im Allgemeinen enthält ein Modul „interne“ Bestandteile, die „außerhalb“ (d.h. von anderen Moduln) nicht verwendbar sein sollen. Diejenigen Bestandteile eines Moduls, die von anderen verwendbar sind, bilden seine ***Schnittstelle***.
- ***Modularisierung***: Zerlegung einer Gesamtaufgabe (zu verarbeitende Daten und Verarbeitungsalgorithmen) in Teile und Festlegung der für die einzelnen Teile zu erstellenden Schnittstellen.

Beispiel: Modularisierung der Quittungserstellung

- Die Daten-Modellierung induziert die Rechenstrukturen (also: Moduln) für die Datentypen **string** und **real** (Bezeichnung dieser Moduln: *STRING*, *REAL*).
- Weitere mögliche Moduln:

  WORTDARST: Erzeugung der Wortdarstellung zum Zahlungsbetrag;
  TRENNEN: Grammatikalisch korrekte Trennung von Wortdarstellungen;
  QUITTUNG: Einsetzen der Eingaben
  und Erstellung der endgültigen Fassung der Quittung.

- Schematische Darstellung der Strukturierung des Gesamtproblems:



  (Ein Pfeil von Modul M zu Modul M' bedeutet dabei die Verwendung von Bestandteilen der Schnittstelle von M' in M.)

- Die Schnittstellen von *STRING* und *REAL* sind **string**, **real** und die zugehörigen Basisfunktionen. Die Schnittstelle von *WORTDARST* bildet eine Funktion (in Pseudocode-Notation)

$konvert =$ **function**$(x :$ **real**$)$ **string** :
pre $x \geq 0$
result $konvert(x) =$ Wortdarstellung von x
(Cent-Anteile werden nicht berücksichtigt).

In dieser Festlegung wird (neben der Datenmodellierung) zunächst nur die „Wirkung“ von $konvert$ beschrieben (***Spezifikation*** von $konvert$).

(Die Schnittstellen der übrigen Moduln seien hier nicht weiter diskutiert.)

Vorteile der Modularisierung

- Allgemein: Strukturierung komplexer Programmsysteme.
- ***Abkapselung***: Bei der Entwicklung von (Teil-) Algorithmen eines Moduls muss man sich um die tatsächliche Realisierung (***Implementierung***) der Schnittstellen anderer Moduln nicht kümmern. Sie werden nur gemäß ihrer Spezifikation verwendet.

  Umgekehrt können Implementierungsdetails (z.B. Hilfsfunktionen), wenn sie nicht zur Schnittstelle gehören, außerhalb eines Moduls auch gar nicht verwendet werden. Sie können insbesondere ohne Einfluss nach außen geändert werden.
- Änderungen und/oder Ergänzungen am Gesamtsystem sind überschaubar durchführbar, da sie typischerweise nur einzelne Moduln betreffen oder nur das Hinzufügen neuer Moduln erfordern.
- Die konkreten Programmierungen in den einzelnen Moduln können unabhängig von einander durchgeführt werden.

Arten der Modularisierung

Eine vorteilhafte Modularisierung eines Gesamtproblems kann unterschiedlichen Leitlinien folgen:

- Zusammenfassung von Algorithmen, die ein gewisses Teilproblem lösen (***problemorientierte*** Modularisierung).
- Zusammenfassung von Algorithmen, die sich auf bestimmte Datentypen oder einzelne Daten beziehen (***datenorientierte*** Modularisierung; Beispiele: Rechenstrukturen für Datentypen gemäß Kapitel 4).
- Zusammenfassung von Algorithmen, die „verwandte“ Aufgaben lösen (***funktionsorientierte*** Modularisierung; Beispiel: Statistik-„Bibliothek“).
- Modularisierung unter Verwendung bereits vorhandener Moduln.

Programmentwicklung

Bei umfangreichen Aufgabenstellungen wird sich die endgültige Modularisierung typischerweise schrittweise entwickeln. Erste Strukturierungsansätze müssen oder sollten sinnvollerweise in späteren Entwicklungsschritten verändert werden. Beispiele hierfür sind:

- Weitere Aufteilung eines Moduls in mehrere Moduln.
- Zusammenfassung bisher getrennter Moduln.
- Bestimmung zusätzlicher Bestandteile von Schnittstellen.
- Veränderung der Spezifikation bereits vorhandener Schnittstellen-Bestandteile.

Moduln in SML

- In SML können Moduln mit ihren Schnittstellen direkt beschrieben werden. Dies geschieht in ***Strukturdeklarationen*** für die Moduln und gesonderten ***Signaturdeklarationen*** zur Festlegung der Schnittstellen.
- Der Modul *WORTDARST* im Beispiel der Quittungs-Erstellung kann in SML wie folgt angegeben werden:

```
signature WORTDARSTSig =
sig
   val konvert : real -> string
end

structure WORTDARST : WORTDARSTSig =
struct
...
fun konvert x = ...
...
end
```

  Der Modul $WORTDARST$ enthält außer $konvert$ eventuell noch weitere (Hilfs-) Funktionen. $konvert$ ist einziger Bestandteil der Signatur $WORTDARSTSig$. (Beachte dabei die Verwendung von **val**!) In der Zeile

  structure $WORTDARST : WORTDARSTSig$

  werden die Bestandteile der Signatur (hier also nur $konvert$) als Schnittstelle des Moduls festgelegt.

- Eine Funktion der Schnittstelle eines Moduls M wird (außerhalb von M) in der Form

  $M.\langle$*Funktionsname*$\rangle$

  verwendet (***qualifizierter Zugriff***), im Beispiel etwa:

```
WORTDARST.konvert(218.37)
```

- Durch eine ***open-Deklaration*** der Form

  open M

  wird ein Modul M ***geöffnet***. Anschließend können die Bestandteile seiner Schnittstelle direkt mit ihrem Namen verwendet werden, im Beispiel etwa:

```
open WORTDARST
konvert(218.37)
```

Vorgegebene SML-Moduln

In jedem SML-System sind typischerweise bereits Moduln (in der ***SML Basis Library***) vordefiniert. Manchmal sind in diesen Moduln auch gewisse Standardfunktionen (vgl. Abschnitt 3.2) zusammengefasst.

5.2 Schrittweise Programmentwicklung

Das Verfeinerungsprinzip

- Die Modularisierung einer Gesamtaufgabe folgt einem allgemeinen Grundprinzip:
  - Zerlege eine komplexe Aufgabe (eventuell in mehreren Schritten) immer weiter in Teilaufgaben, bis diese so überschaubar geworden sind, dass man ihre Lösung „direkt“ angeben kann.

  (***Schrittweise Verfeinerung***.)
- Dieses Prinzip kann auch auf die Entwicklung der in Modul-Schnittstellen festgelegten (möglicherweise komplexen) Algorithmen angewendet werden.

Beispiel: Die Konvertierungsfunktion *konvert*

(Annahme zur Vereinfachung: Der Zahlungsbetrag ist mindestens 1 EUR und kleiner als 1000 EUR.)

Spezifikation:

```
fun konvert x = (* Wortdarstellung (string)
                   des EUR-Anteils des Rechnungsbetrags x (real);
                   dabei vorausgesetzt: 0 < x < 1000 *)
```

Schritt 1. Aufbau von $konvert$:

```
fun konvert x = konveur(floor x)
```

Spezifikation von $konveur$:

```
fun konveur n = (* Wortdarstellung von n (int);
                   dabei vorausgesetzt: 0 < n < 1000 *)
```

Schritt 2. Aufbau von $konveur$:

```
fun konveur n = hunderter(n)^letztezwei(n)
```

Spezifikation von *hunderter* und *letztezwei*:

```
fun hunderter n = (* Wortdarstellung des Hunderteranteils von n *)

fun letztezwei n = (* Wortdarstellung des durch die letzten zwei
                      Ziffern von n gegebenen Anteils *)
```

Schritt 3. Aufbau von *hunderter* und *letztezwei*:

```
fun hunderter n =
    if n<100 then ""
    else einer(n div 100)^"hundert"

fun letztezwei n =
    let val l = n mod 100
        val z = zehner(l)
        val e = einer(n mod 10)
    in if l=1  then "eins" else
       if l=11 then "elf" else
       if l=12 then "zwoelf" else
       if l=16 then "sechzehn" else
       if l=17 then "siebzehn" else
       if l<10 then e else
       if l>10 andalso l<20 then e^z else
       if l mod 10 = 0 then z
       else e^"und"^z
    end
```

Spezifikation von *einer* und *zehner*:

```
fun einer k = (* Wortdarstellung von k;
                 dabei vorausgesetzt: 0 <= k <= 9 *)

fun zehner l = (* Wortdarstellung des Zehneranteils von l;
                  dabei vorausgesetzt: 0 <= l <= 99 *)
```

Schritt 4. Realisierung von *einer* und *zehner*:

```
fun einer 0 = ""
  | einer 1 = "ein"
  | einer 2 = "zwei"
  | einer 3 = "drei"
  | einer 4 = "vier"
  | einer 5 = "fuenf"
  | einer 6 = "sechs"
  | einer 7 = "sieben"
  | einer 8 = "acht"
  | einer 9 = "neun"

fun zehner l =
    let val s = l div 10
    in if s=0 then "" else
       if s=1 then "zehn" else
```

```
        if s=2 then "zwanzig" else
        if s=3 then "dreissig" else
        if s=4 then "vierzig" else
        if s=5 then "fuenfzig" else
        if s=6 then "sechzig" else
        if s=7 then "siebzig" else
        if s=8 then "achtzig"
        else "neunzig"
    end
```

Zusammenfassung. Alle angegebenen Funktionen gehören zum Modul *WORTDARST*. Dieser hat damit folgende Gestalt:

```
signature WORTDARSTSig =
sig
   val konvert : real -> string
end

structure WORTDARST : WORTDARSTSig =
struct
   fun einer ...
   fun zehner ...
   fun hunderter ...
   fun letztezwei ...
   fun konveur ...
   fun konvert ...
end
```

5.3 Unterordnung von Algorithmen

Lokale Funktionsdeklarationen (vgl. Abschnitt 2.1)

- In Funktionsrümpfen können in einem let-Term auch lokale (***untergeordnete***) Funktionen deklariert werden:

  ...
  let
    ...
    fun f ...
    ...
  in
  ...
  end

- Beispiel (vgl. Abschnitt 4.4):

```
- fun inssort nil     = nil
    | inssort (x::xt) =
        let
          fun insertel (a:int,nil)   = [a]
            | insertel (a:int,x::xt) = if a <= x then a::x::xt
                                       else x::insertel(a,xt)
        in
          insertel(x,inssort(xt))
        end
val inssort = fn : int list -> int list
```

- Auf diese Weise erhält man eine hierarchische Struktur von (Teil-) Algorithmen (***Blockstruktur***; die einzelnen Funktionen sowie das „Gesamtprogramm“ heißen auch ***Blöcke***).

Bemerkung

let-Terme haben die Gestalt **let** ... **in** t **end**, wobei t ein Term ist. SML bietet auch die Möglichkeit, lokale Deklarationen in anderen Deklarationen anzugeben. Dies geschieht in der Form

local ... **in** D **end**,

z.B.:

```
local
  fun insertel (a:int,nil)   = [a]
    | insertel (a:int,x::xt) = if a <= x then a::x::xt
                               else x::insertel(a,xt)
in
  fun inssort nil     = nil
    | inssort (x::xt) = insertel(x,inssort(xt))
end
```

Parameterunterdrückung

- Die in einer Funktion vorkommenden Größen sind auch innerhalb einer untergeordneten Funktion verwendbar. Parameter der untergeordneten Funktion, für die beim Aufruf solche Größen eingesetzt werden, können vermieden (***unterdrückt***) werden. Die Größen werden dann in der untergeordneten Funktion als ***globale*** Größen direkt verwendet.

- Beispiel (vgl. Abschnitt 4.5):
  Funktion $enthalten1$ mit Unterordnung der Funktion $enthallg$:

```
fun enthalten1(x,a) =
      let
        fun enthallg(x,k,a) =
              if k > dim(x) then false
              else a=get(x,k) orelse enthallg(x,k+1,a)
      in
        enthallg(x,1,a)
      end
```

Unterdrückung der Parameter x und a in $enthallg$:

```
fun enthalten1'(x,a) =
      let
        fun enthallg'(k) =
              if k > dim(x) then false
              else a=get(x,k) orelse enthallg'(k+1)
      in
        enthallg'(1)
      end
```

Gültigkeitsbereiche von Namen

- In untergeordneten Funktionen können Größen gleiche Namen wie in übergeordneten Funktionen haben. Die Verwendung der entsprechenden globalen Größe ist in der untergeordneten Funktion dann nicht möglich, ihr Name ist ***verschattet***.
- Der Bereich eines Programms, in dem eine Größe unter ihrem Namen verwendbar ist, heißt ***Gültigkeitsbereich*** des Namens und besteht allgemein aus dem Block B (***Bindungsbereich***), in dem die Größe eingeführt ist, abzüglich aller Blöcke, die in B enthalten sind und in denen eine Größe mit gleichem Namen eingeführt ist.
- Bemerkung:
  Bei verschatteten Namen ist der Begriff der Umgebung und das Konzept der Termauswertung (vgl. Abschnitt 3.4) entsprechend anzupassen.

5.4 Datenstrukturen und Modularisierung

Rechenstrukturen als Moduln

- Die Verwendung problemorientierter (nicht direkt verfügbarer) Datenstrukturen (vgl. Kapitel 4) lässt sich dem Modularisierungs-Prinzip unterordnen: Rechenstrukturen für entsprechende Datentypen können als (datenorientierte) Moduln konzipiert werden.

- Methodik (vgl. Abschnitte 4.1 und 5.1):
  - Die Spezifikation der Modul-Schnittstellen (d.h. in diesem Fall: der Datentypen und Basisfunktionen der Rechenstruktur) bedeutet die Festlegung des jeweiligen abstrakten Datentyps.
  - Die (bei der Verwendung der Rechenstruktur davon unabhängige) Implementierung der Schnittstellen bestimmt den zugehörigen konkreten Datentyp.

Beispiele (SML-Implementierungen)

1. Reihungen (vgl. Abschnitt 4.5):

```
signature VECTSig =
sig
   type 'a vect                    (* 1 *)
   exception wrongindex            (* 2 *)
   val dim    : 'a vect -> int
   val init   : int * 'a  -> 'a vect
   val get    : 'a vect * int -> 'a
   val update : 'a vect * int * 'a -> 'a vect
end

structure VECT : VECTSig =
struct
   type 'a vect = 'a list
   exception wrongindex
   fun dim x         = length x
   fun init(n,a)     = if n < 0 then raise wrongindex
                       else if n = 0 then nil
                            else a::init(n-1,a)
   fun get(x,i)      = if i < 1 orelse i > length(x)
                       then raise wrongindex
                       else if i = 1 then hd(x)
                            else get(tl(x),i-1)
   fun update(x,i,a) = if i < 1 orelse i > length(x)
                       then raise wrongindex
                       else if i = 1 then a::tl(x)
                            else hd(x)::update(tl(x),i-1,a)
end
```

Zu $(*\ 1\ *)$: In der Signatur *VECTSig* wird der Typ $'a$ **vect** in der angegebenen Form als zur Schnittstelle gehörig angegeben. In *VECT* wird durch eine Typdeklaration die Realisierung von $'a$ **vect** als $'a$ **list** festgelegt.

Zu $(*\ 2\ *)$: Moduln können auch Ausnahmen enthalten. Sie werden innerhalb der Strukturdeklaration deklariert und können auch zur Schnittstelle hinzugenommen werden.

2. Binärbäume (vgl. Abschnitt 4.6):

```
signature BINTREESig =
sig
   type 'a bintree
   exception emptytree
   val empty   : 'a bintree
   val build   : 'a * 'a bintree * 'a bintree -> 'a bintree
   val root    : 'a bintree -> 'a
   val left    : 'a bintree -> 'a bintree
   val right   : 'a bintree -> 'a bintree
   val isempty : 'a bintree -> bool
end

structure BINTREE : BINTREESig =
struct
   datatype 'a bintree = empty |
                         build of 'a * 'a bintree * 'a bintree
                                                        (* 1 *)
   exception emptytree
   fun root empty             = raise emptytree
     | root (build(x,_,_))    = x
   fun left empty             = raise emptytree
     | left (build(_,xl,_))   = xl
   fun right empty            = raise emptytree
     | right (build(_,_,xr))  = xr
   fun isempty empty          = true
     | isempty (build(_,_,_)) = false
end
```

Zu $(*\ 1\ *)$: Der Typ $'a$ **bintree** wird durch eine (rekursive!) datatype-Deklaration realisiert. $'a$ **bintree** ist somit ein Variantentyp mit den beiden Konstruktoren *empty* (Konstante) und *build* (als Injektion) von der Art

$$build : 'a * 'a\ \textbf{bintree} * 'a\ \textbf{bintree} \rightarrow 'a\ \textbf{bintree}.$$

3. Stapel (vgl. Abschnitt 4.4):

(Zur Unterscheidung:

emptyst	statt	*nil*,
push	statt	::,
top	statt	*hd*,
pop	statt	*tl*,
isemptyst	statt	*null*.)

```
signature STACKSig =
sig
   type 'a stack
   exception emptystack
   val emptyst   : 'a stack
   val push     : 'a * 'a stack -> 'a stack
   val top      : 'a stack -> 'a
```

```
   val pop      : 'a stack -> 'a stack
   val isemptyst : 'a stack -> bool
end

structure STACK : STACKSig =
struct
   type 'a stack = 'a list
   exception emptystack
   val emptyst     = nil
   fun push(y,s)   = y :: s
   fun top nil     = raise emptystack
     | top (x::_)  = x
   fun pop nil     = raise emptystack
     | pop (_::xt) = xt
   val isemptyst   = null
end
```

Anwendungsspezifische Datentypen

Moduln wie *VECT*, *BINTREE*, ... beschreiben grundlegende Rechenstrukturen, die in vielen verschiedenen Anwendungen benutzt werden. In gleicher Weise lassen sich auch Rechenstrukturen behandeln, die Datentypen und Funktionen für spezielle Anwendungen bereitstellen.

Beispiel:

Für die Programmierung einer Waage wird eine Rechenstruktur mit den Datentypen $preis$ und $gewicht$ (beide **real**, bei $preis$ mit maximal 2 Dezimalstellen) und den Basisfunktionen $plus$ (Addition zweier Preise) und mal (multipliziert ein Gewicht mit einem Preis und liefert einen (gegebenenfalls aufgerundeten) Preis) benötigt.

Ein passender Modul:

```
signature PREISSig =
sig
   type preis
   type gewicht
   val plus : preis * preis -> preis
   val mal  : gewicht * preis -> preis
end

structure PREIS : PREISSig =
struct
   type preis = real
   type gewicht = real
   fun plus(x : preis,y) = x + y
   fun mal(g,x)  = real(ceil(g * x * 100.0))/100.0
end
```

Anwendungsbeispiel (Berechnung des Preises von 300 g einer Ware mit Kilopreis 8.35 und 1.278 kg einer Ware mit Kilopreis 4.99):

```
- open PREIS
  plus(mal(0.3,8.35),mal(1.278,4.99))
val it = 8.89 : preis
```

Formale Modul-Spezifikationen

- Die Bedeutung der Konstanten und Funktionszeichen einer Modul-Schnittstelle kann oft präziser als durch verbale Beschreibungen auch in einer vollständig formalen Weise (***axiomatisch***) durch die Angabe der gewünschten charakteristischen Eigenschaften (***Axiome***) spezifiziert werden. Dies ist insbesondere bei der Spezifikation von Moduln für Datenstrukturen eine typische Technik.

- Beispiel: Stapel

  Verbale Spezifikation (der Schnittstelle) des Moduls *STACK*:

  – $emptyst$ ist der leere Stapel,
  – $push$ fügt ein Element zum Stapel hinzu,
  – (usw.)

  Die durch $emptyst$, $push$, top, ... bezeichneten Schnittstellen-Elemente sollen eine Reihe von charakteristischen Eigenschaften erfüllen. Diese können zur Vervollständigung der Signaturdeklaration von *STACKSig* in dieser (als Kommentare) angegeben werden:

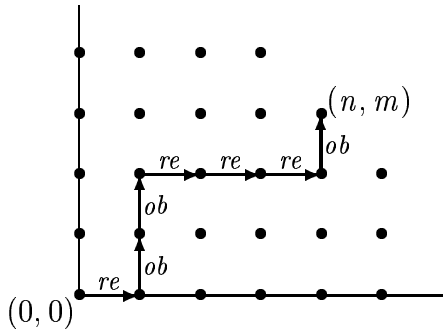
```
signature STACKSig =
sig
   type 'a stack
   exception emptystack
   val emptyst   : 'a stack
   val push      : 'a * 'a stack -> 'a stack
   val top       : 'a stack -> 'a
   val pop       : 'a stack -> 'a stack
   val isemptyst : 'a stack -> bool
(* Axiome:
   isemptyst(emptyst) = true,
   isemptyst(push(x,s)) = false,
   top(emptyst) = emptystack  (Ausnahme),
   top(push(x,s)) = x,
   pop(emptyst) = emptystack  (Ausnahme),
   pop(push(x,s)) = s *)
end
```

- Für die Programmiermethodik bedeutet dies, dass die in der zugehörigen Strukturdeklaration definierten Funktionen diese Axiome erfüllen müssen.

5.5 Modellierung und Implementierung

Ein Anwendungsbeispiel

Gegeben sei ein Gitterpunkt (n, m), $n, m \in \mathbb{N}_0$ der xy-Ebene. Ein Weg vom Ursprung $(0, 0)$ zum Punkt (n, m) ist eine Folge von „Schritten“ re bzw. ob, die vom Ursprung zu (n, m) führen. Dabei bedeutet re jeweils den Schritt von einem Punkt (k, l) „nach rechts“ zum Punkt $(k + 1, l)$ und ob den Schritt von (k, l) „nach oben“ zu $(k, l + 1)$:



Zu bestimmen ist die Gesamtheit aller möglichen Wege von $(0, 0)$ zu (n, m).

Lösung unter Verwendung direkt verfügbarer Datentypen

Modellierung (und Realisierung in SML):

Punkte (n, m):	**nat** $*$ **nat**,
Schritte:	Aufzählungstyp, bestehend aus re und ob,
Wege:	Listen von Schritten,
Gesamtheit aller Wege:	Liste von Wegen.

Algorithmus (unter Verwendung der als Standardfunktion verfügbaren Funktion map):

```
datatype schritt = re | ob
exception negArg
fun AlleWege(n,m) =
    let
       fun verl_re(w) = w @ [re]   (* Weg verlaengern um re *)
       fun verl_ob(w) = w @ [ob]   (* Weg verlaengern um ob *)
    in
       if n<0 orelse m<0 then raise negArg
       else if n=0 then
                    if      m=0 then nil
                    else if m=1 then [[ob]]
                    else             map(verl_ob)(AlleWege(0,m-1))
            else if n=1 andalso m=0
                    then             [[re]]
            else if n>1 andalso m=0
                    then             map(verl_re)(AlleWege(n-1,0))
            else                     map(verl_re)(AlleWege(n-1,m)) @
                                     map(verl_ob)(AlleWege(n,m-1))
    end
```

Lösung mit einem anwendungsspezifischen abstrakten Datentyp

Modellierung der Gesamtheit aller Wege:
Als „Menge“ von Wegen mit den für den Algorithmus benötigten Basisfunktionen.
(Schritte und Wege nicht konkret festgelegt, Punkte wie bisher.)

Spezifikation der Schnittstelle der entsprechenden Rechenstruktur *AWSET*:

```
signature AWSETSig =
sig
   type schritt
   type weg
   type wegemenge
   val re : schritt
   val ob : schritt
   val einschritt  : schritt -> weg
                           (* Weg bestehend aus einem Schritt *)
   val keinewege   : wegemenge
                           (* Leere Wegemenge *)
   val einweg      : weg -> wegemenge
                           (* Wegemenge bestehend aus einem Weg *)
   val unionwege   : wegemenge * wegemenge -> wegemenge
                           (* Vereinigung zweier Wegemengen *)
   val verl_rechts : wegemenge -> wegemenge
                           (* Verlaengerung aller Wege um re *)
   val verl_oben   : wegemenge -> wegemenge
                           (* Verlaengerung aller Wege um ob *)
end
```

Algorithmus unter Verwendung dieser Schnittstelle:

```
exception negArg
fun AlleWege'(n,m) =
       if n<0 orelse m<0 then raise negArg
       else if n=0 then
                   if      m=0 then keinewege
                   else if m=1 then einweg(einschritt(ob))
                   else             verl_oben(AlleWege'(0,m-1))
            else if n=1 andalso m=0 then einweg(einschritt(re))
            else if n>1 andalso m=0 then
                                     verl_rechts(AlleWege'(n-1,0))
            else unionwege(verl_rechts(AlleWege'(n-1,m)),
                           verl_oben(AlleWege'(n,m-1)))
```

Implementierungen von *AWSET*

- *AWSET* kann dadurch implementiert werden, dass der Typ $wegemenge$ als Listentyp (und $schritt$ und weg wie oben) realisiert werden:

```
structure AWSET : AWSETSig =
struct
   datatype schritt = re | ob
```

```
   type weg = schritt list
   type wegemenge = weg list
   fun einschritt(s) = [s]
   val keinewege = nil
   fun einweg(w) = [w]
   fun unionwege(x,y) = x @ y
   fun verl_rechts(x) = map(fn w => w @ [re])(x)
   fun verl_oben(x) = map(fn w => w @ [ob])(x)
      (* Deklarationen fuer re und ob nicht noetig,
         durch Deklaration von schritt erledigt *)
end
```

(Auf „Implementierungsebene“ sind bei dieser Realisierung die Funktionen $AlleWege$ und $AlleWege'$ (i.w.) gleich.)

- Es sind aber auch andere Implementierungen möglich, z.B. (eine Menge (von Wegen) als ***charakteristische Funktion***):

```
structure AWSET : AWSETSig =
struct
   datatype schritt = re | ob
   type weg = schritt list
   type wegemenge = weg -> bool
   fun einschritt(s) = [s]
   val keinewege = fn w => false
   fun einweg(w) = fn v => v=w
   fun unionwege(x,y) = fn w => x(w) orelse y(w)
       (* Hilfsfunktionen: *)
       fun front [u]  = nil
         | front (v::vt) = v::front(vt)
             (* front bestimmt den Anfang einer Liste ohne ihr
                letztes Element *)
       fun last [u]  = u
         | last (_::vt) = last(vt)
   fun verl_rechts(x) = fn w => not(null(w)) andalso x(front(w))
                                              andalso last(w)=re
   fun verl_oben(x) = fn w => not(null(w)) andalso x(front(w))
                                            andalso last(w)=ob
end
```

Die Rechenstruktur der endlichen Mengen

- Der (grundlegende) Datentyp *typ* **set** enthält endliche Mengen $\{a_1, \ldots, a_n\}$ von Elementen $a_1, \ldots, a_n$ des Typs *typ* (ohne irgendeine Anordnungsstruktur wie bei Listen, Reihungen oder Binärbäumen; eine solche Struktur ist in verschiedenen Anwendungen nicht problemrelevant).

- Definition der Signatur (mit axiomatischer Spezifikation) als SML-Signaturdeklaration (die gewünschte Rechenstruktur ist in SML nicht direkt verfügbar):

```
signature SETSig =
sig
   type ''a set   (* Beachte: ''a als Gleichheitstyp *)
   exception E
   val emptyset   : ''a set     (* Leere Menge *)
   val insert     : ''a * ''a set -> ''a set
                                (* Einfuegen eines Elements *)
   val delete     : ''a * ''a set -> ''a set
                                (* Wegnehmen eines Elements *)
   val any        : ''a set -> ''a
                                (* Zugriff auf ein Element *)
   val member     : ''a * ''a set -> bool    (* Enthaltensein *)
   val isemptyset : ''a set -> bool    (* Leersein *)
(* Axiome:
   isemptyset(emptyset) = true,
   isemptyset(insert(a,x)) = false,
   insert(b,insert(a,x)) = insert(a,insert(b,x)),
   insert(a,insert(a,x)) = insert(a,x),
   delete(a,emptyset) = emptyset,
   delete(a,insert(a,x)) = delete(a,x),
   delete(a,insert(b,x)) = insert(b,delete(a,x)),   falls a <> b,
   member(a,emptyset) = false,
   member(a,insert(a,x)) = true,
   member(a,insert(b,x)) = member(a,x),   falls a <> b,
   any(emptyset) = E     (Ausnahme),
   member(any(insert(a,x)),insert(a,x)) = true.  *)
end
```

- Die (***Auswahl-***) Funktion *any* greift auf ein Element der Menge zu. Die Spezifikation legt nicht fest, welches Element es ist. Dies ist bei der Verwendung dieser Rechenstruktur typischerweise auch irrelevant.

- Für die Rechenstruktur *SET* können Grundalgorithmen definiert werden, z.B.:

  1. Vereinigung zweier Mengen

     ```
     fun union(x,y) = if isemptyset(x) then y
                      else let
                              val a = any(x)
                           in
                              union(delete(a,x),insert(a,y))
                           end
     ```

  2. Anwendung einer Funktion auf alle Elemente einer endlichen Menge (vgl. *map*)

     ```
     fun setmap f x = if isemptyset(x) then emptyset
                      else let
                              val a = any(x)
                           in
                              insert(f(a),setmap(f)(delete(a,x)))
                           end
     ```

- *SET* kann in SML auf verschiedene Weisen implementiert werden, z.B. (analog zu oben) dadurch, dass Mengen durch Listen realisiert werden:

```
structure SET : SETSig =
struct
   type ''a set = ''a list
   exception E
   val emptyset = nil
   val isemptyset = null
   fun member(a,nil)   = false
     | member(a,x::xt) = a=x orelse member(a,xt)
   fun insert(a,x) = if member(a,x) then x else a::x
   fun delete(a,nil)   = nil
     | delete(a,x::xt) = if a=x then xt
                         else x :: delete(a,xt)
   fun any nil    = raise E
     | any (x::_) = x
end
```

Verwendung von *SET* zur Implementierung von *AWSET* (mit $union$ und $setmap$)

```
structure AWSET : AWSETSig =
struct
   datatype schritt = re | ob
   type weg = schritt list
   type wegemenge = weg set
   fun einschritt(s) = [s]
   val keinewege = emptyset
   fun einweg(w) = insert(w,emptyset)
   fun unionwege(x,y) = union(x,y)
   fun verl_rechts(x) = setmap(fn w => w @ [re])(x)
   fun verl_oben(x) = setmap(fn w => w @ [ob])(x)
end
```

Lösung des Anwendungsbeispiels mit *SET*

Modellierung der Gesamtheit aller Wege: Als $schritt$ **list set**.

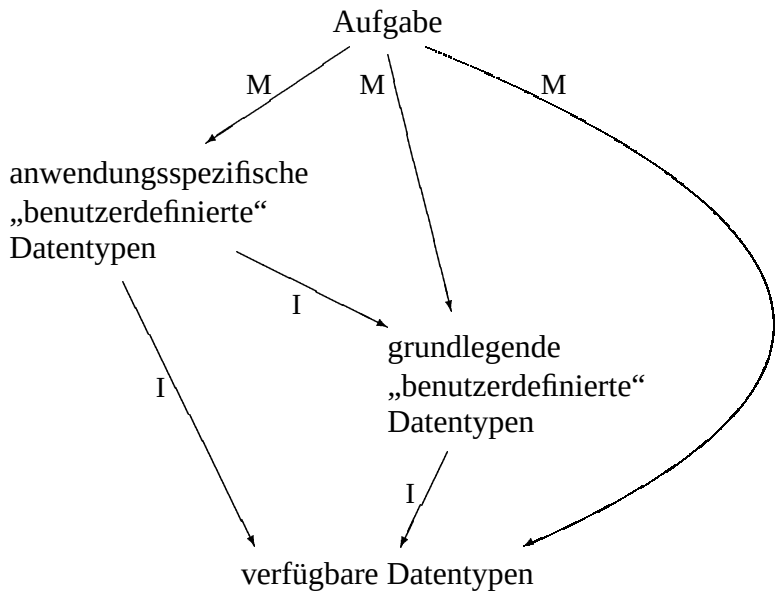
Algorithmus (unter Verwendung von $union$ und $setmap$):

```
datatype schritt = re | ob
exception negArg
fun AlleWege(n,m) =
    let
       fun verl_re(w) = w @ [re]
       fun verl_ob(w) = w @ [ob]
    in
```

```
    if n<0 orelse m<0 then raise negArg
    else if n=0 then
                if      m=0 then emptyset
                else if m=1 then insert([ob],emptyset)
                else             setmap(verl_ob)(AlleWege(0,m-1))
         else if n=1 andalso m=0 then insert([re],emptyset)
         else if n>1 andalso m=0 then
                                  setmap(verl_re)(AlleWege(n-1,0))
         else union(setmap(verl_re)(AlleWege(n-1,m)),
                    setmap(verl_ob)(AlleWege(n,m-1)))
end
```

Zusammenfassung

Für die Modellierung und Implementierung (des Anwendungsbeispiels) ergeben sich folgende typischen Möglichkeiten:



M: Modellierung I: Implementierung

Kapitel 6
Effiziente Algorithmen

6.1 Effizienz und Komplexität

Begriffsbestimmungen

- Bei der Entwicklung eines Algorithmus ist auch dessen ***Effizienz*** ein wichtiges Ziel. Diese ist ein Maß für den „Aufwand“, den der Algorithmus bei seiner Ausführung (auf einer Rechenanlage) verursacht. Je geringer der Aufwand, umso ***effizienter*** ist der Algorithmus.
- Wesentliche Messgrößen der Effizienz:
  - Ausführungsdauer (***Rechenzeit***),
  - benötigter Speicherumfang (***Speicherplatz***).
- Die Effizienz eines Algorithmus hängt ab
  - vom Aufbau des Algorithmus,
  - von der Realisierung der algorithmischen Konzepte auf einer Rechenanlage, von konkreten Maschineneigenschaften.
- Die ***Komplexität*** eines Algorithmus ist sein inhärent durch seinen Aufbau bestimmter Ausführungsaufwand (in Abhängigkeit gewisser Eingabegrößen). Je geringer dieser Aufwand, umso geringer die Komplexität.
- Genauere Unterscheidung (gemäß obiger Unterteilung):
  - ***Zeitkomplexität***,
  - ***Platzkomplexität***.
- Zur „Messung“ der Zeitkomplexität (im Folgenden fast ausschließlich betrachtet) wird idealisierend angenommen, dass als ***elementar*** ausgezeichnete Auswertungsschritte eine Ausführungsdauer von einer gewissen Zeiteinheit haben. Die Komplexität ist dann bestimmt durch die Anzahl der auszuführenden elementaren Einzelschritte.

Beispiel

Die Funktion

```
fun sum nil     = 0
  | sum (x::xt) = x + sum(xt)
```

summiert alle Komponenten einer Liste ganzer Zahlen.

Beispiel-Auswertung von sum gemäß dem Substitutionsmodell (vgl. Abschnitt 3.4):

$$\begin{aligned} sum(3,8,2,4) &\rightsquigarrow 3 + sum(8,2,4) \\ &\rightsquigarrow 3 + (8 + sum(2,4)) \\ &\rightsquigarrow 3 + (8 + (2 + sum(4))) \\ &\rightsquigarrow 3 + (8 + (2 + (4 + sum(nil)))) \\ &\rightsquigarrow 3 + (8 + (2 + (4 + 0))) \\ &\rightsquigarrow 3 + (8 + (2 + 4)) \\ &\rightsquigarrow 3 + (8 + 6) \\ &\rightsquigarrow 3 + 14 \\ &\rightsquigarrow 17 \end{aligned}$$

Die Ersetzungsschritte können als die elementaren Auswertungsschritte („Steuerung“ der Rekursion und Ausführung von Basisfunktionen) angesehen werden. Das Beispiel erfordert 9 solche Schritte.

Allgemein gilt für die Anzahl $T(n)$ der Ersetzungsschritte (d.h. die Zeitkomplexität von sum) in Abhängigkeit von der Länge n der Liste, mit der sum aufgerufen wird:

$$T(n) = 2 \cdot n + 1.$$

Anwendungsabhängige Komplexität

Außer von der „Größe der Argumente“ kann die Komplexität eines Algorithmus auch noch von weiteren Eigenschaften der Argumente abhängen. Beispiel: Die Funktion

```
fun sum0 nil = 0
  | sum0 (0::_) = 0
  | sum0 (x::xt) = x + sum0(xt)
```

summiert alle Komponenten einer Liste ganzer Zahlen bis zur ersten auftretenden 0.

Beispiel-Auswertungen:

a) $sum0(0,8,2,4) \rightsquigarrow 0$

b) $$\begin{aligned} sum0(3,0,2,4) &\rightsquigarrow 3 + sum0(0,2,4) \\ &\rightsquigarrow 3 + 0 \\ &\rightsquigarrow 3 \end{aligned}$$

c) $sum0(3,8,2,4) \rightsquigarrow \ldots$ (wie bei sum)

Die Anzahl der Schritte hängt davon ab, ob und wo 0 in der Liste (erstmals) vorkommt. Begriffe hierfür:

Komplexität im schlechtesten Fall:
Maximale Komplexität aller möglichen Argumente („gleicher Größe“).

Komplexität im Mittel:
Durchschnittliche Komplexität unter der Annahme, dass bei oftmaligen Anwendungen des Algorithmus alle möglichen Argumente („gleicher Größe“) gleich häufig vorkommen.

Im Beispiel:

Zeitkomplexität im schlechtesten Fall:	$T_s(n) = 2 \cdot n + 1.$
Zeitkomplexität im Mittel:	$T_m(n) = n + 1.$

Die *O*-Notation (Mathematische Definition)

Seien $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x \geq 0\}$ und $f, g : \mathbb{N}_0 \to \mathbb{R}^+$. g ***ist von der Ordnung*** f (geschrieben: $g(n)$ ist $O(f(n))$ oder auch: $g(n) = O(f(n))$), wenn es $c, n_0 \in \mathbb{N}_0$ gibt, so dass für alle $n \geq n_0$ gilt:

$$g(n) \leq c \cdot f(n).$$

Beispiele: $T_1(n) = 2 \cdot n + 5$: $T_1(n)$ ist $O(n)$ (d.h. $O(f(n))$ mit $f(n) = n$).
$T_2(n) = n^2 + 3$: $T_2(n)$ ist $O(n^2)$ (aber nicht $O(n)$).

Größenordnungen von Komplexitäten

- Komplexitätsangaben der Art

  $T(n)$ ist $O(f(n))$

  beschreiben die Komplexität eines Algorithmus in der Größenordnung ihres Anwachsens mit wachsendem n (***asymptotische Komplexität***).

- Ist $T_1(n)$ von einer Ordnung $f(n)$, $T_2(n)$ jedoch nicht, so ist $T_2(n)$ eine höhere asymptotische Komplexität als $T_1(n)$.

- Einige typische asymptotische Komplexitäten (mit wachsender Höhe):

		z.B.: $T(n) =$
$O(1)$	(***konstant***)	100
$O(log(n))$	(***logarithmisch***)	$k \cdot log(n)$
$O(n)$	(***linear***)	$k_1 \cdot n + k_2$
$O(n^2)$	(***quadratisch***)	$k_1 \cdot n^2 + k_2 \cdot n + k_3$
$O(2^n)$	(***exponentiell***)	$2^n + n^{5000}$

Entwicklung effizienter Algorithmen

Eine möglichst niedrige Komplexität eines Algorithmus bedeutet hohe Effizienz.

Einige typische grundlegende (z.T. ineinander übergehende) Vorgehensweisen zur Entwicklung effizienter Algorithmen (oder zur Effizienz-Verbesserung bereits konzipierter Algorithmen) in diesem Sinne:

- Wahl günstiger Rekursionsformen,
- Wahl günstiger Datenstrukturen (und deren Implementierungen),
- Änderung der Algorithmusidee.

6.2 Repetitive Rekursion

Auswertung rekursiver Funktionen

- Die Auswertung eines Aufrufs einer rekursiven Funktion (veranschaulicht im Substitutionsmodell) folgt im Allgemeinen einem systematischen Schema von immer neuen Aufrufen und anschließender schrittweiser Durchführung der „angehäuften“ Berechnungen gemäß den eingesetzten Basisfunktionen (vgl. Abschnitt 3.4).

- Beispiel: Die Auswertung der Fakultätsfunktion

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

  verläuft für $n = 4$ wie folgt:

$$
\begin{array}{ll}
fak(4) \rightsquigarrow 4 * fak(3) & \left.\vphantom{\begin{array}{l}a\\a\\a\\a\\a\end{array}}\right\} \text{Aufruffolge} \\
\quad \rightsquigarrow 4 * (3 * fak(2)) & \\
\quad \rightsquigarrow 4 * (3 * (2 * fak(1))) & \\
\quad \rightsquigarrow 4 * (3 * (2 * (1 * fak(0)))) & \\
\quad \rightsquigarrow 4 * (3 * (2 * (1 * 1))) & \\
\quad \rightsquigarrow 4 * (3 * (2 * 1)) & \left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right\} \text{Tatsächliche Berechnungen} \\
\quad \rightsquigarrow 4 * (3 * 2) & \\
\quad \rightsquigarrow 4 * 6 & \\
\quad \rightsquigarrow 24 &
\end{array}
$$

- Die schrittweisen Berechnungen nach dem letzten Aufruf sind dadurch hervorgerufen, dass der rekursive Aufruf in der Definition der Funktion noch mit anderen Größen „weiterverarbeitet“ wird (im Beispiel: Multiplikation des rekursiven Aufrufs $fak(n-1)$ mit n). Ist Letzteres nicht der Fall (d.h. steht der rekursive Aufruf „allein“), so entfallen diese Berechnungen (***repetitive***, ***iterative***, ***endständige*** Rekursion).

  Beispiel: Die Funktion

```
fun potenz(m,n) =    (* m>1, n>0 *)
     if n=1 orelse m=n then true
     else if n mod m <> 0 then false
          else potenz(m,n div m)
```

  bestimmt, ob n eine ganzzahlige Potenz von m ist ($n = m^k$ mit $k \in \mathbb{N}_0$).

  Beispiel-Auswertung:

$$
\begin{array}{ll}
potenz(3,54) \rightsquigarrow potenz(3,18) & \left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right\} \text{nur Aufruffolge} \\
\quad \rightsquigarrow potenz(3,6) & \\
\quad \rightsquigarrow potenz(3,2) & \\
\quad \rightsquigarrow false &
\end{array}
$$

(Das Ergebnis des letzten Aufrufs ist bereits das Gesamtergebnis.)

Transformation von rekursiven Funktionen in repetitive Form

Gewisse Rekursionsformen lassen sich durch Einbettung in eine repetitive Form transformieren. Ein oft anwendbares Schema besteht darin, eine allgemeinere Funktion zu definieren, in der die Größen (im Fakultäts-Beispiel: n), die bei der Weiterverarbeitung mit dem rekursiven Aufruf zur Anwendung kommen, in zusätzlichen Parametern mitgeführt und diese mit der gewünschten Funktion (im Beispiel: fak) gemäß deren rekursiver Aufrufform „verknüpft“ (im Beispiel: multipliziert) werden.

Effizienzvergleiche (2 Beispiele)

1. Im Falle von fak wird die allgemeinere Funktion

   $fakallg$: **nat** $*$ **nat** $\rightarrow$ **nat**,
   $fakallg(n, k) = k * n!$

   definiert. Es gilt: $fak(n) = fakallg(n, 1)$.

   Berechnung von $fakallg$:

   ```
   fun fakallg(n,k) = if n=0 then k else fakallg(n-1,k*n)
   ```

   $fakallg$ ist repetitiv rekursiv. Beispiel-Auswertung:

   $$\begin{aligned} fakallg(4,1) &\rightsquigarrow fakallg(3, 1*4) \\ &\rightsquigarrow fakallg(3,4) \\ &\rightsquigarrow fakallg(2, 3*4) \\ &\rightsquigarrow fakallg(2,12) \\ &\rightsquigarrow fakallg(1, 2*12) \\ &\rightsquigarrow fakallg(1,24) \\ &\rightsquigarrow fakallg(0, 1*24) \\ &\rightsquigarrow fakallg(0,24) \\ &\rightsquigarrow 24 \end{aligned}$$

   Die Anzahl der Berechnungsschritte (einschließlich der auszuführenden Multiplikationen, die „direkt“ auf dem neuen Parameter k ***akkumuliert*** werden), ist gleich wie bei fak. Die Auswertung von $fakallg$ benötigt allerdings weniger Speicherplatz (zur „Aufbewahrung“ der noch „unerledigten“ Terme) und ist auf vielen Rechenanlagen auch effizienter zu realisieren (in Bezug auf die „Organisation“ der Rekursion).

2. Für eine Liste $y = (y_1, \ldots, y_n)$ ganzer Zahlen und eine ganze Zahl m sei

   $y \mathbin{++} m = (y_1 + m, y_2 + m, \ldots, y_n + m)$.

   ($++$ ist leicht mit Hilfe von map programmierbar.) Die unter Verwendung von $++$ formulierte Funktion

   ```
   fun f nil     = 1
     | f (x::xt) = (x+1) * f(xt ++ 1)
   ```

berechnet f : **int list** $\rightarrow$ **int** mit $f(x_1, \ldots, x_n) = (x_1 + 1) * (x_2 + 2) * \ldots * (x_n + n)$.

Beispiel-Auswertung:

$$\begin{aligned}
f(3,4,0,1) &\rightsquigarrow (3+1) * f((4,0,1) +\!\!+\ 1) \\
&\rightsquigarrow \ldots \rightsquigarrow 4 * f(5,1,2) && (1)\\
&\rightsquigarrow 4 * (5+1) * f((1,2) +\!\!+\ 1) \\
&\rightsquigarrow \ldots \rightsquigarrow 4 * 6 * f(2,3) && (2)\\
&\rightsquigarrow 4 * 6 * (2+1) * f((3) +\!\!+\ 1) \\
&\rightsquigarrow \ldots \rightsquigarrow 4 * 6 * 3 * f(4) && (3)\\
&\rightsquigarrow 4 * 6 * 3 * (4+1) * f(nil +\!\!+\ 1) \\
&\rightsquigarrow \ldots \rightsquigarrow 4 * 6 * 3 * 5 * f(nil) && (4)\\
&\rightsquigarrow 4 * 6 * 3 * 5 * 1 \\
&\rightsquigarrow \ldots \rightsquigarrow 360
\end{aligned}$$

Allgemein: Ist n die Länge der Liste, auf die f angewendet wird, so benötigt die Auswertung $n + 1$ Schritte bis zur Stelle (1) (da die Auswertung von $+\!\!+\ n - 1$ Additionen bedeutet), n Schritte von (1) nach (2), $n - 1$ Schritte von (2) nach (3) usw., schließlich 3 Schritte bis zum Aufruf $f(nil)$ und dann noch $n + 1$ Schritte bis zum Ende. Die Anzahl der Schritte ist also insgesamt

$$3 + 3 + 4 + 5 + \ldots + n + n + 1 + n + 1 = \tfrac{n*(n+1)}{2} + 2 * n + 2 = O(n^2),$$

die Zeitkomplexität von f ist somit $O(n^2)$.

Gemäß obigem Einbettungs-Schema sei *fallg* definiert durch:

fallg : **int list** $*$ **int** $\rightarrow$ **int**,
$fallg(x, m) = m * f(x)$.

Es gilt: $f(x) = fallg(x, 1)$.

Berechnung von *fallg*:

```
fun fallg(nil,m)    = m
  | fallg(x::xt,m) = fallg(xt ++ 1,m*(x+1))
```

fallg ist repetitiv rekursiv. Beispiel-Auswertung:

$$\begin{aligned}
fallg((3,4,0,1),1) &\rightsquigarrow fallg((4,0,1) +\!\!+\ 1, 1 * (3+1)) \\
&\rightsquigarrow \ldots \rightsquigarrow fallg((5,1,2),4) \\
&\rightsquigarrow fallg((1,2) +\!\!+\ 1, 4 * (5+1)) \\
&\rightsquigarrow \ldots \rightsquigarrow fallg((2,3),24) \\
&\rightsquigarrow fallg((3) +\!\!+\ 1, 24 * (2+1)) \\
&\rightsquigarrow \ldots \rightsquigarrow 360
\end{aligned}$$

Die Zeitkomplexität auch dieses Algorithmus ist $O(n^2)$. (Argumente wie bei f.) Der Effizienzgewinn ist von gleicher Art wie im Fakultäts-Beispiel.

Aber: Weitere Einbettung möglich zu

$fallg'$: **int list** $*$ **int** $*$ **int** $\rightarrow$ **int**,
$fallg'(x, m, k) = m * f(x +\!\!+ k)$.

(Auch die „Verarbeitung“ des Arguments von f mit $+\!\!+$ wird in einem eigenen Parameter mitgeführt.) Es gilt: $f(x) = fallg'(x, 1, 0)$.

Berechnung von $fallg'$:

```
fun fallg'(nil,m,k)    = m
  | fallg'(x::xt,m,k) = fallg'(xt,m*(x+k+1),k+1)
```

$fallg'$ ist wieder repetitiv rekursiv. Beispiel-Auswertung:

$$\begin{aligned}
fallg'((3,4,0,1),1,0) &\rightsquigarrow fallg'((4,0,1),1*(3+0+1),0+1)\\
&\rightsquigarrow \ldots \rightsquigarrow f((4,0,1),4,1)\\
&\rightsquigarrow fallg'((0,1),4*(4+1+1),1+1)\\
&\rightsquigarrow \ldots \rightsquigarrow fallg'((0,1),24,2)\\
&\rightsquigarrow fallg'((1),24*(0+2+1),2+1)\\
&\rightsquigarrow \ldots \rightsquigarrow fallg'((1),72,3)\\
&\rightsquigarrow fallg'(nil,72*(1+3+1),3+1)\\
&\rightsquigarrow \ldots \rightsquigarrow fallg'(nil,360,4)\\
&\rightsquigarrow 360
\end{aligned}$$

Allgemein: Die Anzahl der Berechnungsschritte und damit die Zeitkomplexität dieses Algorithmus ist $O(n)$. (Die Veränderung der Listenelemente entfällt, sie wird für den jeweiligen Schritt auf k akkumuliert.) $fallg'$ hat somit sogar eine niedrigere asymptotische Zeitkomplexität als f (neben den anderen Effizienzeffekten wie bei $fallg$).

Nicht-lineare Rekursionen

- Die im Vorangegangenen behandelten Rekursionsformen haben gemeinsam, dass in den Fallunterscheidungen im Rumpf der Funktion in jedem Fall höchstens ein (und in den Bedingungen der Fallunterscheidungen gar kein) rekursiver Aufruf vorkommt (***lineare Rekursion***). Allgemeinere (***nicht-lineare***) Rekursionsformen lassen sich nur in Einzelfällen in repetitive (und damit effizientere) Formen transformieren.

- Beispiel: Die Fibonacci-Funktion (vgl. Abschnitt 2.3)

```
fun fib n = if n=0 orelse n=1 then 1 else fib(n-1) + fib(n-2)
```

ist nicht-linear. Beispiel-Auswertung:

$$\begin{aligned}
fib(4) &\rightsquigarrow fib(3) + fib(2)\\
&\rightsquigarrow (fib(2) + fib(1)) + fib(2)\\
&\rightsquigarrow (fib(2) + fib(1)) + (fib(1) + fib(0))\\
&\rightsquigarrow ((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))\\
&\rightsquigarrow ((1 + fib(0)) + fib(1)) + (fib(1) + fib(0))\\
&\rightsquigarrow ((1 + 1) + fib(1)) + (fib(1) + fib(0))\\
&\rightsquigarrow \ldots
\end{aligned}$$

Allgemein: $fib(n)$ erzeugt („ungefähr“) 2^n rekursive Aufrufe (und ebenso viele anschließende Additionen), d.h.: fib hat exponentielle Zeitkomplexität.

Einbettung:

$$fiballg(n,k,j) = k * fib(n+1) + j * fib(n) \qquad (k,j \in \mathbb{N}_0).$$

Dann gilt: $fib(n) = fiballg(n,0,1)$.

Berechnung von $fiballg$ (repetitiv rekursiv):

```
fun fiballg(n,k,j) = if n=0 then k+j else fiballg(n-1,k+j,k)
```

Beispiel-Auswertung:

$$\begin{aligned} fib(4,0,1) &\rightsquigarrow fiballg(3,0+1,0) \\ &\rightsquigarrow fiballg(3,1,0) \\ &\rightsquigarrow fiballg(2,1+0,1) \\ &\rightsquigarrow fiballg(2,1,1) \\ &\rightsquigarrow fiballg(1,1+1,1) \\ &\rightsquigarrow fiballg(1,2,1) \\ &\rightsquigarrow fiballg(0,2+1,2) \\ &\rightsquigarrow fiballg(0,3,2) \\ &\rightsquigarrow 3+2 \\ &\rightsquigarrow 5 \end{aligned}$$

Allgemein: $fiballg$ hat lineare Zeitkomplexität, ist also erheblich effizienter als fib.

6.3 Effiziente Datenstrukturen

Beispiel: Das Suchproblem (vgl. Abschnitte 4.4, 4.5, 4.6)

In einem Datenbestand soll ein bestimmtes Datenelement gesucht werden.
(Einfache) Formulierung hier:

Gegeben: Multimenge x von ganzen Zahlen (mit n Elementen), $a \in$ **int**;
zu bestimmen: Ist a in x enthalten?

Triviale Lösung

Modellierung von x als Datentyp **int set** (falls x Menge ist) oder als ein analog definierbarer Datentyp (etwa: **int multiset**) für Multimengen:

```
fun suchen1 (x:multiset,a) = member(a,x)
```

Zeitkomplexität: konstant (1 Basisfunktion).

Aber: $member$ ist (auf heutigen Rechenanlagen) nicht als elementarer Verarbeitungsschritt verfügbar. Er erfordert eine Implementierung auf der Basis anderer Datenstrukturen (z.B. Listen, vgl. Abschnitt 5.5).

Lineares Suchen

Modellierung von x als Liste, Reihung oder Binärbaum (bzw. Implementierung von **int multiset** durch eine dieser Datenstrukturen): Suchalgorithmen $enthalten$ (Abschnitt 4.4), $enthalten1$ (Abschnitt 4.5), $enthalten2$ (Abschnitt 4.6).

Gemeinsame Grundidee: Untersuche der Reihe nach alle Elemente von x, bis a (gegebenenfalls) gefunden ist (***lineares Suchen***). Die Basisfunktionen der Datentypen können als elementare Verarbeitungsschritte angesehen werden. Beispiel: $enthalten1$ (jetzt als $suchen2$):

```
fun suchen2(x,a) =
     let
       fun enthallg(x,k,a) =  (* Vorbedingung: k >= 1 *)
            if k > dim(x) then false
            else a=get(x,k) orelse enthallg(x,k+1,a)
     in
       enthallg(x,1,a)
     end
```

Zeitkomplexität (im schlechtesten Fall und im Mittel): $O(n)$.

Sortierte Reihungen

Modellierung von x als sortierte Reihung (d.h. $x = (x_1, \ldots, x_n)$ mit $x_1 \leq x_2 \leq \ldots \leq x_n$): Suchen durch „Bisektion“ (***binäres Suchen***) möglich.

Idee:

- Bestimme „mittleres Element“ x_m von x.
- Falls $a = x_m$: a gefunden.
  Falls $a < x_m$: Suche weiter in $(x_1, \ldots, x_{m-1})$.
  Falls $a > x_m$: Suche weiter in $(x_{m+1}, \ldots, x_n)$.

Rekursiver Algorithmus ($suchen3$) durch Einbettung:

```
fun such3allg(x:int vect,a,l,r) =  (* Vorbedingung: 1<=l,r<=dim(x);
                                      Suchen von a in (x_l,...,x_r) *)
   let
     val m = (l+r) div 2
   in
     if l > r then false
     else if a < get(x,m) then such3allg(x,a,l,m-1)
     else if a > get(x,m) then such3allg(x,a,m+1,r)
                          else true
   end

fun suchen3(x:int vect,a) =  (* Vorbedingung: x sortiert *)
   such3allg(x,a,1,dim(x))
```

Zeitkomplexität: $O(log(n))$.

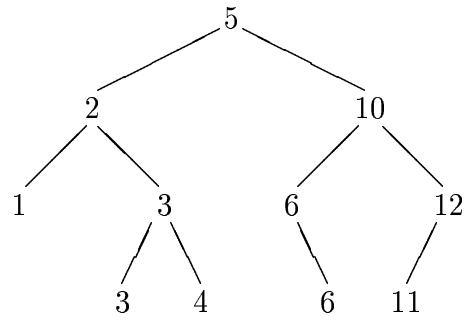
Veranschaulichung der Effizienzverbesserung:
Bei jedem „Suchschritt“ wird die Anzahl der noch zu durchsuchenden Daten halbiert. Beim linearen Suchen wird diese Anzahl nur immer um 1 verringert.

Binäre Suchbäume

Ein Binärbaum z (hier über **int**; analog über anderen Datenmengen) heißt ***binärer Suchbaum*** (***sortiert***), wenn für jeden Teilbaum (x, xl, xr) von z gilt:

Für alle Knoten u_1 von xl und alle Knoten u_2 von xr ist $u_1 \leq x \leq u_2$.

Beispiel:



(Es gilt: Ist z ein binärer Suchbaum, so ist $linsym(z)$ die sortierte Liste der Knoten von z.)

(Binäres) Suchen in binärem Suchbaum:

```
fun suchen4(x:int bintree,a) =  (* Vorbedingung:
                                   x ist binaerer Suchbaum *)
     if isempty(x) then false
     else if a < root(x) then suchen4(left(x),a)
     else if a > root(x) then suchen4(right(x),a)
                         else true
```

Zeitkomplexität: $O(log(n))$
(unter der Voraussetzung, dass x „möglichst ausgeglichen“ ist, d.h. dass für alle Teilbäume gilt: Der jeweilige linke und rechte Unterbaum haben (ungefähr) gleich viele Knoten).

Bemerkung

Datenbestände können auf viele weitere Arten modelliert (bzw. implementiert) werden. Meist sind nicht nur ein, sondern mehrere verschiedene Algorithmen – eventuell in unterschiedlicher Häufigkeit – anzuwenden. (Beispiel: In einer Kartei sollen Einträge gesucht, neue Einträge eingefügt, nicht mehr benötigte Einträge entfernt werden.) Die Auswahl einer geeigneten Datenstruktur muss verschiedene derartige Aspekte berücksichtigen und gegebenenfalls gegeneinander abwägen.

6.4 Ein effizientes Sortierverfahren

Das Sortierproblem (hier für Listen ganzer Zahlen)

Eine Liste ganzer Zahlen soll sortiert werden.

Algorithmus in den Abschnitten 4.4 und 5.3 (Sortieren durch Einfügen):

```
fun inssort nil     = nil
  | inssort (x::xt) =
      let
        fun insertel (a:int,nil)   = [a]
          | insertel (a:int,x::xt) = if a <= x then a::x::xt
                                     else x::insertel(a,xt)
      in
        insertel(x,inssort(xt))
      end
```

Komplexität von *inssort*

Beispiel-Auswertung:

$$
\begin{aligned}
inssort(3,7,1,4) &\rightsquigarrow insertel(3, inssort(7,1,4)) \\
&\rightsquigarrow insertel(3, insertel(7, inssort(1,4))) \\
&\rightsquigarrow \ldots \rightsquigarrow insertel(3, insertel(7, insertel(1, insertel(4, nil)))) \\
&\rightsquigarrow insertel(3, insertel(7, insertel(1, (4)))) \qquad (*) \\
&\rightsquigarrow \ldots \rightsquigarrow insertel(3, insertel(7, (1,4))) \\
&\rightsquigarrow \ldots \rightsquigarrow insertel(3, (1,4,7)) \\
&\rightsquigarrow \ldots \rightsquigarrow (1,3,4,7)
\end{aligned}
$$

Allgemein: Die Anzahl der Schritte bis $(*)$ ist $O(n)$. Ab $(*)$ müssen n Elemente in Listen wachsender Längen $1, 2, \ldots, n-1$ einsortiert werden. Dies erfordert (im schlechtesten Fall und im Mittel) eine Anzahl von Schritten, die proportional ist zu $1 + 2 \ldots + (n-1)$.

Die Zeitkomplexität von $inssort$ ist somit $O(n^2)$.

Sortieren durch Verschmelzen

```
(* Aufspalten: *)
fun split nil        = (nil,nil)
  | split (x::nil)   = ([x],nil)
  | split (x::y::xt) = let
                         val x_u = #1(split(xt))
                         val x_g = #2(split(xt))
                       in
                         (x::x_u,y::x_g)
                       end
```

```
(* Verschmelzen: *)
fun merge(xt,nil)      = xt
  | merge(nil,yt)      = yt
  | merge(x::xt,y::yt) = if x <= y then x::merge(xt,y::yt)
                                   else y::merge(x::xt,yt)
(* Sortieren: *)
fun mergesort nil        = nil
  | mergesort (x::nil)   = [x]
  | mergesort (x::y::xt) = let
                             val x_u = #1(split(x::y::xt))
                             val x_g = #2(split(x::y::xt))
                           in
                             merge(mergesort(x_u),mergesort(x_g))
                           end
```

Komplexität von *mergesort*

Beispiel-Auswertung: (Schreibe m für $merge$ und s für $mergesort$)

$$
\begin{aligned}
& s(13,9,2,15,7,3,9,6) \\
& \quad \rightsquigarrow \ldots \rightsquigarrow m(s(13,2,7,9), s(9,15,3,6)) && (1)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow m(m(s(13,7), s(2,9)), m(s(9,3), s(15,6))) && (2)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow m(m(m(s(13), s(7)), m(s(2), s(9))) \\
& \qquad\qquad m(m(s(9), s(3)), m(s(15), s(6)))) && (3)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow m(m(m((13), (7)), m((2), (9))) \\
& \qquad\qquad m(m((9), (3)), m((15), (6)))) && (4)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow m(m((7,13), (2,9)), m((3,9), (6,15))) && (5)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow m((2,7,9,13), (3,6,9,15)) && (6)\\
& \quad \rightsquigarrow \ldots \rightsquigarrow (2,3,6,7,9,9,13,15) && (7)
\end{aligned}
$$

Allgemein: Bei Auswertung von $mergesort(x)$ für eine Liste der Länge n ist die Anzahl der Schritte zu den Auswertungsstellen $(1), (2), \ldots, (7)$ jeweils $O(n)$. Bis zur Stelle (4) wird die Länge der zu betrachtenden Listen jeweils halbiert, anschließend jeweils verdoppelt. Die Anzahl der Stellen $(1), (2), \ldots, (7)$ ist also $O(log(n))$. (Beide Betrachtungen gelten für den schlechtesten Fall und im Mittel.)

Die Zeitkomplexität von $mergesort$ ist somit $O(n\, log(n))$ und damit geringer als die von $inssort$.

Kapitel 7
Denotationelle Semantik funktionaler Programme

7.1 Funktionen als Fixpunkte

Arten von Semantiken

- Die Termauswertung

  $$W : \text{Term} \mapsto \text{Wert}$$

  (vgl. Abschnitt 3.4) definiert eine (***operationelle***) Semantik (eines Teils) von SML (-Programmen).

- Operationelle Semantiken von Programmiersprachen: sehr „implementierungsnah“ (sie beschreiben den „Ablauf“ von elementaren Schritten bei der Programmausführung). Andere, weniger detaillierte Semantik-Arten:
  - ***Axiomatische*** Semantik:
    Die Bedeutung der einzelnen Sprachkonstrukte wird durch charakteristische Eigenschaften (ähnlich wie in Abschnitt 5.4) beschrieben. (Beispiel: siehe Kapitel 8.)
  - ***Denotationelle*** (***funktionale***) Semantik:
    Einem Programm wird als seine Bedeutung ein ***semantisches Modell*** zugeordnet, das (im wesentlichen) das funktionale (d.h. „Ein/Ausgabe“-) Verhalten des Programms festlegt (ohne die Details der „Zwischenschritte“).

Denotationelle Semantik von Funktionen

- Ein funktionales Programm ist eine Funktion, die syntaktisch durch eine Gleichung definiert ist und deren denotationelle Bedeutung durch die Lösung der Gleichung gegeben ist. (Wir betrachten der Einfachheit halber keine verschränkt rekursiven Funktionen.) Beispiel (mathematische Notation):

  $$g(n) = n + 3 \qquad \text{(genauer gemäß Abschnitt 2.1: } g = \lambda(n).n + 3\text{)}.$$

  (Eindeutige) Lösung für g: Funktion $n \mapsto n + 3$.

- Hauptproblem: Rekursive Funktionen, definiert durch Gleichungen der Art

  $$g(\ldots) = \ldots g(\ldots) \ldots .$$

  Beispiel: $g : \mathbb{N}_0 \to \mathbb{N}_0$,

  $$g(n) = \begin{cases} 1, & \text{falls } n = 0 \\ g(n+1) & \text{sonst.} \end{cases} \qquad (*)$$

  Diese Gleichung hat viele Lösungen für g: Jede Funktion

$$g_k(n) = \begin{cases} 1, & \text{falls } n = 0 \\ k & \text{sonst} \end{cases}$$

($k = 0, 1, 2, \ldots$) ist eine Lösung (erfüllt $(*)$), ebenso die (partielle) Funktion

$$g'(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Allgemein: Es muss festgelegt werden, welche Lösung einer rekursiven Funktionsdefinition (falls überhaupt Lösungen existieren) als deren Bedeutung gelten soll.

„undefiniert“ als Wert

Um alle Funktionen als total ansehen zu können, kann ein spezielles Datenelement ω (für „undefiniert“) eingeführt werden. Ist A eine Menge, so sei $A^\omega = A \cup \{\omega\}$.

Beispiel: g' (von oben) als totale Funktion:

$$g_\omega : \mathbb{N}_0 \to \mathbb{N}_0^\omega,$$
$$g_\omega(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \omega & \text{sonst.} \end{cases}$$

(g_ω ist Lösung der Gleichung $(*)$ für $g : \mathbb{N}_0 \to \mathbb{N}_0^\omega$.)

Fixpunkt-Funktionale

- Für jede Lösung $g : \mathbb{N}_0 \to \mathbb{N}_0^\omega$ der Gleichung

  $$g(n) = \begin{cases} 1, & \text{falls } n = 0 \\ g(n+1) & \text{sonst} \end{cases}$$

  gilt $G(g) = g$ (g ist ***Fixpunkt*** von G) für das ***Fixpunkt-Funktional***

  $$G : (\mathbb{N}_0 \to \mathbb{N}_0^\omega) \to (\mathbb{N}_0 \to \mathbb{N}_0^\omega),$$
  $$G(h) = \begin{cases} 1, & \text{falls } n = 0 \\ h(n+1) & \text{sonst.} \end{cases}$$

- Allgemeines Ziel: Bedeutung von (rekursiven) Funktionen als Fixpunkte geeigneter Funktionale.

  Fragen dabei:

  - Unter welchen Bedingungen existieren Fixpunkte?
  - Festlegung eines ausgezeichneten Fixpunkts, falls mehrere existieren.

  Antworten hierzu: (Mathematische) ***Fixpunkttheorie***.

7.2 Grundzüge der Fixpunkttheorie

Definition. Eine binäre Relation $\sqsubseteq$ auf einer Menge A (d.h. $\sqsubseteq \subseteq A \times A$) heißt ***partielle Ordnung*** (auf A), wenn für alle $x, y, z \in A$ gilt:

1. $x \sqsubseteq x$ (***Reflexivität***)
2. $x \sqsubseteq y, y \sqsubseteq x \Rightarrow x = y$ (***Antisymmetrie***)
3. $x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z$ (***Transitivität***)

Definition. Ein Element $\bot$ einer Menge A mit partieller Ordnung $\sqsubseteq$ mit

$$\bot \sqsubseteq x \qquad \text{für alle } x \in A$$

heißt ***kleinstes Element*** (***bottom***) von A.

Definition. Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge A. Eine nicht-leere Teilmenge B von A heißt ***Kette*** (in A), wenn für je zwei Elemente $x, y \in B$ gilt: $x \sqsubseteq y$ oder $y \sqsubseteq x$.

Definition. Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge A. Eine nicht-leere Teilmenge B von A heißt ***gerichtet***, wenn es für je zwei Elemente $x, y \in B$ ein $z \in B$ gibt mit $x \sqsubseteq z$ und $y \sqsubseteq z$.

Satz 1. Jede Kette in einer Menge A (mit einer partiellen Ordnung) ist gerichtet.

Vorbemerkung. Alle im Folgenden betrachteten Funktionen seien total.

Definition. Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge A. Eine Funktion $f : A \to A$ heißt ***monoton***, wenn für alle $x, y \in A$ gilt:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y).$$

Satz 2. Sei A eine Menge mit partieller Ordnung und kleinstem Element. Ist $f : A \to A$ eine monotone Funktion, so ist $\{f^i(\bot) \mid i \in \mathbb{N}_0\}$ eine Kette in A.

Definition. Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge A, $B \subseteq A$. Ein Element $z \in A$ heißt ***kleinste obere Schranke*** (***Supremum***) ***von*** B (Schreibweise: $sup\ B$), wenn gilt:

a) $x \sqsubseteq z$ für alle $x \in B$.

b) $x \sqsubseteq y$ für alle $x \in B \Rightarrow z \sqsubseteq y$.

Definition. Eine partielle Ordnung $\sqsubseteq$ auf einer Menge A heißt ***vollständig*** (und A heißt ***vollständig geordnet***, ***complete partial order***, ***cpo***), wenn es für jede gerichtete Teilmenge B von A ein Element $x \in A$ gibt mit $x = sup\ B$.

Definition. Sei A eine cpo. Eine Funktion $f : A \to A$ heißt (***ketten-***) ***stetig***, wenn für jede Kette B in A gilt: $sup\, f(B)$ existiert in A, und es ist $f(sup\ B) = sup\, f(B)$.
(Dabei: $f(B) = \{f(x) \mid x \in B\}$.)

Satz 3. Sei A eine cpo. Jede stetige Funktion $f : A \to A$ ist monoton.

Definition. Sei A eine cpo, $f : A \to A$ eine stetige Funktion. Ein Element $x \in A$ heißt ***kleinster Fixpunkt von*** f, wenn gilt:

$$f(x) = x \quad \text{und} \quad f(y) = y \ \Rightarrow\ x \sqsubseteq y.$$

(Es gilt: f hat höchstens einen kleinen Fixpunkt.)

Satz 4. (Fixpunktsatz von Kleene)
Sei A eine cpo mit kleinstem Element, $f : A \to A$ eine stetige Funktion. Das Element

$$x = sup\,\{f^i(\bot) \mid i \in \mathbb{N}_0\}$$

existiert in A und ist kleinster Fixpunkt von f. (Bezeichnung: $fix(f)$.)

7.3 Denotationelle Semantik für SML-Sprachelemente

Mit Hilfe der Ergebnisse von Abschnitt 7.2 kann einem funktionalen (SML-) Programm eine denotationelle Semantik (im Sinne von Abschnitt 7.1) gegeben werden. Ein Abriss der Vorgehensweise ist wie folgt.

Erweiterung von Datentypen

Für jeden Datentyp typ sei ein Element $\omega \notin typ$ (vgl. Abschnitt 7.1) ausgezeichnet. Die Menge $typ^\omega = typ \cup \{\omega\}$ mit der ***flachen Ordnung***

$$x \sqsubseteq y \Leftrightarrow x = \omega \text{ oder } x = y$$

bildet eine cpo mit ω als kleinstem Element.

Die Semantik von Termen

Die einem Term t in der denotationellen Semantik zugeordnete Bedeutung wird mit $[\![t]\!]$ bezeichnet.

- Für Terme t, die keine Funktionsabstraktionen, keine Funktionsanwendungen und keine Namen von Funktionen sind, ist $[\![t]\!]$ definiert gemäß der Auswertungsfunktion W aus Abschnitt 3.4 (bezüglich einer gegebenen Umgebung).

- Ist t eine Funktionsabstraktion der Gestalt

  fn $x \Rightarrow t'$

  und vom Typ $typ_1 \to typ_2$, so ist $[\![t]\!]$ die Funktion

  $[\![t]\!] : typ_1 \to typ_2^\omega$,
  $[\![t]\!](a) = [\![t']\!]$, wobei $[\![t']\!]$ bestimmt wird bezüglich der Umgebung, die sich aus der aktuellen Umgebung durch Hinzunahme der Bindung <x,a> ergibt.

- Ist t eine Funktionsanwendung $u(v_1, \ldots, v_n)$, so ist

  $$[\![t]\!] = \begin{cases} [\![u]\!]([\![v_1]\!], \ldots, [\![v_2]\!]), & \text{falls dieser Wert definiert ist} \\ \omega & \text{sonst.} \end{cases}$$

Funktionsdeklarationen

Es sei

fun $h\ x = t$

eine Deklaration einer Funktion vom Typ $typ_1 \to typ_2$. Die Semantik $[\![h]\!]$ von h wird wie folgt definiert.

- Sei $\sqsubseteq_{fl}$ die flache Ordnung auf typ_2^ω. Die Menge $typ_1 \to typ_2^\omega$ aller (totalen) Funktionen mit der ***Approximationsordnung***

  $f \sqsubseteq g \Leftrightarrow f(a) \sqsubseteq_{fl} g(a)$ für alle $a \in typ_1$

  bildet eine cpo mit dem kleinsten Element

  $\Omega : typ_1 \to typ_2^\omega$,
  $\Omega(a) = \omega$ für alle $a \in typ_1$.

- Das Fixpunkt-Funktional

  $F : (typ_1 \to typ_2^\omega) \to (typ_1 \to typ_2^\omega)$,
  $F(f)(a) = [\![t]\!]$, wobei $[\![t]\!]$ bestimmt wird bezüglich der Umgebung, die sich aus der aktuellen Umgebung durch Hinzunahme der Bindungen <h,f> und <x,a> ergibt.

  ist stetig. Dann ist

  $[\![h]\!] = fix(F)$,

  d.h.: $[\![h]\!] = sup\,\{F^i(\Omega) \mid i \in \mathbb{N}_0\}$.

Bemerkung

Die beschriebene Vorgehensweise kann ebenso für Funktionen höherer Ordnung angewendet werden. Für Funktionstypen wird dann nicht die flache Ordnung, sondern die Approximationsordnung zugrunde gelegt.

Beispiele (vgl. Abschnitt 7.1)

1. `fun h(n) = n+3`

   Mit $F(f)(n) = n + 3$ erhält man:

   $$\begin{aligned} &F^0(\Omega)(n) = \Omega(n) = \omega, \\ &F^1(\Omega)(n) = F(\Omega)(n) = n + 3, \\ &F^2(\Omega)(n) = F(F(\Omega))(n) = n + 3, \\ &\vdots \\ &F^i(\Omega)(n) = n + 3 \quad \text{für alle } i \geq 1. \end{aligned}$$

   Damit: $[\![h]\!](n) = n + 3$.

2. `fun g(n) = if n=0 then 1 else g(n+1)` (für $n \geq 0$)

   Mit $F(f)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ f(n+1) & \text{sonst} \end{cases}$
   erhält man:

   $$\begin{aligned} &F^0(\Omega)(n) = \Omega(n) = \omega, \\ &F^1(\Omega)(n) = F(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \Omega(n+1) = \omega & \text{sonst,} \end{cases} \\ &F^2(\Omega)(n) = F(F(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ F(\Omega)(n+1) = \omega & \text{sonst,} \end{cases} \\ &\vdots \\ &F^i(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \omega & \text{sonst} \end{cases} \quad \text{für alle } i \geq 1. \end{aligned}$$

   Damit: $[\![g]\!](n) = \begin{cases} 1, & \text{falls } n = 0 \\ \omega & \text{sonst.} \end{cases}$

3. `fun fak(n) = if n=0 then 1 else n*fak(n-1)` (für $n \geq 0$)

   Mit $F(f)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * f(n-1) & \text{sonst} \end{cases}$
   erhält man:

   $$\begin{aligned} &F^0(\Omega)(n) = \Omega(n) = \omega, \\ &F^1(\Omega)(n) = F(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * \Omega(n-1) = \omega & \text{sonst,} \end{cases} \end{aligned}$$

$$F^2(\Omega)(n) = F(F(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ \omega & \text{sonst,} \end{cases}$$

$$F^3(\Omega)(n) = F(F^2(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F^2(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ 2, & \text{falls } n = 2 \\ \omega & \text{sonst,} \end{cases}$$

$$F^4(\Omega)(n) = F(F^3(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F^3(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ 2, & \text{falls } n = 2 \\ 6, & \text{falls } n = 3 \\ \omega & \text{sonst} \end{cases}$$

usw.

Die $F^i(\Omega)$ ***approximieren*** die Fakultätsfunktion $n \mapsto n!$ „immer genauer“ (für immer mehr Argumente $0, 1, 2, 3, \ldots$):

$$F^i(\Omega)(n) = \begin{cases} n! & \text{für } n < i \\ \omega & \text{sonst.} \end{cases}$$

Für das Supremum (den „Limes“) gilt:

$$[\![fak]\!](n) = n!$$

Kapitel 8
Imperative Programmierung

8.1 Zustände, Anweisungen, Variablen

von-Neumann-Maschinen

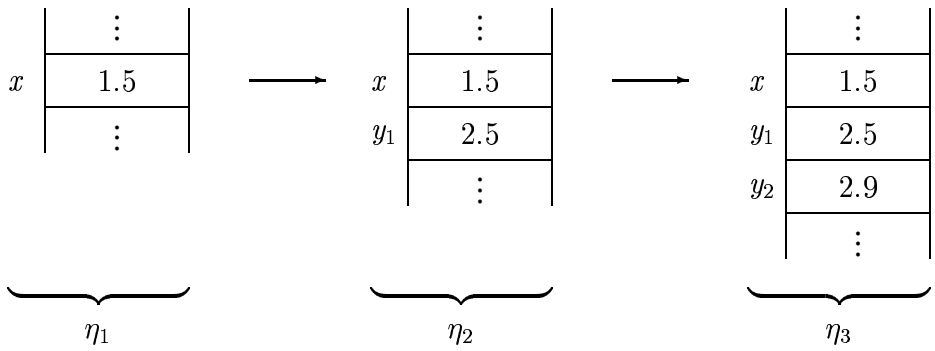
- Heute gebräuchliche Rechner folgen der von-Neumann-Rechner-Architektur. Sie besitzen einen ***Speicher*** zur Aufnahme von (insbesondere) Daten. Die Inhalte des (aus einzelnen, linear angeordneten ***Speicherzellen*** bestehenden) Speichers bestimmen seinen ***Zustand***.
- Die Verarbeitungsschritte des Rechners bestehen darin, die Inhalte gewisser Speicherzellen, d.h. den Zustand zu verändern. Die Abarbeitung eines Algorithmus auf dem Rechner wird durch eine Folge solcher Schritte realisiert.

Beispiel

Die Funktion (beachte Änderung der Pseudocode-Schreibweise gegenüber Kapitel 2)

function $f\,(x$: **real**$)$ **real** :
  let $y_1 = x + 1.0$
    $y_2 = y_1 + 1.0/y_1$
  in $y_2 * y_2$
  end

berechnet $(x + 1 + \frac{1}{x+1})^2$ für beliebiges $x \neq -1$. Die Abarbeitung dieses Algorithmus für einen aktuellen Parameter x_0 (d.h. die Auswertung von $f\,(x_0)$) beginnt in einem Zustand, in dem in einer dem formalen Parameter x zugeordneten Speicherzelle der Wert x_0 steht. Skizze des Ablaufs der Verarbeitungsschritte (für $x_0 = 1.5$):



η_1, η_2, η_3: Zustände. Die beiden Zustandsveränderungen realisieren die Elementdeklarationen für y_1 und y_2.

Imperative (prozedurale) Programmierung

- Grundidee: Formulierung von Algorithmen mit ***Anweisungen***, die (Folgen von) Zustandsänderungen explizit beschreiben.
- Grundlegende Anweisungen: ***Zuweisungen*** der allgemeinen Form

  $$a := t.$$

  a: (***Programm-, Zustands-***) ***Variable***
  (informell zu verstehen als „Name einer Speicherzelle“ mit dem „Inhalt der Zelle“ als ihrem Wert);
  t: Term (wie bisher, darf aber nun auch Variablen enthalten).

  $a := t$ bewirkt eine Zustandsänderung: Der Wert von t im „alten“ Zustand wird Wert von a im „neuen“ Zustand. (Bei Auswertung von t in einem Zustand werden in t enthaltene Variablen mit den Werten berücksichtigt, die sie in diesem Zustand haben.)
- Beschreibung der Zustandsänderungen im Beispiel von oben:

  $$y_1 := x + 1.0;$$
  $$y_2 := y_1 + 1.0/y_1$$

  („;“: siehe unten.)

  Statt der Variablen y_2 kann hier auch y_1 zum Ablegen des Wertes von $y_1 + 1/y_1$ wiederverwendet werden (da der „alte“ Wert von y_1 danach nicht mehr gebraucht wird).

  f als imperativer Algorithmus (mit dieser Vereinfachung und y für y_1):

  function $f'(x : \textbf{real})$ **real** :
  var y : **real**; $(*1*)$
  $y := x + 1.0;$
  $y := y + 1.0/y;$
  $y * y$ $(*2*)$

  $(*1*)$: ***Variablendeklaration*** (für die (***lokale***) Variable y; mit Typangabe).
  $(*2*)$: „Ergebnisterm“ (wie bisher, aber ohne **let** . . . **in** . . . **end**).

Anweisungen

- Zuweisungen: Einfache Anweisungen; beschreiben Einzelschritte im Verlauf von aufeinanderfolgenden Zustandsänderungen.
- Darüber hinaus: Zusammengesetzte Anweisungen zur Beschreibung der (Steuerung der) Aufeinanderfolge von Zustandsänderungen. „Grundausstattung“:
  - ***Sequentielle Komposition***,
  - ***Bedingte Anweisungen***,
  - ***Wiederholungsanweisungen*** (***Schleifen***).

Sequentielle Komposition

Allgemeine Gestalt:

$S_1; S_2$

Dabei: S_1, S_2 Anweisungen.

Informelle Bedeutung: Nacheinander-Ausführen von zuerst S_1 und anschließend S_2.

Beispiel: $y := x + 1.0; y := y + 1.0/y$ in obiger Funktion f'.

(In Analogie zu diesem Konstrukt trennen wir in unserem Pseudocode auch Variablendeklarationen und Ergebnisterme durch Strichpunkte ab.)

Bedingte Anweisungen

Allgemeine Gestalten:

if b **then** S_1 **else** S_2 **end** (1)
if b **then** S_1 **end** (2)

Dabei: b Bedingung (wie früher), S_1, S_2 Anweisungen (***Zweige*** der bedingten Anweisung).

Informelle Bedeutung: Fallunterscheidung gemäß Wert von b; falls dieser $true$ ist, Ausführung von S_1, andernfalls Ausführung von S_2 bei (1) bzw. keine Zustandsänderung bei (2).

Beispiele:

1. Berechnung von $f(x) = \begin{cases} x^2 + 1, & \text{falls } x < 1 \\ x^2 - 1 & \text{sonst} \end{cases}$

```
function f(x : real) real :
  var y : real;
  y := x * x;
  if x < 1.0 then y := y + 1.0
             else y := y - 1.0
  end;
  y
```

2. (vgl. Abschnitt 2.2)

```
function Fahrtzeit''(s_ab : nat, m_ab : nat, s_an : nat, m_an : nat) nat :
  var z : nat;
  z := (s_an - s_ab) * 60 + m_an - m_ab;
  if z < 0 then z := z + 1440 end;
  z
```

Wiederholungsanweisungen (Schleifen)

Allgemeine Gestalt:

while b **do** S **end**

Dabei: b Bedingung, S Anweisung (***Rumpf*** der Wiederholungsanweisung).

Informelle Bedeutung: S wird nacheinander wiederholt ausgeführt. Vor jeder Ausführung von S wird der Wert von b bestimmt. Die Wiederholungen (und damit die Gesamtanweisung) enden, wenn dieser Wert erstmals $false$ ist. (Ist dies nie der Fall, „endet“ die Anweisung nicht; der Algorithmus, der die Anweisung enthält, terminiert nicht.)

Das Konzept der Wiederholungsanweisung nennt man auch ***Iteration***. Algorithmen mit solchen Anweisungen heißen ***iterativ***.

Beispiele:

1. Fakultät

```
function fakit(n : nat) nat :
  var erg : nat;        (* Zur sukzessiven Ergebnisberechnung *)
  var j : nat;          (* Zur „Zählung“ der zu multiplizierenden Zahlen *)
  erg := 1;
  j := 1;
  while j ≤ n do
    erg := j * erg;
    j := j + 1
  end;
  erg
```

2. Größter gemeinsamer Teiler ($m, n \geq 1$)

```
function ggtit(m : nat, n : nat) nat :
  var x : nat;
  var y : nat;
  x := m;
  y := n;
  while x ≠ y do
    if x > y then x := x − y else y := y − x end
  end;
  x
```

3. Suchen eines Datenelements in einer Reihung natürlicher Zahlen
   (vgl. *enthalten1* in Abschnitt 4.5)

```
function enthalten1it(x : nat vect, a : nat) bool :
  var gefunden : bool;
  var j : nat;
  gefunden := false;
  j := 1;
  while j ≤ dim(x) ∧ ¬gefunden do
    if a = get(x, j) then gefunden := true end
  end;
  gefunden
```

(Beachte: Wegen der direkten Zugriffsmöglichkeit auf einzelne Reihungskomponenten eignen sich Reihungen besonders gut für die Bearbeitung mit iterativen Algorithmen; siehe auch Abschnitt 8.3.)

Andere Schleifenformen

Außer der oben eingeführten (***while***-) Schleifenform: Noch weitere Formen gebräuchlich, z.B.:

for $i = t_1$ **to** t_2 **do** S **end**

(***Laufanweisung, gezählte Wiederholungsanweisung, for-Schleife***) steht für:

var i : **nat**;
$\vdots$
$i := t_1$;
while $i \leq t_2$ **do** $S; i := i + 1$ **end**

(der ***Zähler*** i kommt sonst nirgends und auch in t_1 und t_2 nicht vor und wird in S nicht verändert).

Informelle Bedeutung: Sind m und n die Werte von t_1 bzw. t_2, so wird S $(n - m + 1)$-mal hintereinander ausgeführt (falls $m \leq n$), wobei i die Werte $m, m + 1, \ldots, n$ durchläuft. Gilt $m > n$, so wird S gar nicht ausgeführt.

Beispiel: Fakultät (siehe Beispiel oben, j als Zähler)

function $fakit'(n$: **nat**$)$ **nat** :
  var erg : **nat**;
  $erg := 1$;
  for $j = 1$ **to** n **do** $erg := j * erg$ **end**;
  erg

Bemerkungen

- Das Konzept der Iteration spielt eine ähnliche Rolle wie die Rekursion in der funktionalen Programmierung (vgl. obige Beispiele und Abschnitt 8.4). Insbesondere: Terminierungsproblematik bei Wiederholungsanweisungen.
- Die oben nur informell beschriebene Bedeutung der imperativen Konzepte ist auch zu einer präzisen (operationellen) Semantik-Definition formalisierbar. Vorgehensweise (nur angedeutet):
  - Formale Definition eines Zustands als Abbildung

    $$\eta : VAR \to WERTE$$

    von der Menge VAR der Variablen des Algorithmus in die Menge $WERTE$ der möglichen Werte (einschließlich dem „undefinierten“ Wert ω).
  - Definition einer Auswertungsfunktion W_η für Terme (Wert eines Terms im Zustand η; analog zu W in Abschnitt 3.4; für Variablen a ist $W_\eta(a) = \eta(a)$).

– Semantik einer Anweisung S: ***Zustandsübergangsfunktion***

$$\Phi_S : ZUSTÄNDE \to ZUSTÄNDE$$

($ZUSTÄNDE$ = Menge aller Zustände). Beispiel:

$$\Phi_{a:=t}(\eta) = \eta',$$

wobei $\eta'(a) = W_\eta(t)$ und $\eta'(b) = \eta(b)$ für alle von a verschiedenen Variablen $b \in VAR$.

8.2 Prozeduren

Prozedurabstraktion

- Eine Klasse gleichartiger Anweisungen bestimmt ein imperatives „Verarbeitungsmuster“. Dieses kann mit Hilfe von ***Variablen*** (***Referenz***) ***-parametern*** (d.h. Parametern, die für Variablen stehen) ausgedrückt werden.

  Beispiel: „Erhöhe Variablenwert um 1“ (Variable vom Typ **nat**).

  Einzelanweisungen: $a := a + 1$,
  $zaehler := zaehler + 1$,
  $b_1 := b_1 + 1$
  usw.

  Verarbeitungsmuster: $x := x + 1$ (x Variablenparameter).

- Analog zur Bildung von Funktionen aus Termen (Funktionsabstraktion) kann aus einer derart parametrisierten Anweisung S (mit den Variablenparametern $x_1, x_2, \ldots, x_n$) eine ***Prozedur*** gebildet werden (***Prozedurabstraktion***), die in Form einer ***Prozedurdeklaration*** angegeben wird:

  procedure $\langle$*Prozedurname*$\rangle$(**var** $x_1 : typ_1, \ldots,$ **var** $x_n : typ_n$) : S **end**

  (S heißt ***Rumpf*** der Prozedur.) Im Beispiel:

  procedure $incrvar$(**var** x : **nat**) :
  $x := x + 1$
  end

- Einzelne Verarbeitungen gemäß dem so definierten Verarbeitungsmuster ergeben sich durch ***Prozeduraufrufe*** mit Variablen als Argumenten (aktuellen Parametern) für die (formalen) Variablenparameter. Prozeduraufrufe können in anderen Algorithmen als weitere Form von Anweisungen verwendet werden, z.B.:

  $incrvar(a)$,
  $incrvar(zaehler)$,
  $b_1 := c$; **if** $b_1 = 0$ **then** $incrvar(b_1)$ **end**.

- Außer Variablenparameter können Prozeduren auch Parameter für Werte (im bisherigen Sinne; im jetzigen Kontext ***Wertparameter*** genannt) enthalten. Beispiel:

  procedure $incrvarwert$(**var** x : **nat**, y : **nat**) :
  $x := x + y$
  end

- Die operationelle Semantik eines Prozeduraufrufs im Sinne der Ausführungen im vorigen Abschnitt ist die durch den Prozedurrumpf beschriebene Zustandübergangsfunktion, wobei die Namen der einander entsprechenden formalen und aktuellen Variablenparameter miteinander identifiziert werden. Diese Art der „Parameterübergabe“ heißt ***Referenzaufruf*** (***call-by-reference***). Wertparameter werden wie bei Funktionen durch Wertaufruf „übergeben“ (vgl. Abschnitt 3.4).

Lokale Variablen in Prozeduren

Wie bei Funktionen kann auch der Rumpf einer Prozedur vorangestellte (lokale) Variablendeklarationen enthalten.

Beispiel: Prozedur zum Vertauschen der Werte zweier **nat**-Variablen:

procedure $vertauschen$(**var** x : **nat**, **var** y : **nat**) :
  var h : **nat**; $(*$ zur „Zwischenspeicherung“ $*)$
  $h := x$;
  $x := y$;
  $y := h$
end

Unterordnung von Algorithmen

- Prozeduren und Funktionen (in diesem imperativen Kontext oft auch ***Funktionsprozeduren*** genannt) können „nebeneinander“ verwendet werden. Darüber hinaus können sie (analog wie in Abschnitt 5.3 beschrieben) gegenseitig untergeordnet werden. (Pseudocode-Schreibweise jetzt: analog zu lokalen Variablendeklarationen.) Hinsichtlich Variablenparametern bedeutet dies, dass in Prozeduraufrufen als aktuelle Parameter auch formale Variablenparameter übergeordneter Prozeduren auftreten dürfen.

  Beispiele:

  1. **procedure** $addf$(**var** x : **nat**, y : **nat**) :
     function f(z : **nat**) **nat** : $z * z$;
     $x := x + f(y)$
     end
  2. **function** g(y : **nat**) **nat** :
     procedure $incrvar$(**var** x : **nat**) : $x := x + 1$ **end**;
     var a : **nat**;
     $a := y * y$;
     $incrvar(a)$;
     a

3.
procedure $malincr$(**var** x : **nat**) :
  procedure $malz$(**var** y : **nat**, z : **nat**) : $y := z * y$ **end**;
  $malz(x, 3)$;
  $x := x + 1$
end

- Bei der Unterordnung können (analog wie bei Funktionen, vgl. Abschnitt 5.3) Parameter gegebenenfalls unterdrückt werden und statt dessen ***globale*** Größen verwendet werden. (Beachte: Dadurch können auch ***parameterlose*** Prozeduren entstehen.)

  Beispiele:

1.
procedure $malincr'$(**var** x : **nat**) :
  procedure $malz'(z$: **nat**) : $x := z * x$ **end**;
  $malz'(3)$;
  $x := x + 1$
end

2.
procedure $malincr''$(**var** x : **nat**) :
  procedure $mal3()$: $x := 3 * x$ **end**;
  $mal3()$;
  $x := x + 1$
end

- Die Unterordnung von Funktionen und Prozeduren (die wieder ***Blöcke*** genannt werden) erzeugt eine ***Blockstruktur*** wie in Abschnitt 5.3 beschrieben. Die Begriffe ***Bindungsbereich*** und ***Gültigkeitsbereich*** von Namen überträgt sich im jetzigen Kontext auch auf Bezeichnungen von Variablen und Variablenparametern.

Gebrauch von Variablenparametern

- Eigenschaft aller Variablenparameter in den vorstehenden Beispielen: Parameter kommt in „linken und rechten Seiten“ von Zuweisungen vor (***Transientparameter***).

  Bei Aufruf mit einer Variablen a: Wert von a im Aufrufzustand wird in der Prozedur verarbeitet; nach Beendigung des Aufrufs hat a einen neuen Wert.

- Weitere mögliche Verwendung von Variablenparametern:
  - als ***Eingabeparameter***: Wert im Aufrufzustand wird verwendet, aber nicht verändert. (Dafür kann grundsätzlich auch ein Wertparameter benutzt werden.)
  - als ***Ausgabeparameter***: Wert im Aufrufzustand ist nicht relevant.

  Beispiel:

procedure p(**var** x : **nat**, **var** y : **nat**, **var** z : **nat**) :
  $z := x + z$;
  $y := x$
end

  x: Eingabeparameter, y: Ausgabeparameter, z: Transientparameter.

Modularisierung mit Prozeduren

- Die methodischen Ausführungen in Kapitel 5 zum Konzept der Modularisierung gelten auch im Bereich der imperativen Programmierung. In diesem Kontext kann ein Modul außer Datentypen und Funktionen auch Variablen und Prozeduren enthalten. Die Variablen definieren einen ***Zustandsraum***, Prozeduren ermöglichen Zustandsänderungen.

- Beispiel: Für ein Bankkonto sind u.a. folgende Angaben relevant: Kontonummer, Kontostand (usw.); es sollen „Operationen“ wie „Betrag buchen“ (usw.) ausgeführt werden können. Ein solches Konto lässt sich als Modul konzipieren:

  module $KONTO$
  var $kontonummer$: **nat**;
  var $kontostand$: **real**;
  $\vdots$
  procedure $buchen(betrag$: **real**$)$:
  $kontostand := kontostand + betrag$
  end;
  $\vdots$
  end

  Dabei sollte etwa $buchen$ zur Schnittstelle von $KONTO$ gehören, die enthaltenen Variablen jedoch nicht.

  (Bemerkung: Realistischerweise wird nicht nur ein Konto, sondern es werden viele derartige Konten zu verwalten sein. Dies führt direkt auf Grundkonzepte der Programmiersprache Java, siehe Informatik II.)

8.3 Datenstrukturen in der imperativen Programmierung

Variablen von zusammengesetzten Typen

- Außer Variablen für Werte elementarer Typen können auch Variablen für komplexe Datenstrukturen verwendet werden.

  Dabei üblich:

  - Statt Konstruktor-Funktionen sind bestimmte Anwendungen dieser Funktionen auf Variablen als ***Basisprozeduren*** verfügbar.
  - Sonstige Basisfunktionen sind in bisheriger Weise verfügbar.

- Typische in imperativen Programmiersprachen (in dieser modifizierten Form) direkt verfügbare Datenstrukturen: Tupel und Reihungen.

Verbunde (vgl. Abschnitt 4.2)

- Die imperativen Fassungen von Tupeln (meist verwendet in der Form von Records) werden auch ***Verbunde*** genannt. Typbezeichnung z.B.:

  record $sel_1 : typ_1, \ldots, sel_n : typ_n$ **end**

  (zur Unterscheidung statt $\{sel_1 : typ_1, \ldots, sel_n : typ_n\}$ in SML).

  Basisfunktionen: $sel_1, \ldots, sel_n$ (Selektoren $\#sel_i$ in SML-Schreibweise).

  Basisprozeduren (für $i = 1, \ldots, n$):

  procedure $selupd_i$(**var** x : **record** $sel_1 : typ_1, \ldots, sel_n : typ_n$ **end**, $y : typ_i$) :
  $x := (sel_1(x), \ldots, sel_{i-1}(x), y, sel_{i+1}(x), \ldots, sel_n(x))$
  end

  (***Selektive Änderung***: Veränderung der i-ten Komponente des Tupels.)

- Typische Schreibweise für einen Aufruf $selupd_i(a, t)$ mit einer Variablen oder einem Variablenparameter der Art **var** a : **record** . . . **end** und einem Term t vom Typ typ_i:

  $sel_i(a) := t.$

- Beispiel:

  type $Person =$ **record** $Name$: **string**, $Alter$: **nat end**;
  procedure $Geburtstag$(**var** $p : Person, name$: **string**) :
  if $Name(p) = name$ **then** $Alter(p) := Alter(p) + 1$ **end**
  end

- Beachte: Für a wie oben bezeichnet $sel_i(a)$ einerseits die i-te Komponente des Wertes von a, andererseits wird es in

  $sel_i(a) := t.$

  wie eine Variable verwendet. In Fortführung dieser Schreibweise kann $sel_i(a)$ auch als aktueller Variablenparameter (des Typs typ_i) verwendet werden.

  Beispiel (mit $Person$ wie oben und der Prozedur $incrvar$ aus Abschnitt 8.2):

  procedure $Geburtstag'$(**var** $p : Person, name$: **string**) :
  if $Name(p) = name$ **then** $incvar(Alter(p))$ **end**
  end

Felder (vgl. Abschnitt 4.5)

- Die imperativen Fassungen von Reihungen werden auch ***Felder*** (***Arrays***) genannt. Typbezeichnung z.B.:

  typ **array**$[n]$

(statt typ **vect**). Dabei ist $n \in \mathbb{N}_0$ die jeweils mit anzugebende Länge der Felder dieses Typs.

Basisfunktion: get (wie bisher; dim nicht benötigt).

Basisprozedur:

procedure $arrupd$(**var** $x : typ$ **array**$[n], i :$ **nat**, $y : typ$) :
  pre $1 \leq i \leq n$;
  $x := update(x, i, y)$
end

(Wie oben: selektive Änderung.)

- Typische Schreibweisen:

  $a[i]$ für $get(a, i)$,
  $a[i] := t$ für $arrupd(a, i, t)$.

- Beispiele:

  1. Initialisierung eines Feldes

     procedure $initial$(**var** $x : typ$ **array**$[n], b : typ$) :
       for $i = 1$ **to** n **do** $x[i] := b$ **end**
     end

  2. Sortieren eines Feldes ganzer Zahlen durch Einfügen (vgl. Abschnitte 4.4 und 5.3)

     procedure $inssortarray$(**var** $x :$ **int array**$[n]$) :
       procedure $insertelarray$($k :$ **nat**) :
         var $i :$ **nat**;
         var $j :$ **nat**;
         $i := k$;
         $j := k - 1$;
         while $i > 1 \wedge x[i] > x[j]$ **do**
           $vertauschen(x[i], x[j])$;
           $i := i - 1$;
           $j := j - 1$
         end
       end;
       for $i = 2$ **to** n **do** $insertelarray(i)$ **end**
     end

     (Verwendet: $vertauschen$ aus Abschnitt 8.2.)

- Bemerkung:

  In Typbezeichnung typ **array**$[n]$: Feldindizes $1, \ldots, n$.
  Meist auch allgemeinere Indizierung möglich, z.B.:

  typ **array**$[i : j]$ (Indizes $i, \ldots, j$).

Mehrstufige Felder

- Die in Verbunden und Feldern vorkommenden Typen können selbst wieder Verbunde oder Felder sein. Insbesondere: Felder der Art

  typ **array**$[n_l]$ ... **array**$[n_2]$ **array**$[n_1]$,

  kürzer geschrieben als

  typ **array**$[n_1, n_2, \ldots, n_l]$,

  heißen ***mehrstufige Felder***.

  Schreibweise: $a[i_1, i_2, \ldots, i_l]$ statt $a[i_1][i_2]\ldots[i_l]$

- Beispiel: Der Typ

  type $matrix =$ **real array**$[n, n]$

  umfasst reelle $n \times n$-Matrizen als Datenelemente. Die Prozedur

  ```
  procedure transponieren(var x : matrix) :
    for i = 1 to n do
      for j = i + 1 to n do
        vertauschen(a[i, j], a[j, i])
      end
    end
  end
  ```

  beschreibt das „Transponieren“ einer solchen Matrix (unter Verwendung der Prozedur $vertauschen$ aus Abschnitt 8.2).

  Beachte (analog zur entsprechenden Bemerkung bei Verbunden): $a[i, j]$ und $a[j, i]$ werden als aktuelle Variablenparameter verwendet.

Variablen für variante Daten (vgl. Abschnitt 4.3)

Neben Verbunden und Feldern sind in imperativen Programmiersprachen meist auch Variablen für variante Daten (oder zumindest Spezialfälle davon) verfügbar. Beispiel (mit Aufzählungstyp):

```
type Farbe = rot | blau | gelb | gruen;
var f : Farbe;
if f = blau then f := rot end
```

Dynamische Datenstrukturen (vgl. Abschnitte 4.4 und 4.6)

- Listen- und baumartige Datenstrukturen sind ***dynamische*** Strukturen (mit Basisfunktionen zur Veränderung der „Gestalt“ eines Datenelements). Sie sind in imperativen Programmiersprachen üblicherweise nicht direkt verfügbar, können aber in moderneren Sprachen im Rahmen eines Modul-Konzepts eingeführt werden (im Sinne von Abschnitt 5.4).

- Dabei kann man grundsätzlich so vorgehen wie für Verbunde und Felder beschrieben, z.B.:

  procedure $append$(**var** x : typ **list**, y : typ) :
  $x := y :: x$
  end

  könnte eine Basisprozedur für eine imperative Behandlung von Listen sein.

- Bei derartigen Modularisierungen fasst man oft auch eine Variable des betreffenden Typs mit den Basisfunktionen und -prozeduren zusammen, in denen dann die Variable als globale Größe verwendet wird (vgl. Abschnitt 8.2).

  Beispiel (unter Verwendung eines Datentyps **nat stack**; vgl. Abschnitt 5.4):

  module $KELLER$
  type nat stack;
  var s : **nat stack**;
  $isemptys = isemptyst(s)$;
  $tops = top(s)$;
  procedure $creates() : s := emptyst$ **end**;
  procedure $pushs(x$: **nat**$) : s := push(x, s)$ **end**;
  procedure $pops() : s := pop(s)$ **end**
  end

Implementierung dynamischer Datenstrukturen

- Zur Realisierung listen- und baumartiger Datenstrukturen enthalten imperative Programmiersprachen meist noch einen speziellen Typ von Datenelementen, die ***Zeiger*** (***Verweise***, ***Pointer***) genannt werden.

  Intuition: Ein Zeiger identifiziert eindeutig eine Speicherzelle.

- Alle Komponenten einer dynamischen Datenstruktur werden jeweils mit einem oder mehreren Zeigern zu einem Tupel zusammengefasst, die Zeiger realisieren den „strukturellen Zusammenhang“ (das ***Geflecht***) der Komponenten.

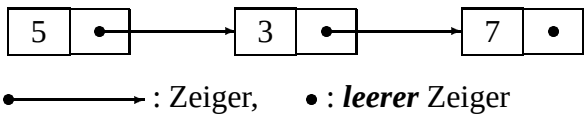
- Als Bild:

  Liste: $(5, 3, 7)$

  Speicherzellen für die Komponenten:

  Geflecht-Realisierung der Liste:

8.4 Rekursion und Iteration

Beispiel: Größter gemeinsamer Teiler

- Repetitiv rekursiver Algorithmus (vgl. Abschnitt 3.4):

  function $ggt(m : \textbf{nat}, n : \textbf{nat})$ **nat** :
  if $m = n$ **then** m
  else if $m > n$ **then** $ggt(m - n, n)$
  else $ggt(m, n - m)$

  Beispiel-Auswertung:

  $$\begin{aligned} ggt(14, 6) &\rightsquigarrow ggt(8, 6) \\ &\rightsquigarrow ggt(2, 6) \\ &\rightsquigarrow ggt(2, 4) \\ &\rightsquigarrow ggt(2, 2) \\ &\rightsquigarrow 2 \end{aligned}$$

- Iterativer Algorithmus (vgl. Abschnitt 8.1):

  function $ggtit(m : \textbf{nat}, n : \textbf{nat})$ **nat** :
  var x : **nat**;
  var y : **nat**;
  $x := m$;
  $y := n$;
  while $x \neq y$ **do**
  if $x > y$ **then** $x := x - y$ **else** $y := y - x$ **end**
  end;
  x

  Beispiel-Ablauf (nur für x und y, nach jedem „Iterationsschritt“):

x	14	8	2	2	2	← Endergebnis
y	6	6	6	4	2	

- Es gilt: Die Variablen x und y in $ggtit$ durchlaufen in der iterativen Lösung genau die Werte der aktuellen Parameter bei der Abfolge der rekursiven Aufrufe in ggt.

Allgemeiner Zusammenhang

Iterative Berechnungen spiegeln den Auswertungsablauf von repetitiven Rekursionen wider. Ist f eine repetitiv rekursive Funktion mit den Parametern $x_1, \ldots, x_n$, so lässt sich f durch einen iterativen Algorithmus mit Variablen $xvar_1, \ldots, xvar_n$ nach folgendem Schema berechnen:

$xvar_1 := x_1$;
$\vdots$
$xvar_n := x_n$;
while Terminierungsbedingungen der Rekursion
(ausgedrückt für $xvar_1, \ldots, xvar_n$) sind nicht erfüllt
do Besetze $xvar_1, \ldots, xvar_n$ mit den neuen Parameterwerten gemäß Rekursion
end;
Ergebnisterm (ausgedrückt für $xvar_1, \ldots, xvar_n$) gemäß Basisfällen der Rekursion

Weitere Beispiele

1. Repetitiv rekursive Funktion $potenz$ aus Abschnitt 6.2:

   function $potenz(m$: **nat**, n : **nat**$)$ **bool** :
   if $n = 1 \vee m = n$ **then** $true$
   else if $n \; mod \; m \neq 0$ **then** $false$
   else $potenz(m, n \; div \; m)$

   Iterative Fassung gemäß obigem Schema:

   function $potenzit(m$: **nat**, n : **nat**$)$ **bool** :
   var $mvar$: **nat**;
   var $nvar$: **nat**;
   $mvar := m$;
   $nvar := n$;
   while $nvar \neq 1 \wedge mvar \neq nvar \wedge nvar \; mod \; mvar = 0$ **do**
   $nvar := nvar \; div \; mvar$
   end;
   $nvar = 1 \vee mvar = nvar$

   Vereinfachung ($mvar$ unnötig):

   function $potenzit'(m$: **nat**, n : **nat**$)$ **bool** :
   var $nvar$: **nat**;
   $nvar := n$;
   while $nvar \neq 1 \wedge m \neq nvar \wedge nvar \; mod \; m = 0$ **do**
   $nvar := nvar \; div \; m$
   end;
   $nvar = 1 \vee m = nvar$

2. Fakultät: Repetitiv rekursive Lösung durch Einbettung (vgl. Abschnitt 6.2) mit:

   function $fakallg(n$: **nat**, k : **nat**$)$ **nat** :
   if $n = 0$ **then** k **else** $fakallg(n - 1, k * n)$

   Iterative Fassung gemäß obigem Schema:

```
function fakallgit(n : nat, k : nat) nat :
  var nvar : nat;
  var kvar : nat;
  nvar := n;
  kvar := k;
  while nvar ≠ 0 do
    kvar := kvar * nvar;
    nvar := nvar − 1
  end;
  kvar
```

Es gilt $fak(n) = fakallg(n, 1)$. Dies kann in $fakallgit$ direkt berücksichtigt werden: k wird durch 1 ersetzt (und der Parameter k gestrichen). Damit: Iterativer Algorithmus zur Fakultätsberechnung (mit j statt $nvar$ und erg statt $kvar$):

```
function fakit''(n : nat) nat :
  var j : nat;
  var erg : nat;
  j := n;
  erg := 1;
  while j ≠ 0 do
    erg := erg * j;
    j := j − 1
  end;
  erg
```

($fakit''$ unterscheidet sich von $fakit$ aus Abschnitt 8.1 dadurch, dass die einzelnen Multiplikationen in umgekehrter Reihenfolge durchgeführt werden.)

Iteration mit Kellern

- Allgemein können beliebige Rekursionsformen unter Verwendung von ***Kellern*** (etwa in der Fassung des Moduls $KELLER$ aus Abschnitt 8.3) in Iterationen umgewandelt werden.

  Grundprinzip: Die aktuellen Parameter noch unerledigter rekursiver Aufrufe werden im Keller „gespeichert“. Gemäß dem Aufrufmechanismus werden sie in umgekehrter Reihenfolge benötigt. Dies entspricht genau den Stapel-Zugriffsmöglichkeiten des Kellers.

- Beispiel: Ackermann-Funktion (vgl. Abschnitt 2.3)

```
function ack(m : nat, n : nat) nat :
  if m = 0 then n + 1
  else if n = 0 then ack(m − 1, 1)
  else ack(m − 1, ack(m, n − 1))
```

Iterativer Algorithmus unter Verwendung des (leicht modifizierten) Moduls $KELLER$ aus Abschnitt 8.3 (die Schnittstelle von $KELLER$ enthalte

$$s\_enthaelt\_mehr\_als\_ein\_element = \neg isemptyst(s) \wedge \neg isempty(pop(s))$$

als zusätzliche Konstante):

function $ackit(m$: **nat**, n : **nat**$)$ **nat** :
  var $mpar$: **nat**;
  var $npar$: **nat**;
  $creates$;
  $pushs(m)$;
  $pushs(n)$;
  while $s\_enthaelt\_mehr\_als\_ein\_element$ **do**
    $npar := tops$;
    $pops$;
    $mpar := tops$;
    $pops$;
    if $mpar = 0$ **then** $pushs(npar + 1)$
    else if $npar = 0$ **then** $pushs(mpar - 1); pushs(1)$
        else $pushs(mpar - 1); pushs(mpar); pushs(npar - 1)$ **end**
    end
  end;
  $tops$

Rekursive Prozeduren

Das Konzept der Rekursion kann auch auf Prozeduren übertragen werden: ***Rekursive Prozeduren*** sind Prozeduren, die sich in ihrem Rumpf selbst aufrufen.

Beispiel: Fakultätsberechnung durch eine rekursive Prozedur

procedure $fakrekproc(n$: **nat**, **var** erg : **nat**$)$:
  $(*\ erg : Ausgabeparameter\ *)$
  if $n = 0$ **then** $erg := 1$
  else $fakrekproc(n - 1, erg)$;
      $erg := n * erg$
  end
end

Ein Aufruf $fakrekproc(k, x)$ mit $k \in \mathbb{N}_0$ und einer Variablen x vom Typ **nat** bewirkt, dass x den Wert $k!$ erhält.

8.5 Axiomatische Semantik und Hoare-Kalkül

Hoare-Formeln

Für imperative Programmierungskonzepte ist neben einer operationellen Semantik oft eine axiomatische Semantik (vgl. Diskussion in Abschnitt 7.1) nützlich. Eine typische derartige Semantik ist gegeben durch den ***Hoare-*** (***Zusicherungs-***) ***Kalkül***: Die Semantik einer Anweisung S wird beschrieben durch eine ***Hoare-Formel*** der Form

$$\{P\}\, S\, \{Q\}$$

Die ***Zusicherungen*** P (***Vorbedingung***, ***precondition***) und Q (***Nachbedingung***, ***postcondition***) beschreiben Eigenschaften von Zuständen (mit den in S enthaltenen Variablen).

Informelle Bedeutung:

> „Wenn P in einem Zustand vor Ausführung von S gilt und diese Ausführung terminiert, so gilt Q im Zustand nach der Ausführung."

Das Zuweisungsaxiom

Die Semantik einer Zuweisung

$$a := t$$

ist gegeben durch das ***Zuweisungsaxiom***

$$\{Q[t/a]\}\, S\, \{Q\}.$$

Dabei ist $Q[t/a]$ die Zusicherung, die aus Q entsteht, wenn man alle Vorkommen der Variablen a durch den Term t ersetzt.

Regeln für zusammengesetzte Anweisungen

- Für jede Art der Zusammensetzung von Anweisungen gibt es eine Regel, die eine Hoare-Formel für die entsprechend zusammengesetzte Anweisung in Abhängigkeit von der Gültigkeit von Hoare-Formeln für die Bestandteile der Anweisung bestimmt.

  Beispiele im Folgenden: Regeln für sequentielle Komposition, bedingte Anweisungen und (while-) Schleifen. (Schreibweise der Regeln: wie Typisierungsregeln, vgl. Abschnitt 3.5.)

- Sequentielle Komposition:

  $$\{P\}\, S_1\, \{R\}, \{R\}\, S_2\, \{Q\} \vdash \{P\}\, S_1; S_2\, \{Q\}$$

- Bedingte Anweisungen:

  $$\{b \wedge P\}\, S_1\, \{Q\}, \{\neg b \wedge P\}\, S_2\, \{Q\} \vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\, \{Q\},$$
  $$\{b \wedge P\}\, S\, \{Q\}, \neg b \wedge P \Rightarrow Q \vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ S\ \textbf{end}\, \{Q\}$$

Dabei bedeutet $R_1 \Rightarrow R_2$ für zwei Zusicherungen, dass R_2 aus R_1 folgt.

- while-Schleifen:

  $$\{b \wedge P\}\, S\, \{P\} \vdash \{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end}\ \{P \wedge \neg b\}$$

  Die Zusicherung P heißt ***Schleifeninvariante***: Sie wird durch einen einzelnen Schleifendurchlauf (d.h. Ausführung von S in einem Zustand, in dem außer P auch noch b gilt) nicht „verletzt“. Gilt sie zu Beginn der Schleife, so gilt sie dann (zusammen mit $\neg b$) auch nach deren Beendigung.

- Anwendungen dieser Regeln werden häufig in Form von ***annotierten*** Anweisungen geschrieben: Die Zusicherungen werden direkt an die „passenden“ Stellen in die zusammengesetzten Anweisungen („in den Programmtext“, wie Kommentare) eingefügt. Beispiel:

  $$\begin{array}{l} \quad\{z < 2 * (a+1)\} \\ a := a + 1; \\ \quad\{z < 2 * a\} \\ a := 2 * a \\ \quad\{z < a\} \end{array}$$

Die Folgeregeln

Um die vorgenannten Regeln sinnvoll anwenden zu können, benötigt man noch die allgemeinen (***Folge-***) Regeln

$$P \Rightarrow P_1, \{P_1\}\, S\, \{Q\} \vdash \{P\}\, S\, \{Q\},$$
$$\{P\}\, S\, \{Q_1\}, Q_1 \Rightarrow Q \vdash \{P\}\, S\, \{Q\}$$

Beispiel-Anwendung:

$$\begin{array}{l} \quad\{x = y\} \\ \textbf{if}\ x > 0\ \textbf{then} \\ \quad\{x > 0 \wedge x = y\} \\ \quad a := x \\ \quad\{a > 0 \wedge x = y\} \\ \textbf{else} \\ \quad\left.\begin{array}{l}\{x \le 0 \wedge x = y\} \\ \{1 - x > 0 \wedge x = y\}\end{array}\right\} \text{„}\Rightarrow\text{“} \\ \quad a := 1 - x \\ \quad\{a > 0 \wedge x = y\} \\ \textbf{end} \\ \quad\left.\begin{array}{l}\{a > 0 \wedge x = y\} \\ \{a > 0\}\end{array}\right\} \text{„}\Rightarrow\text{“} \end{array}$$

Verifikation imperativer Programme

Der Hoare-Kalkül eignet sich (zusätzlich zur Semantik-Definition) auch zur Verifikation von imperativen Programmen (vgl. Abschnitte 2.3 und 3.4).

Beispiel: Der Algorithmus

function $exp2(n : \textbf{nat})\ \textbf{nat}$:
  var a : **nat**;
  var i : **nat**;
  $a := 1;$
  $i := n;$
  while $i \neq 0$ **do**
    $a := 2 * a;$
    $i := i - 1$
  end;
  a

$\Big\}\ S$

berechnet 2^n. Seine Korrektheit wird ausgedrückt durch die Hoare-Formel

$$\{true\}\ S\ \{a = 2^n\}$$

(die Vorbedingung $true$ bedeutet „keine Einschränkung“) und kann wie folgt formal bewiesen werden:

$\{true\}$
$\{1 = 1 \wedge n = n\}$
$a := 1;$
$\{a = 1 \wedge n = n\}$
$i := n;$
$\{a = 1 \wedge i = n\}$
$\{a * 2^i = 2^n\}$ (Schleifeninvariante)
while $i \neq 0$ **do**
$\{i \neq 0 \wedge a * 2^i = 2^n\}$
$\{2 * a * 2^{i-1} = 2^n\}$
$a := 2 * a;$
$\{a * 2^{i-1} = 2^n\}$
$i := i - 1$
$\{a * 2^i = 2^n\}$
end
$\{a * 2^i = 2^n \wedge i = 0\}$
$\{a = 2^n\}$

Beachte: Ein solcher Beweis zeigt wegen der Bedeutung der Hoare-Formeln nur die partielle Korrektheit eines imperativen Programms. Der Nachweis der Terminierung (von Schleifen) erfordert andere Argumente.