LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Informatik I

Kurzskriptum zur Vorlesung

Wintersemester 2005/06

F. Kröger

INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für Programmierung und Softwaretechnik

Inhaltsverzeichnis

Einleitung

Informatik (computer science, informatics)

- [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.

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: Verbindung von Informatik mit anderen Wissenschaften.

Inhalt dieser Vorlesung:

- Einführung in einen zentralen Bereich von (2): Grundlegende Konzepte, Methoden und Techniken der systematischen Informationsverarbeitung,
- abgestützt auf Methoden aus (3).

Einige historische Daten

- 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.
- 1647: Gottfried Wilhelm Leibniz konstruiert eine Rechenmaschine mit Staffelwalzen für die vier Grundrechenarten.

- 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.
- 1946: J.P. Eckert und J.W. Mauchly stellen den ersten voll elektronischen Rechner ENIAC 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

Algorithmen

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 $n \in \mathbb{N}$

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

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

Mathematische Modellierung

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 „einhundertdreiundsechzig“.
- 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: vier natürliche Zahlen; ***Parameter*** hierfür: $s_{ab}, m_{ab}, s_{an}, m_{an}$;
Ergebnis: natürliche Zahl;
Berechnung:

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

Algorithmus(idee) beinhaltet:

- Daten 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).
- ***Effektivität*** der elementaren Verarbeitungsschritte: Jeder elementare Verarbeitungsschritt muss von der zugrunde liegenden „Verarbeitungseinheit“ (***Prozessor***) tatsächlich ausführbar sein.

Eigenschaften 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.Allg. für die Entwicklung notwendig.

1.3 Programmierung

Darstellung von Algorithmen

Zur Ausführung auf einem Computer muss ein Algorithmus (einschließlich der zu verarbeitenden Daten) vollkommen formal in einer ***Programmiersprache*** (als ***Programm***) dargestellt werden.

Beispiel: Darstellung des Fahrtzeit-Algorithmus

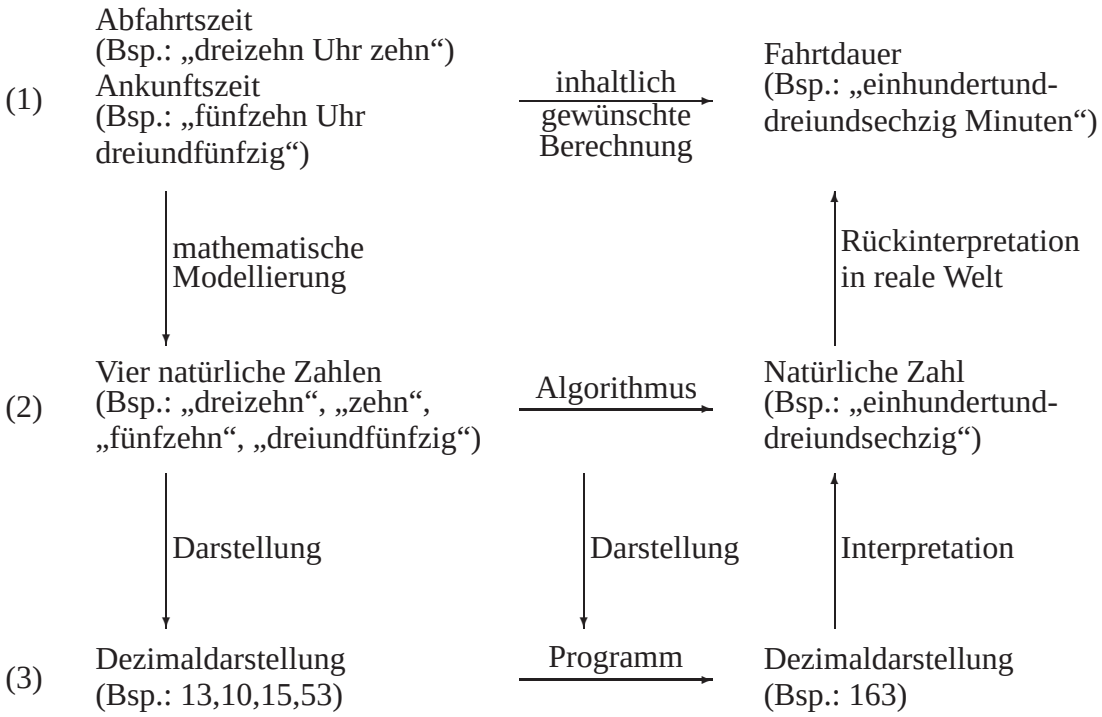
- in SML (in dieser Vorlesung verwendete Programmiersprache):

```
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):

```
int fahrtzeit(int s_ab,int m_ab,int s_an,int m_an)
{
 return((s_an - s_ab) * 60 + m_an - m_ab);
}
```

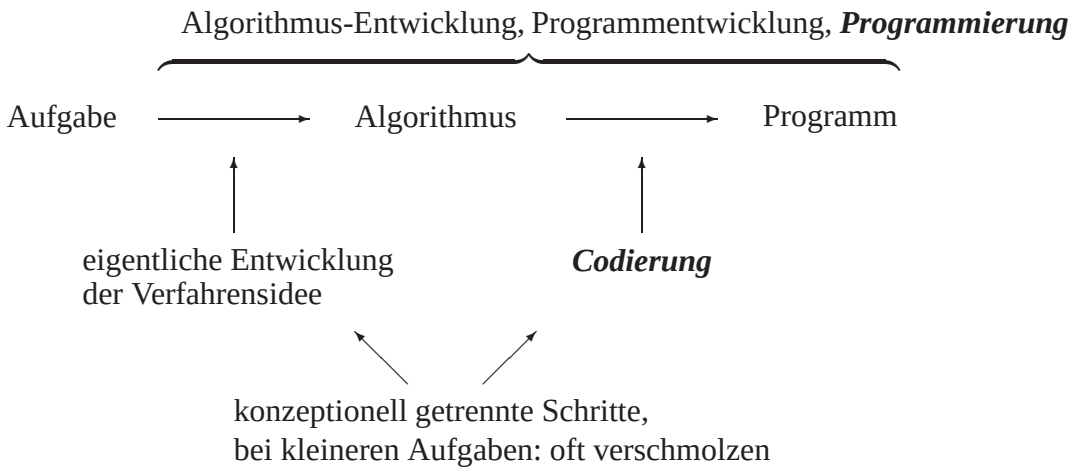
Gesamtbild (Fahrtzeit-Beispiel)



(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:



Pseudocode

- Zur Aufschreibung von Algorithmen (in seinen Entwicklungsstufen) ist es methodisch sinnvoll, nicht eine konkrete Programmiersprache, sondern eine programmiersprachenähnliche Darstellung unter Verwendung typischer algorithmischer Konzepte, mathematischer Schreibweisen und eventuell auch verbaler Zusätze (***Pseudocode***) zu verwenden.

- 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

Entwicklung der Programmiersprachen

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

Beispiel (Fahrtzeit-Algorithmus in ix386-Maschinensprache):

```
pushl   %ebp
movl    %esp, %ebp
movl    16(%ebp), %eax
subl    8(%ebp), %eax
leal    (%eax,%eax,2), %edx
leal    (%edx,%edx,4), %eax
sall    $2, %eax
addl    20(%ebp), %eax
subl    12(%ebp), %eax
popl    %ebp
```

- Die Bereitstellung von „problemorientierten“ („höheren“) Programmiersprachen (und ihre automatische Übersetzung in „Maschinencode“), in denen so formuliert werden kann wie in den Beispielen zu Beginn des Abschnitts, ist eine wichtige Errungenschaft der Informatik-Entwicklung.

Programmierstile und -sprachen

- Die algorithmischen Konzepte von verschiedenen problemorientierten Programmiersprachen sind zum Teil gleichartig, zum Teil charakterisieren sie unterschiedliche ***Programmierstile***.

- In dieser Vorlesung behandelter Programmierstil: ***Funktionale*** (***applikative***) Programmierung (Algorithmus als Formulierung des funktionalen Zusammenhangs zwischen Eingabe- und Resultatdaten; vgl. Fahrtzeit-Algorithmus).

  Beispiele für Programmiersprachen: Lisp, ML, Miranda, Haskell.

- Weitere wichtige Programmierstile: ***Imperative*** (***prozedurale***) Programmierung (Algorithmus „verändert Größen“ durch „Anweisungen“, z.B.: „Erhöhe x um 1“; Sprachen: Fortran, Algol, Pascal, Modula, C), ***objektorientierte*** Programmierung (Sprachen: Eiffel, Oberon, C++, Java), ***logische*** Programmierung (Sprachen: Prolog und Varianten).

- Die Programmiersprache ML: 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

$\mathbb{N}$:	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**),
$\mathbb{N} \times \mathbb{Z} \times \mathbb{Z}$ (**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

- Aus Abschnitt 1.2: Der ***Term***

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

  mit den (***freien***) Parametern („Platzhaltern“ für konkrete Daten) $s_{ab}, m_{ab}, s_{an}, m_{an}$ beschreibt das allgemeine „Rechenmuster“ zur Fahrtzeit-Berechnung. Er fasst Einzelberechnungen für jeweils durch vier natürliche Zahlen gegebene Uhrzeiten wie

  $(15 - 13) \cdot 60 + 53 - 10$ („Ab: 13 Uhr 10, An: 15 Uhr 53“)
  $(15 - 13) \cdot 60 + 12 - 24$ („Ab: 13 Uhr 24, An: 15 Uhr 12“)
  usw.

  zusammen und definiert eine Funktion

  $$\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N},$$
  $$(s_{ab}, m_{ab}, s_{an}, m_{an}) \mapsto (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab},$$

  (mathematisch) geschrieben auch:

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

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

- Die Funktion

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

  ist die Rechenvorschrift, d.h. der Algorithmus (im Sinne der funktionalen Programmierung) 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 Ausführungen des Algorithmus für eine konkrete Eingabe und lassen sich als Anwendungen der Funktion (***Funktionsaufrufe***) mit der Eingabe

als Argument formulieren, z.B. (in Pseudocode-Notation):

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

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 $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$).

Namen für Funktionen

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

  in mathematischer Notation:

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

  in Pseudocode:

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

  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):

$$\begin{aligned} MinStd = \textbf{function}&(m : \textbf{nat})\ \textbf{nat} \times \textbf{nat} : \\ &\textbf{result}\ \text{Umrechnung von } m \text{ Minuten in } (\text{Stunden}, \text{Minuten}) \\ &(m\ div\ 60, m\ mod\ 60) \\ Fahrtzeit = \textbf{function}&(s_{ab} : \textbf{nat}, m_{ab} : \textbf{nat}, s_{an} : \textbf{nat}, m_{an} : \textbf{nat})\ \textbf{nat} : \\ &\textbf{result}\ \text{Fahrtzeit in Minuten} \\ &(s_{an} - s_{ab}) * 60 + m_{an} - m_{ab} \\ Fahrtzeit' = \textbf{function}&(s_{ab} : \textbf{nat}, m_{ab} : \textbf{nat}, s_{an} : \textbf{nat}, m_{an} : \textbf{nat})\ \textbf{nat} \times \textbf{nat} : \\ &\textbf{result}\ \text{Fahrtzeit in } (\text{Stunden}, \text{Minuten}) \\ &MinStd(Fahrtzeit(s_{ab}, m_{ab}, s_{an}, m_{an})) \end{aligned}$$

Beachte bei $MinStd$ und $Fahrtzeit'$: Resultatdaten sind vom Typ **nat** $\times$ **nat**, d.h. 2-Tupel (***Paare***) natürlicher Zahlen.

Funktionale Algorithmen

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

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

  (die sich auf andere solche Funktionen abstützen kann) aus

  $typ_1 \times \ldots \times typ_n \rightarrow typ$.

  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:

  $betrag$.

- 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

Fallunterscheidungen

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

  Algorithmus:

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

- Allgemein:

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

  $$\textbf{if}\ b\ \textbf{then}\ t_1\ \textbf{else}\ t_2$$

  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.

Bedingungen

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

- 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 $\textbf{real} \times \textbf{real} \to \textbf{bool}$)

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

- 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:

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

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

2.3 Rekursive Funktionen

Beispiel: Fakultät

Fakultät(sfunktion): $! : \mathbb{N} \to \mathbb{N}$,
$n! = 1 \cdot 2 \cdot \ldots \cdot n \quad (0! = 1)$.

Einzelberechnungen:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 && = 0! \cdot 1 && (= 1 \cdot 0!) \\ 2! &= 1 \cdot 2 && = 1! \cdot 2 && (= 2 \cdot 1!) \\ 3! &= 1 \cdot 2 \cdot 3 && = 2! \cdot 3 && (= 3 \cdot 2!) \\ 4! &= 1 \cdot 2 \cdot 3 \cdot 4 && = 3! \cdot 4 && (= 4 \cdot 3!) \\ &\vdots \end{aligned}$$

Allgemeines „Rechenmuster“: $0! = 1$,
$n! = n \cdot (n-1)!$ für $n > 0$.

Andere Schreibweisen hierfür: $0! = 1.$
$(n+1)! = (n+1) \cdot n!\,.$

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

In Pseudocode: $fak = \textbf{function}(n : \textbf{nat})\ \textbf{nat}$:
if $n = 0$ **then** 1
else $n * fak(n-1)$

Grundprinzip:
Der Term zur Berechnung von fak enthält einen Aufruf von fak (***rekursive Definition***).

Einzelberechnung durch Funktionsaufruf (z.B.):

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

Rekursive Funktionen

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

Weitere Beispiele:

- Exponentiation: $exp : \textbf{int} \times \textbf{nat} \to \textbf{int}$, $exp(x,n) = x^n$

  $exp = \textbf{function}(x : \textbf{int}, n : \textbf{nat})\ \textbf{int}$:
  if $n = 0$ **then** 1
  else $x * exp(x, n-1)$

  Beispielrechnung:

$$\begin{aligned} exp(3,4) &= 3 \cdot exp(3,3) \\ &= 3 \cdot 3 \cdot exp(3,2) \\ &= 3 \cdot 3 \cdot 3 \cdot exp(3,1) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot exp(3,0) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot 1 \\ &= 81. \end{aligned}$$

- Fibonacci-Funktion:

  $fib = \textbf{function}(n : \textbf{nat})\ \textbf{nat}$:
  if $n = 0 \vee n = 1$ **then** 1
  else $fib(n-1) + fib(n-2)$

Beispielrechnung:

$$\begin{aligned} \mathit{fib}(4) &= \mathit{fib}(3) + \mathit{fib}(2) \\ &= \mathit{fib}(2) + \mathit{fib}(1) + \mathit{fib}(1) + \mathit{fib}(0) \\ &= \mathit{fib}(1) + \mathit{fib}(0) + 1 + 1 + 1 \\ &= 1 + 1 + 1 + 1 + 1 \\ &= 5. \end{aligned}$$

- Ackermann-Funktion:

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

Beispielrechnung:

$$\begin{aligned} ack(1, 2) &= ack(0, ack(1, 1)) \\ &= ack(0, ack(0, ack(1, 0))) \\ &= ack(0, ack(0, ack(0, 1))) \\ &= ack(0, ack(0, 2)) \\ &= ack(0, 3) \\ &= 4. \end{aligned}$$

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

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

Kein direkter rekursiver Ansatz für (1), Verallgemeinerung:

(2) Bestimme, ob n durch keine Zahl $i \in \mathbb{N}$ 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:

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

$$\begin{aligned} istprim = \textbf{function}(&n : \textbf{nat})\ \textbf{bool} : \\ &n > 1 \wedge keineteiler(n, 2) \end{aligned}$$

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:

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

Beispielrechnung:

$$\begin{aligned} gerade(5) &= ungerade(4) \\ &= gerade(3) \\ &= ungerade(2) \\ &= gerade(1) \\ &= ungerade(0) \\ &= false. \end{aligned}$$

Weitere Beispiele

1. Für $m, n \in \mathbb{N}$ soll die kleinste Primzahl k mit $m < k \leq n$ bestimmt werden (falls eine solche Primzahl existiert, andernfalls soll $n+1$ das Ergebnis der Berechnung sein).

   Algorithmus (unter Verwendung von $istprim$):

$$\begin{aligned} kleinsteprimzahl = \textbf{function}(m : \textbf{nat}, n : \textbf{nat})\,\textbf{nat} :& \\ \textbf{if } m \geq n \textbf{ then } n+1& \\ \textbf{else if } kleinsteprimzahl(m, n-1) < n& \\ \textbf{then } kleinsteprimzahl(m, n-1)& \\ \textbf{else if } istprim(n) \textbf{ then } n& \\ \textbf{else } n+1& \end{aligned}$$

   Beispielrechnung für $kleinsteprimzahl(8, 12)$:

$$\begin{aligned} & kleinsteprimzahl(8,8) = 9, \\ \text{also:}\quad & kleinsteprimzahl(8,9) = 10, \\ \text{also:}\quad & kleinsteprimzahl(8,10) = 11, \\ \text{also:}\quad & kleinsteprimzahl(8,11) = 11, \\ \text{also:}\quad & kleinsteprimzahl(8,12) = kleinsteprimzahl(8,11) \\ & \qquad\qquad\qquad\quad = 11. \end{aligned}$$

2. Für $n \in \mathbb{N}$ soll die Anzahl $anztup(n)$ der n-Tupel aus $\mathbb{N}^n$ bestimmt werden, die als Komponenten nur die Zahlen $1, 2, 3, 4$ und dabei die 1 in gerader Anzahl enthalten. ($\mathbb{N}^n = \underbrace{\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N}}_{n}$.)

   Algorithmus (unter Verwendung von exp):

$$\begin{aligned} anztup = \ & \textbf{function}(n : \textbf{nat})\ \textbf{nat} : \\ & \textbf{if } n = 0 \textbf{ then } 1 \\ & \textbf{else } exp(4, n-1) + 2 * anztup(n-1) \end{aligned}$$

Beispielrechnung:

$$\begin{aligned} anztup(3) &= exp(4,2) + 2 \cdot anztup(2) \\ &= 16 + 2 \cdot (exp(4,1) + 2 \cdot anztup(1)) \\ &= 16 + 2 \cdot (4 + 2 \cdot (exp(4,0) + 2 \cdot anztup(0))) \\ &= 16 + 2 \cdot (4 + 2 \cdot (1 + 2 \cdot 1)) \\ &= 36. \end{aligned}$$

3. Für $n \in \mathbb{N}$ soll die „ganzzahlige Wurzel“ $\lfloor \sqrt{n} \rfloor$ von n (d.i. dasjenige $k \in \mathbb{N}$ mit $k \leq \sqrt{n} < k+1$) bestimmt werden.

   Algorithmus (mit Einbettung):

$$\begin{aligned} wurzelallgemein = \ & \textbf{function}(m : \textbf{nat}, n : \textbf{nat})\ \textbf{nat} : \\ & \textbf{pre } m * m \leq n \\ & \textbf{if } (m+1) * (m+1) > n \textbf{ then } m \\ & \textbf{else } wurzelallgemein(m+1, n) \\ wurzel = \ & \textbf{function}(n : \textbf{nat})\ \textbf{nat} : \\ & wurzelallgemein(0, n) \end{aligned}$$

   Beispielrechnung:

$$\begin{aligned} wurzel(10) &= wurzelallgemein(0, 10) \\ &= wurzelallgemein(1, 10) \\ &= wurzelallgemein(2, 10) \\ &= wurzelallgemein(3, 10) \\ &= 3. \end{aligned}$$

4. Die Menge $\mathbb{N} \times \mathbb{N}$ sei wie folgt durch eine totale Ordnung angeordnet:

$$(0,0), (1,0), (0,1), (2,0), (1,1), (0,2), (3,0), (2,1), (1,2), (0,3), \ldots$$

   (Es folgen jeweils Gruppen von Paaren (x, y) mit $x + y = 0, 1, 2, 3, \ldots$ aufeinander. Jede derartige Gruppe mit $x+y = m$ beginnt mit $(m, 0)$ und zählt dann m sukzessive um 1 bis 0 herunter.)

   Zu $n \in \mathbb{N}$ soll das n-te Paar bestimmt werden (wobei $(0,0)$ als 0-tes Paar gezählt wird).

   Algorithmus:

$$\begin{aligned} paar1 = \ & \textbf{function}(n : \textbf{nat})\ \textbf{nat} : \\ & \textbf{if } n = 0 \textbf{ then } 0 \\ & \textbf{else if } paar1(n-1) = 0 \textbf{ then } paar2(n-1) + 1 \\ & \quad \textbf{else } paar1(n-1) - 1 \\ paar2 = \ & \textbf{function}(n : \textbf{nat})\ \textbf{nat} : \\ & \textbf{if } n = 0 \textbf{ then } 0 \\ & \textbf{else if } paar1(n-1) = 0 \textbf{ then } 0 \\ & \quad \textbf{else } paar2(n-1) + 1 \end{aligned}$$

$$paar = \textbf{function}(n : \textbf{nat})\ \textbf{nat} \times \textbf{nat} : \\ (paar1(n), paar2(n))$$

Beispielrechnung für $paar(4)$:

$$\begin{array}{ll} & paar1(0) = 0, \quad paar2(0) = 0, \\ \text{also:} & paar1(1) = paar2(0) + 1 = 1, \quad paar2(1) = 0, \\ \text{also:} & paar1(2) = paar1(1) - 1 = 0, \quad paar2(2) = paar2(1) + 1 = 1, \\ \text{also:} & paar1(3) = paar2(2) + 1 = 2, \quad paar2(3) = 0, \\ \text{also:} & paar1(4) = paar1(3) - 1 = 1, \quad paar2(4) = paar2(3) + 1 = 1, \\ \text{also:} & paar(4) = (1, 1). \end{array}$$

Bemerkung

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

Beispiel Exponentiation:

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

Beispielrechnung:

$$\begin{aligned} exp'(2, 6) &= exp'(4, 3) \\ &= 4 \cdot exp'(16, 1) \\ &= 4 \cdot 16 \cdot exp'(256, 0) \\ &= 4 \cdot 16 \cdot 1 \\ &= 64. \end{aligned}$$

2.4 Eigenschaften rekursiver Funktionen

Das Terminierungsproblem

- Konzept der Rekursion: Sehr mächtig, aber auch mit „Gefahren“.
- Insbesondere: Rekursion ist eine Quelle von Undefiniertheit.
  Beispiel (vgl. fak):

  $$endlos = \textbf{function}(n : \textbf{nat})\ \textbf{nat} : \\ \quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * endlos(n + 1)$$

  Es ist $endlos(0) = 1$, und für $n > 0$ ist $endlos(n)$ undefiniert.

- ***Terminierungsproblem:***
  Terminiert eine rekursive Funktion wie gewünscht (d.h. für alle Argumente aus dem beabsichtigten Definitionsbereich)?

Fundierte Relationen

- Mögliches Argument für den Nachweis der Terminierung einer rekursiven Funktion f für eine Eingabe x (informell):
  - Der Aufruf von f für x zieht eine Folge weiterer Aufrufe von f mit Argumenten $x_1, x_2, x_3, \ldots$ nach sich. Diese Folge kann nicht unendlich sein.
- Zur Formalisierung und Verallgemeinerung dieser Argumentation wird definiert:
  Eine binäre Relation $\prec$ auf einer Menge M heißt ***fundiert***, falls es keine unendliche Folge $a_0, a_1, a_2, \ldots$ von Elementen von M gibt mit $a_{i+1} \prec a_i$ für alle $i \in \mathbb{N}$.

  Beispiele:

  Fundierte Relationen auf $\mathbb{N}$:
  - $m \prec n \Leftrightarrow m < n$.
  - $m \prec n \Leftrightarrow n = m + 1$.

  Fundierte Relationen auf $\mathbb{N} \times \mathbb{N}$:
  - $(m, n) \prec (k, l) \Leftrightarrow m < k$ oder $m = k, n < l$.
  - $(m, n) \prec (k, l) \Leftrightarrow (m, n)$ steht in der im Beispiel $paar$ (Abschnitt 2.3) gegebenen Ordnung vor (k, l).

Abstiegsfunktionen

Allgemeines Schema zum Terminierungsnachweis von (nicht verschränkt) rekursiven Funktionen:
Seien

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

eine rekursiv definierte Funktion, $A \subseteq typ_1 \times \ldots \times typ_n$, M eine Menge, $\prec$ eine fundierte Relation auf M und

$$h : typ_1 \times \ldots \times typ_n \to M$$

eine Funktion (***Abstiegsfunktion (für f)***) mit folgenden Eigenschaften:
Zieht ein Aufruf $f(x_1, \ldots, x_n)$ Aufrufe $f(y_1, \ldots, y_n)$ in t nach sich, so gilt:

- Falls $(x_1, \ldots, x_n) \in A$, so $(y_1, \ldots, y_n) \in A$ (und die Auswertung von t liefert einen definierten Wert, falls die Aufrufe $f(y_1, \ldots, y_n)$ definiert sind).
- $h(y_1, \ldots, y_n) \prec h(x_1, \ldots, x_n)$.

Dann gilt: f terminiert für jedes Argument $(x_1, \ldots, x_n) \in A$ (m.a.W.: $A \subseteq D(f)$).

(Beachte: Wesentlich für die Anwendung dieser Methode ist das Auffinden einer Abstiegsfunktion mit geeignetem M und $\prec$.)

Beispiele

1. $fak = \textbf{function}(n : \textbf{nat})\, \textbf{nat} :$
   $\quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * fak(n-1)$

   Setze $A = \mathbb{N}$, und wähle $\mathbb{N}$ als M, $<$ als $\prec$ und die Abstiegsfunktion $h(n) = n$.

   Dann folgt: fak terminiert für alle $n \in \mathbb{N}$.

2. $keineteiler = \textbf{function}(n : \textbf{nat}, k : \textbf{nat})\, \textbf{bool} :$
   $\quad \textbf{pre } 2 \leq k \leq n$
   $\quad \textbf{if } k = n \textbf{ then } true \textbf{ else } n \; mod \; k \neq 0 \wedge keineteiler(n, k+1)$

   Setze $A = \{(n,k) \in \mathbb{N} \times \mathbb{N} \mid 2 \leq k \leq n\}$, und wähle $\mathbb{N}$ als M, $<$ als $\prec$ und die Abstiegsfunktion $h(n,k) = n - k$.

   Dann folgt: $keineteiler$ terminiert für alle $(n,k) \in \mathbb{N} \times \mathbb{N}$ mit $2 \leq k \leq n$.

3. $ack = \textbf{function}(m : \textbf{nat}, n : \textbf{nat})\, \textbf{nat} :$
   $\quad \textbf{if } m = 0 \textbf{ then } n + 1$
   $\quad \textbf{else if } n = 0 \textbf{ then } ack(m-1, 1)$
   $\quad \textbf{else } ack(m-1, ack(m, n-1))$

   Setze $A = \mathbb{N} \times \mathbb{N}$, und wähle $\mathbb{N} \times \mathbb{N}$ als M,

   $$(m,n) \prec (k,l) \Leftrightarrow m < k \text{ oder } m = k, n < l$$

   und die Abstiegsfunktion $h(m,n) = (m,n)$.

   Dann folgt: ack terminiert für alle $(m,n) \in \mathbb{N} \times \mathbb{N}$.

Bemerkung

Sei

$$\begin{aligned} unklar = \textbf{function}(n : \textbf{nat})\, \textbf{nat} :& \\ &\textbf{if } n = 0 \vee n = 1 \textbf{ then } 1 \\ &\textbf{else if } n \; mod \; 2 = 1 \textbf{ then } unklar(3 * n + 1) \\ &\quad \textbf{else } unklar(n \; div \; 2) \end{aligned}$$

Es ist ein offenes Problem, ob $unklar$ für jedes Argument $n \in \mathbb{N}$ terminiert.

Weitere Eigenschaften rekursiver Funktionen

Die Terminierung ist eine wichtige nicht-triviale Eigenschaft von rekursiven Funktionen. Allgemein können auch andere Eigenschaften relevant (und nicht offensichtlich) sein.

Beispiele:

- ***Korrektheit:***
  Die rekursive Funktion löst die gestellte Aufgabe.
- „Wertverlauf"-Eigenschaften (Monotonie, Beschränktheit, ...).
- Aussagen über die benötigte Anzahl von Ausführungen von Basisfunktionen.

Nachweis solcher Eigenschaften: Durch formale Beweise möglich und eventuell angebracht. Grundlegendes Beweismittel: ***Induktion***. (Nachweis von Terminierung und/oder Korrektheit: ***Programmverifikation***.)

Das Induktionsprinzip

Allgemeines Schema zum Nachweis, dass für alle Elemente einer Menge A eine bestimmte Eigenschaft $\mathcal{E}$ gilt:
Sei M eine Menge, $\prec$ eine fundierte Relation auf M und $h : A \to M$ eine Funktion, so dass für alle $x \in A$ gilt:

- Aus der Annahme (***Induktionsvoraussetzung***), dass $\mathcal{E}$ für alle $y \in A$ mit $h(y) \prec h(x)$ gilt, folgt, dass $\mathcal{E}$ auch für x gilt (***Induktionsschluss***).

Dann gilt $\mathcal{E}$ für alle $x \in A$.

Sprechweise bei konkreter Anwendung dieses Beweisprinzips: ***Induktion nach*** $h(x)$ (***bzgl.*** $\prec$ ***auf*** M).

Insbesondere: $A \subseteq \mathbb{N}$, $M = \mathbb{N}$, $m \prec n \Leftrightarrow m < n$ oder $m \prec n \Leftrightarrow n = m + 1$, $h(n) = n$: ***Induktion nach*** n (***vollständige Induktion***).
Beispiel ($A = \mathbb{N}$, $m \prec n \Leftrightarrow n = m + 1$):

Zu zeigen:

- $\mathcal{E}$ gilt für 0 (ohne Voraussetzungen).
- Gilt $\mathcal{E}$ für n, so auch für $n + 1$.

Dann gilt $\mathcal{E}$ für alle $n \in \mathbb{N}$.

Bemerkung

Das oben angegebene allgemeine Schema für Terminierungsbeweise von rekursiven Funktionen

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

kann selbst durch Induktion gerechtfertigt werden (ist also ein Spezialfall des Induktionsprinzips):
Die Behauptung ist:

$f(x_1, \ldots, x_n)$ ist für alle $(x_1, \ldots, x_n) \in A$ definiert.

Aus den angegebenen Voraussetzungen folgt mit der Induktionsvoraussetzung, dass die Aufrufe $f(y_1, \ldots, y_n)$, die $f(x_1, \ldots, x_n)$ nach sich zieht, definiert sind und somit die Auswertung von t einen definierten Wert für $f(x_1, \ldots, x_n)$ ergibt.

Beispiele

1. Behauptung: Für die Ackermann-Funktion

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

   gilt:

   $ack(m, n) > n$ für alle $m, n \in \mathbb{N}$ (d.h.: für alle $(m, n) \in \mathbb{N} \times \mathbb{N}$).

   Induktionsbeweis:

   - $A = \mathbb{N} \times \mathbb{N}$, $M = \mathbb{N} \times \mathbb{N}$, $(m, n) \prec (k, l) \Leftrightarrow m < k$ oder $m = k, n < l$, $h(m, n) = (m, n)$. (Induktion nach (m, n) bzgl. $\prec$ auf $\mathbb{N} \times \mathbb{N}$.)
   - Falls $ack(m', n') > n'$ für $(m', n') \prec (m, n)$, so $ack(m, n) > n$.
   - Damit: $ack(m, n) > n$ für alle $(m, n) \in \mathbb{N} \times \mathbb{N}$.

2. Behauptung: Für die Fibonacci-Funktion

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

   und $n \in \mathbb{N}$ sei $g(n)$ die Anzahl der auszuführenden Additionen bei einer Berechnung von $\mathit{fib}(n)$. Es gilt:

   $g(n) \leq 2^{n-2}$ für alle $n \in \mathbb{N}$ mit $n \geq 2$.

   Induktionsbeweis:

   - Induktion nach n. ($A = \{n \in \mathbb{N} \mid n \geq 2\}$.)
   - Falls $g(m) \leq 2^{m-2}$ für $m < n, m \geq 2$, so $g(n) \leq 2^{n-2}$ für $n \geq 2$.
   - Damit: $g(n) \leq 2^{n-2}$ für alle $n \in \mathbb{N}$ mit $n \geq 2$.

3. Behauptung: Die Funktion

   $$\begin{aligned} ggt = \textbf{function}(m : \textbf{nat}, n : \textbf{nat})\ \textbf{nat} :\\ \quad \textbf{pre } m > 0, n > 0\\ \quad\ \textbf{if } m = n \textbf{ then } m\\ \quad\ \textbf{else if } m > n \textbf{ then } ggt(m - n, n)\\ \quad\ \textbf{else } ggt(m, n - m) \end{aligned}$$

   (***Euklidischer Algorithmus***) berechnet den größten gemeinsamen Teiler von m und n, d.h. für $m, n > 0$ gilt $ggt(m, n) = g(m, n)$, wobei $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definiert ist durch ($m, n > 0$):

$$g(m,n) = l \Leftrightarrow \begin{array}{l} l \text{ ist Teiler von } m \text{ und } n, \\ \text{und für jeden Teiler } k \text{ von } m \text{ und } n \text{ ist } k \leq l. \end{array}$$

Induktionsbeweis:

- Induktion nach $m+n$. ($A = \{(m,n) \in \mathbb{N} \times \mathbb{N} \mid m > 0, n > 0\}$, $h(m,n) = m+n$.)
- Falls $ggt(m',n') = g(m',n')$ für m', n' mit $m'+n' < m+n$, $m', n' > 0$ gilt, so gilt $ggt(m,n) = g(m,n)$ für m, n mit $m, n > 0$.
- Damit: $ggt(m,n) = g(m,n)$ für alle $m, n \in \mathbb{N}$ mit $m, n > 0$.

Totale und partielle Korrektheit

- Die (Korrektheits-) Aussage im letzten Beispiel beinhaltet auch die Terminierung für die beabsichtigte Menge möglicher Eingaben. Dies bedeutet genauer die ***totale Korrektheit*** der Funktion ggt.

  Allgemein: f ist total korrekt bezüglich einer (zu berechnenden) Funktion g, wenn für alle Argumente x, für die g definiert ist, gilt: f terminiert für x, und es ist $f(x) = g(x)$.

- Eine schwächere Eigenschaft ist die ***partielle Korrektheit***: Es wird nur gefordert, dass $f(x) = g(x)$ gilt für alle Argumente $x \in D(g)$, für die f terminiert.

  Offensichtlich gilt:

  „partielle Korrektheit + Terminierung = totale Korrektheit“.

Eine Beweismethode für partielle Korrektheit

Die partielle Korrektheit einer rekursiven Funktion kann man häufig unabhängig von der Terminierung und mit anderen Methoden beweisen, z.B. nach folgendem allgemeinen Schema:

Seien

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

eine (nicht verschränkt) rekursiv definierte Funktion und

$$g : typ_1 \times \ldots \times typ_n \rightarrow typ.$$

Für alle $(x_1, \ldots, x_n) \in D(g)$ gelte:

- Aus der Annahme, dass $f(y_1, \ldots, y_n) = g(y_1, \ldots, y_n)$ für alle von dem Aufruf $f(x_1, \ldots, x_n)$ hervorgerufenen Aufrufe $f(y_1, \ldots, y_n)$ mit $(y_1, \ldots, y_n) \in D(g)$ gilt, folgt $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$.

Dann gilt:
Für alle $(x_1, \ldots, x_n) \in D(g)$, für die f terminiert, gilt $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ (m.a.W.: f ist partiell korrekt bezüglich g).

Beispiele

1. Behauptung: Für die Funktion

$$\begin{aligned} unklar = \textbf{function}(n : \textbf{nat})\ \textbf{nat} :\\ \quad \textbf{if } n = 0 \vee n = 1 \textbf{ then } 1\\ \quad \textbf{else if } n\ mod\ 2 = 1 \textbf{ then } unklar(3 * n + 1)\\ \quad\quad \textbf{else } unklar(n\ div\ 2) \end{aligned}$$

   (s.o.) gilt $unklar(n) = 1$ ($= g(n)$) für alle $n \in \mathbb{N}$, für die $unklar$ terminiert.

   Beweis: Sei $n \in \mathbb{N} = D(g)$ (damit: $3 * n + 1 \in \mathbb{N}$ und $n\ div\ 2 \in \mathbb{N}$).

   - Aus $unklar(3 * n + 1) = 1$ und $unklar(n\ div\ 2) = 1$ folgt $unklar(n) = 1$.
   - Damit: $unklar(n) = 1$ für alle $n \in \mathbb{N}$, für die $unklar$ terminiert.

2. Die oben gezeigte totale Korrektheit der Funktion ggt lässt sich alternativ beweisen durch den Nachweis der beiden Behauptungen

   a) ggt ist partiell korrekt bezüglich der im ggt-Beispiel angegebenen Funktion g.

   b) ggt terminiert für alle $m, n > 0$.

   Beweis von a): Aus $ggt(m-n, n) = g(m-n, n)$ und $ggt(m, n-m) = g(m, n-m)$ folgt $ggt(m, n) = g(m, n)$. Daraus folgt die Behauptung.

   Beweis von b): Mit der Abstiegsfunktion $h(m, n) = m + n$.

2.5 Polymorphe Funktionen

Polymorphe Typen

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

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

Beispiele: $\boldsymbol{\alpha}$,
$\boldsymbol{\beta} \times \boldsymbol{\gamma} \times$ **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**$(x : \boldsymbol{\alpha}, y : \boldsymbol{\beta})\, \boldsymbol{\beta} \times \boldsymbol{\alpha} : (y, x)$

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

2.6 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** $\rightarrow$ **int)**:

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

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

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

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

Funktionen als Argumente

- In wichtigen Anwendungsaufgaben: Algorithmus, der bestimmte Werte allgemein für „beliebige“ Funktionen berechnen soll.
- Dazu: Funktionen mit Funktionen als Argumenten. (Mit dieser Möglichkeit, werden Funktionen höherer Ordnung zu einem sehr mächtigen Konzept.)

Beispiele:

1. Berechnung des Differenzenquotienten für eine Funktion:

   $$\begin{aligned} diffquot = \textbf{function}&(f : \textbf{real} \rightarrow \textbf{real}, x : \textbf{real}, delta : \textbf{real})\ \textbf{real} : \\ &(f(x + delta) - f(x))/delta \end{aligned}$$

   Möglicher Aufruf:

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

   Alternativ:

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

2. Näherungsweise Berechnung der Nullstelle einer Funktion:

   $$\begin{aligned} nullstelle = \textbf{function}&(f : \textbf{real} \rightarrow \textbf{real}, a : \textbf{real}, b : \textbf{real}, eps : \textbf{real})\ \textbf{real} : \\ &\textbf{pre}\ a < b, f(a) < 0, f(b) > 0, eps > 0 \\ &\textbf{let} \\ &c = (a + b)/2 \\ &\textbf{in} \\ &\textbf{if}\ b - a < 2 * eps\ \textbf{then}\ c \\ &\textbf{else if}\ f(c) = 0\ \textbf{then}\ c \\ &\quad \textbf{else if}\ f(c) > 0\ \textbf{then}\ nullstelle(f, a, c, eps) \\ &\quad\quad \textbf{else}\ nullstelle(f, c, b, eps) \\ &\textbf{end} \end{aligned}$$

Einige Grundalgorithmen höherer Ordnung

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 $\circ$:

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

   Anwendungsbeispiel (vgl. Abschnitt 2.1):

$$Fahrtzeit' = comp(MinStd, Fahrtzeit)$$

2. n-fache Iteration einer Funktion

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

   Algorithmus:

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

   Anwendungsbeispiel:

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

Funktionen, Konstanten, Terme, Werte

- Algorithmus ist gegeben durch Deklarationen der Art

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

  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. (Genaueres hierzu: Abschnitt 3.4.)

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 (in SML-Terminologie: ***Ausdrücke***) 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** $f3 \ldots$

Beispiel:

```
- fun gerade n = if n=0 then true else ungerade(n-1)
  and ungerade n = if n=0 then false else gerade(n-1)
```

- 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 = $ **fn** $y \Rightarrow$ **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

Beschränkung im Folgenden:

- Namen werden nicht mehrfach deklariert. Dies wäre (in SML zwar möglich, aber) kein rein funktionales Konzept.
- Der Name it wird nicht in Termen verwendet.

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
```

- **real** umfasst eine Teilmenge der Menge $\mathbb{R}$ (endliche Dezimalbrüche). Bedeutung von 2.3E~1: $2.3 \cdot 10^{-1}$.

- Bei **int** und **real**: In der Praxis nur Darstellungen gewisser maximaler Stellenzahlen möglich (durch jeweiligen Computer bestimmt). Bei Rechenoperationen ergibt sich dadurch die Gefahr von ***Überläufen***. Beispiel (Erklärung: siehe Abschnitt 3.6.):

```
- val a = 800000000
  val adopp = 2*a;
val a = 800000000 : int
uncaught exception overflow
```

- Bei Rechenoperationen mit **real**-Daten können Rundungsfehler auftreten. Beispiel:

```
- val a = 3E14
  val b = a-0.05
  val c = a-b;
val a = 3E14 : real
val b = 3E14 : real
val c = 0.0625 : real
```

- Zeichen sind Elemente einer (für Computer-Anwendungen normierten) Zeichenmenge (dem ***ASCII-Zeichensatz***) und werden in #" und " eingeschlossen dargestellt. Beispiel:

```
- val a = #"a";
val a = #"a" : char
```

- Texte sind aus beliebigen Zeichen zusammengesetzt und werden in " eingeschlossen dargestellt. Beispiel:

```
- val a = "abc"
  val b = "a";
val a = "abc" : string
val b = "a" : string
```

- Die angegebenen Datendarstellungen können als „Namen“ für Werte und damit als (vorgegebene, ***spezielle***) Konstanten im Sinne der Abschnitte 2.1 und 2.6 angesehen werden.

Der ASCII-Zeichensatz

Der ASCII-Zeichensatz enthält die nachfolgenden 128 Zeichen (mit zugehörigen ***Codenummern***).

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

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

Basisfunktionen

Für die Basistypen stehen die nachfolgenden Basisfunktionen zur Verfügung.

- ***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**

Beispiel:

```
- not (10-2 < 7) andalso "abc"="abc"
                  (* Zur Klammerung: Siehe Abschnitt 3.4 *);
val it = true : bool
- "Gu"^"ten"^" "^"Tag"
val it = ``Guten Tag'' : 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 $x_1 \ldots x_m$ und $y_1 \ldots y_n$ (mit Zeichen $x_1, \ldots, x_m, y_1, \ldots, y_n$ des ASCII-Zeichensatzes, $m, n \geq 0$) wie folgt (rekursiv) definiert:

$x_1 \ldots x_m < y_1 \ldots y_n$ genau dann, wenn
entweder: $m = 0$ und $n > 0$
oder: $m, n > 0$ und die Codezahl von x_1 ist kleiner als die von y_1
oder: $m, n > 0$ und $x_1 = y_1$ und $x_2 \ldots x_m < y_2 \ldots y_n$.

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

Bemerkungen

- 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***, ein ***Wort***) über einem Alphabet Σ ist eine endliche Folge von Elementen $\sigma_1, \ldots, \sigma_n$ von Σ, $n \in \mathbb{N}$. (Schreibweise: $\sigma_1\sigma_2 \ldots \sigma_n$.) Für $n = 0$ ist die Folge die ***leere Zeichenreihe*** (Schreibweise: ε).
- Eine ***formale Sprache*** über einem Alphabet Σ ist eine Teilmenge der Menge Σ^* aller Zeichenreihen über Σ.

Syntax von 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

Alphanumerische Identifikatoren sind Zeichenreihen über dem Alphabet

$$\{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \texttt{'}, \texttt{\_}\}$$

gemäß folgenden 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′) $\langle \textit{Buchst} \rangle$ ::= 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

(c′) $\langle \textit{Ziffer} \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 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 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 (ebenfalls gemäß diesen Regeln gebildeten) 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) $\left[\beta\right]$ wird durch eines der β_i ($i = 1, \ldots, n$) ersetzt oder ersatzlos gestrichen.

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: $\mathcal{L}(\langle \mathit{anId} \rangle)$.

Z.B.: `a_1` $\in \mathcal{L}(\langle \mathit{anId} \rangle)$ gemäß folgender Ableitung von `a_1` aus $\langle \mathit{anId} \rangle$:

$$
\begin{aligned}
\langle \mathit{anId} \rangle &\rightsquigarrow \langle \mathit{Buchst} \rangle \left\{ \langle \mathit{Buchst} \rangle \mid \langle \mathit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\}^* && \text{mit (1)} \\
&\rightsquigarrow \texttt{a} \left\{ \langle \mathit{Buchst} \rangle \mid \langle \mathit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\}^* && \text{mit (1)} \\
&\rightsquigarrow \texttt{a} \left\{ \langle \mathit{Buchst} \rangle \mid \langle \mathit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\} \left\{ \langle \mathit{Buchst} \rangle \mid \langle \mathit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\} && \text{mit (3)} \\
&\rightsquigarrow \texttt{a\_} \left\{ \langle \mathit{Buchst} \rangle \mid \langle \mathit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\} && \text{mit (2)} \\
&\rightsquigarrow \texttt{a\_} \langle \mathit{Ziffer} \rangle && \text{mit (2)} \\
&\rightsquigarrow \texttt{a\_1} && \text{mit (1)}
\end{aligned}
$$

Zusätzliche syntaktische Einschränkungen

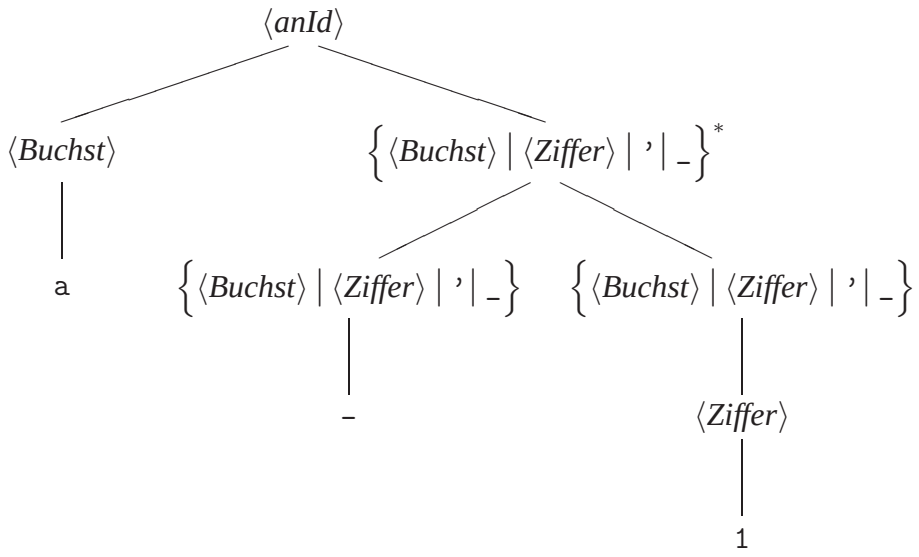
Manche syntaktischen Festlegungen lassen sich nicht (oder nur sehr schwer) innerhalb einer BNF-Grammatik beschreiben. Diese werden durch zusätzliche, verbal formulierte ***Kontextbedingungen*** angegeben.

Komplette Syntaxdefinition im Beispiel:
Alphanumerische Identifikatoren sind definiert durch die BNF-Grammatik für $\langle \mathit{anId} \rangle$ mit der Kontextbedingung:

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

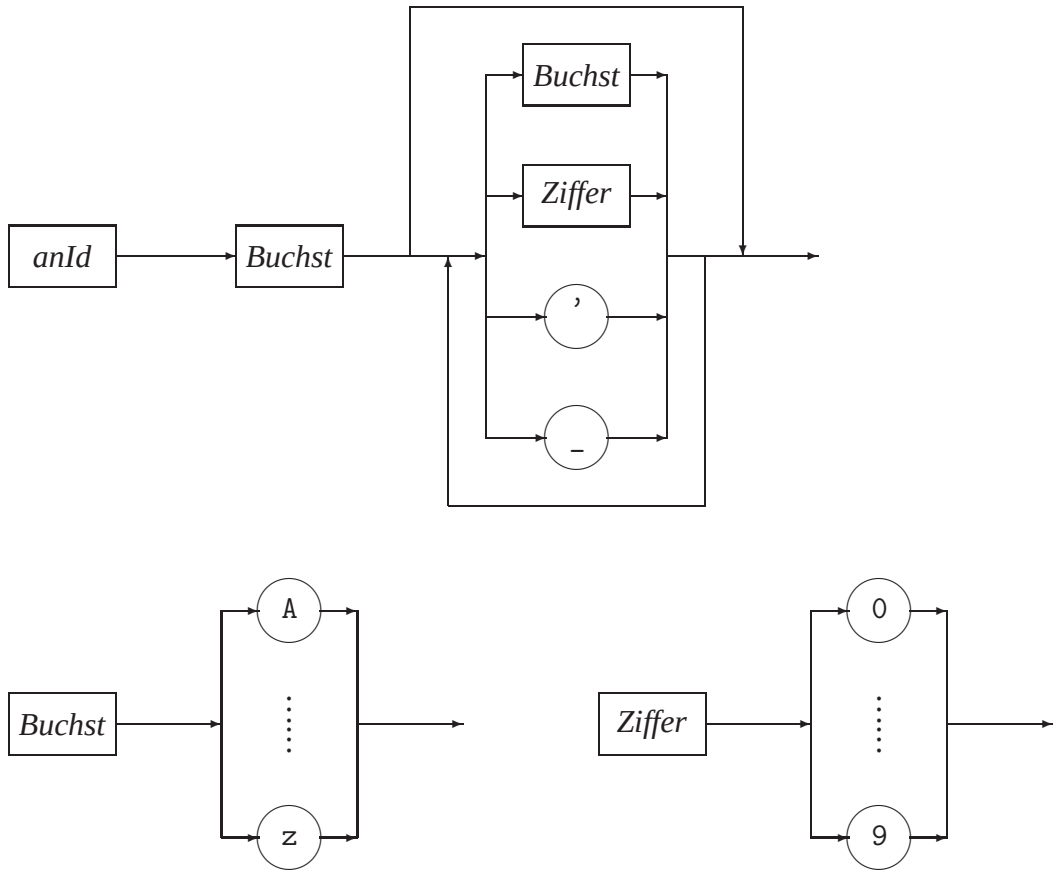
Bemerkung

Ableitungen in BNF-Grammatiken lassen sich grafisch durch ***Ableitungsbäume*** veranschaulichen, z.B. die Ableitung von `a_1` aus $\langle \mathit{anId} \rangle$ durch:



Syntaxdiagramme

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



Allgemein:

- Terminalzeichen in $\bigcirc$,
- Für jedes Nicht-Terminalzeichen A (in $\boxed{}$) ein Diagramm; eine Zeichenreihe aus $\mathcal{L}(A)$ erhält man, indem man in dem Diagramm für A einen beliebigen Weg von $\boxed{A}$ 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`

$$
\begin{array}{lll}
\langle \textit{intKonst} \rangle & ::= & \left[\texttt{~} \right] \left\{ \langle \textit{Ziffer} \rangle \right\}^{+} \\
\langle \textit{realKonst} \rangle & ::= & \langle \textit{intKonst} \rangle \, . \left\{ \langle \textit{Ziffer} \rangle \right\}^{+} \mid \langle \textit{intKonst} \rangle \left[\, . \left\{ \langle \textit{Ziffer} \rangle \right\}^{+} \right] \texttt{E} \, \langle \textit{intKonst} \rangle \\
\langle \textit{charKonst} \rangle & ::= & \texttt{\#"} \langle \textit{Zeichen} \rangle \texttt{"} \\
\langle \textit{stringKonst} \rangle & ::= & \texttt{"} \left\{ \langle \textit{Zeichen} \rangle \right\}^{*} \texttt{"} \\
\langle \textit{Zeichen} \rangle & ::= & \langle \textit{Buchst} \rangle \mid \langle \textit{Ziffer} \rangle \mid \langle \textit{Symbol} \rangle \mid \langle \textit{sonstZeich} \rangle \mid \\
\langle \textit{Name} \rangle & ::= & \langle \textit{anId} \rangle \mid \langle \textit{symbId} \rangle \\
\langle \textit{anId} \rangle & ::= & \langle \textit{Buchst} \rangle \left\{ \langle \textit{Buchst} \rangle \mid \langle \textit{Ziffer} \rangle \mid \texttt{'} \mid \texttt{\_} \right\}^{*} \\
\langle \textit{symbId} \rangle & ::= & \left\{ \langle \textit{Symbol} \rangle \right\}^{+} \\
\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} \\
\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} \\
\langle \textit{Symbol} \rangle & ::= & \texttt{!} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{\$} \mid \texttt{\#} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{:} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{?} \mid \texttt{@} \mid \texttt{\textbackslash} \mid \texttt{~} \mid \texttt{`} \mid \texttt{\^{}} \mid \texttt{|} \\
\langle \textit{sonstZeich} \rangle & ::= & \texttt{'} \mid \texttt{(} \mid \texttt{)} \mid \texttt{,} \mid \texttt{.} \mid \texttt{;} \mid \texttt{[} \mid \texttt{]} \mid \texttt{\_} \mid \texttt{\{} \mid \texttt{\}}
\end{array}
$$

3.4 Termauswertung

Semantik von SML-Funktionen

- Die Semantik ordnet einem Programm einer Programmiersprache eine (formale) Bedeutung zu.
- Für SML heißt das (hier; vgl. Abschnitt 7.1) im Wesentlichen: Die Semantik bestimmt die Bedeutung (im Sinne von Abschnitt 2.6: den Wert) f^{sem} von SML-Funktionen f.
- Dazu allgemeiner nötig: ***Auswertung*** von (allgemeinen) Termen.

Präzedenz- und Assoziierungsregeln für Infix-Operatoren

Zur eindeutigen Festlegung der „syntaktischen Struktur“ von Termen wird festgelegt:

- 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***.

  Ist insbesondere a Name einer Funktion, so ist w die (durch die SML-Semantik bestimmte) Funktion a^{sem}.

- Ein Term t kann in (Abhängigkeit von) einer Umgebung U ausgewertet werden. Liefert dies ein Ergebnis, so erhält man einen Wert $W^U(t)$. (Andernfalls: $W^U(t)$ ***undefiniert***.)

- Zu Beginn einer Sitzung ist die Umgebung leer. Jede Wertdeklaration erweitert die jeweils aktuelle Umgebung U wie folgt:

  – Ist die Wertdeklaration von der Form **val** $a = t$ und ist $W^U(t)$ definiert, so wird U die neue Bindung <a, $W^U(t)$> hinzugefügt. (Ist $W^U(t)$ undefiniert, so liefert das System eine Fehlermeldung.)

  – Ist die Wertdeklaration von der Form **val rec** $a =$ **fn** $(x_1, \ldots, x_n) \Rightarrow t'$ (im Fall $n = 1$ auch ohne Klammern), so wird U die neue Bindung <a, a^{sem}> hinzugefügt, wobei a^{sem} die durch

  $$a^{sem}(w_1, \ldots, w_n) = W^{U \cup \{<x_1,w_1>,\ldots,<x_n,w_n>,<a,a^{sem}>\}}(t')$$

  (rekursiv) definierte Funktion ist.

  (Für verschränkt rekursive Funktionen: analog.)

Auswertung

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

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

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

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

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

$$t^{sem}(w_1, \ldots, w_n) = W^{U \cup \{<x_1,w_1>,\ldots,<x_n,w_n>\}}(t')$$

definierte Funktion t^{sem}.

(5) 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 $W^U(t_1)$ und $W^U(t_2)$ definiert, so ist $W^U(t) = -W^U(t_1)$ bzw. $W^U(t) = \neg W^U(t_1)$ bzw. $W^U(t) = W^U(t_1) \oplus W^U(t_2)$ (falls $W^U(t_1) \oplus W^U(t_2)$ definiert ist). Andernfalls ist $W^U(t)$ undefiniert.

(6) Sei t eine Funktionsanwendung der Form $t_1 t_2$. $W^U(t_1)$ sei die Funktion t_1^{sem} und $W^U(t_2) = w$. Sind diese beiden Werte definiert und ist $w \in D(t_1^{sem})$, so ist $W^U(t) = t_1^{sem}(w)$. Andernfalls ist $W^U(t)$ undefiniert.

(7) Ist t von der Form **let** $wd_1 \ldots\ wd_n$ **in** t' **end** mit Wertdeklarationen $wd_1 \ldots\ wd_n$, so sei:

$U_1 =$ Erweiterung von U gemäß wd_1,
$U_2 =$ Erweiterung von U_1 gemäß wd_2,
$\vdots$
$U_n =$ Erweiterung von U_{n-1} gemäß wd_n.

Sind alle Auswertungen in diesen Erweiterungen sowie $W^{U_n}(t')$ definiert, so ist $W^U(t) = W^{U_n}(t')$. Andernfalls ist $W^U(t)$ undefiniert.

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

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

Beispiele

1. Sitzung: | Umgebung nach Eingabe (zu Beginn: $\emptyset$):

`val a = 3;`	$U_1 = \{<a, 3>\}$
`fun adda x = x+a;`	$U_2 = \{<a, 3>, <adda, adda^{sem}>\}$ (mit $adda^{sem}(w) = W^{U_1 \cup \{<x,w>\}}(x + a)$)
`adda(2);`	

Auswertung von $adda(2)$ in U_2:

Es ist $W^{U_2}(adda) = adda^{sem}$ und $W^{U_2}(2) = 2$ gemäß (2) bzw. (1), also:

$$\begin{aligned} W^{U}(adda(2)) &\overset{(6)}{=} adda^{sem}(2) \\ &= W^{U_1 \cup \{<x,2>\}}(x + a) \\ &\overset{(5)}{=} W^{U_1 \cup \{<x,2>\}}(x) + W^{U_1 \cup \{<x,2>\}}(a) \\ &\overset{(2)}{=} 2 + 3 \\ &= 5. \end{aligned}$$

2. Sitzung: Umgebung nach Eingabe (zu Beginn: $\emptyset$):

```
val a = 5;
let
val z1=a+3
val z2=z1*z1
in z1*z2 end;
```

$U = \{<a, 5>\}$

Auswertung von **let** ... **in** ... **end** in U:

Es ist: $W^{U}(a + 3) = 8$, also: $U_1 = \{<a, 5>, <z1, 8>\}$,
$W^{U_1}(z1 * z1) = 64$, also: $U_2 = \{<a, 5>, <z1, 8>, <z2, 64>\}$.

Mit (7) ist dann:

$$W^{U}(\textbf{let} \ldots \textbf{in} \ldots \textbf{end}) = W^{U_2}(z1 * z2) = 8 \cdot 64 = 512.$$

3. Die Funktionsvereinbarung

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

(zu Beginn einer Sitzung) erzeugt die Umgebung $U = \{<fak, fak^{sem}>\}$ mit

$$fak^{sem}(w) = W^{\{<n,w>,<fak,fak^{sem}>\}}(\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * fak(n-1)).$$

Auswertung eines Aufrufs $fak(3)$ in U:

$$\begin{aligned} &W^{U}(fak(3)) \\ &= fak^{sem}(3) \\ &= W^{\{<n,3>,<fak,fak^{sem}>\}}(\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * fak(n-1)) \\ &= W^{\{<n,3>,<fak,fak^{sem}>\}}(n * fak(n-1)) \\ &= W^{\{<n,3>,<fak,fak^{sem}>\}}(n) \cdot W^{\{<n,3>,<fak,fak^{sem}>\}}(fak(n-1)) \\ &= 3 \cdot fak^{sem}(2) \\ &= 3 \cdot W^{\{<n,2>,<fak,fak^{sem}>\}}(\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * fak(n-1)) \\ &\vdots \\ &= 3 \cdot (2 \cdot (1 \cdot W^{\{<n,0>,<fak,fak^{sem}>\}}(\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * fak(n-1)))) \\ &= 3 \cdot (2 \cdot (1 \cdot W^{\{<n,0>,<fak,fak^{sem}>\}}(1))) \\ &= 3 \cdot (2 \cdot (1 \cdot 1)) \\ &= 6. \end{aligned}$$

Bemerkungen

- Die durch die obigen Regeln beschriebene Auswertung ist formal eine rekursiv definierte Funktion

  $$W : \text{Menge von Umgebungen} \times \text{Menge von Termen} \to \text{Menge von Werten}.$$

  Die Rekursion verläuft nach dem ***Termaufbau***, d.h. der „syntaktischen Struktur“ der Terme einschließlich der Präzedenzen und Assoziierungen: Der Wert eines Terms greift i. Allg. auf Werte von „Teiltermen“ zurück.

  M.a.W.: Eine adäquate fundierte Relation für die Anwendung des Induktionsprinzips im Sinne von Abschnitt 2.4 (***strukturelle Induktion***) ist (auf der Menge der Terme) gegeben durch:

  $$t_1 \prec t_2 \Leftrightarrow t_1 \text{ ist Teilterm von } t_2.$$

- Eine Termauswertung gemäß der Funktion W bildet insbesondere die schon in Kapitel 2 informell durchgeführten Berechnungen für rekursive Funktionen formal nach.

  Beispiel: Auswertung des Aufrufs $fak(3)$ (s.o.), verkürzt aufgeschrieben als

  $$\begin{aligned} W^U(fak(3)) = fak^{sem}(3) &= 3 \cdot fak^{sem}(2) \\ &= 3 \cdot (2 \cdot fak^{sem}(1)) \\ &= 3 \cdot (2 \cdot (1 \cdot fak^{sem}(0))) \\ &= 3 \cdot (2 \cdot (1 \cdot 1)) \\ &= 6. \end{aligned}$$

  Dies entspricht genau der entsprechenden „Auswertung“ in Abschnitt 2.3.

  Allgemein (für beliebige Terme) kann man eine Auswertung vereinfacht auffassen als eine Folge von Ersetzungen: Ein Term wird ausgewertet, indem Teilterme „sukzessive“ durch deren Werte ersetzt werden (***Substitutionsmodell***).

Strikte und verzögerte Auswertung

- Charakteristisch für die Auswertung von Funktionsanwendungen gemäß den Auswertungsregeln (5) und (6): Es werden erst alle Argumente ausgewertet und dann in die Funktion eingesetzt. Der Wert des Funktionsaufrufs ist undefiniert, wenn nicht alle Argumente definiert sind (***strikte Auswertung***, ***Wertaufruf***, ***call-by-value***).

- Ausgenommen von der strikten Auswertung sind bedingte Terme (aufgefasst als Funktionsanwendungen) und Terme mit den Booleschen Operationen **andalso** und **orelse**. Bei deren ***verzögerter Auswertung*** (***lazy evaluation***, ***call-by-need***) gemäß den Regeln (8) und (9) werden nicht benötigte Terme nicht ausgewertet. Dies bewirkt z.B., dass der Wert von **if** b **then** t_1 **else** t_2 definiert sein kann, obwohl einer der beiden Terme t_1 oder t_2 undefiniert ist.

- Sprechweise auch: Die drei Konstruktionen **if** ... **then** ... **else** ..., **andalso** und **orelse** stellen ***nicht-strikte*** Funktionen dar. Alle sonstigen Funktionen werden als ***strikt*** aufgefasst.

Terminierung und Korrektheit rekursiver Funktionen

Terminierung bzw. Korrektheit einer rekursiven Funktion f im Sinne der Ausführungen in Abschnitt 2.4 lassen sich für die Beschreibung von f in SML mit Hilfe des Auswertungskonzepts wie folgt formal angeben. (Analoges gilt auch für andere Eigenschaften von f.)

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

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

  Die partielle Korrektheit bedeutet, dass alle Auswertungen, die nach endlich vielen Schritten zu Ende kommen, die gewünschten Werte liefern.

Gemäß den Bemerkungen über den Zusammenhang der Auswertungsfunktion W mit informellen Auswertungen in Abschnitt 2.4 übertragen sich die dort (in „SML-unabhängiger“ Schreibweise) beschriebenen Beweismethoden und konkreten Beweise in trivialer Weise auf SML-Funktionen.

Zur Realisierung der Termauswertung

- In der praktischen Realisierung der Termauswertung durch ein SML-System (die in Teilen nach dem Schema der Funktion W verläuft) 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):	$\mathit{Diff}(\mathit{Sum}(10, \mathit{Prod}(x, 85)), \mathit{Sum}(134, y))$

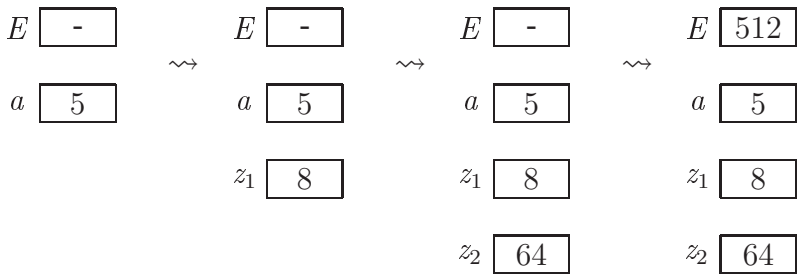
  (Daraus ersichtlich: $x * 85$ ist Teilterm des Gesamtterms, nicht jedoch $10 + x$.)

- Heute gebräuchliche Rechner folgen der ***von-Neumann-Rechner-Architektur***. Sie besitzen einen (aus einzelnen ***Speicherzellen*** bestehenden) ***Speicher*** zur Aufnahme von (insbesondere) Daten. Dies wird bei der Termauswertung auf einem Rechner zur Realisierung der Bindungen von Werten an Namen genutzt.

  Beispiel (vgl. obiges Auswertungsbeispiel): Zu Beginn der Auswertung des Terms

  let
  val $z_1 = a + 3$
  val $z_2 = z_1 * z_1$
  in $z_1 * z_2$ **end**

  steht in einer a zugeordneten Speicherzelle der aktuelle Wert von a (z.B.: 5). Die Auswertung durchläuft dann (stark vereinfacht dargestellt) folgende Schritte (mit weiteren Speicherzellen für z_1, z_2 und das Ergebnis E):



Funktionale – imperative Programmierung

- Allgemein kann in einem von-Neumann-Rechner nicht nur eine Speicherzelle mit einem Wert „gefüllt“ werden, dieser Wert kann auch verändert werden. Die jeweiligen Inhalte des Speichers bestimmen seinen ***Zustand***. Die Verarbeitungsschritte des Rechners bestehen (u.a.) darin, den Zustand zu verändern.
- Anhand dieser grundlegenden Eigenschaft von von-Neumann-Rechnern kann ein grundlegender Unterschied zwischen funktionaler und imperativer Programmierung erläutert werden:
  - Funktionale Programmierung nimmt keinerlei Bezug auf die Realisierung von Berechnungen mit Hilfe von Speicherplätzen.
  - Imperative Programmierung nimmt direkten Bezug auf Speicher und Zustände und formuliert Algorithmen mit ***Anweisungen***, die Zustandsänderungen (auf „höherer Ebene“ als in direkter Maschinensprache) explizit beschreiben.
- Die Schritte in obigem Auswertungsbeispiel sind Zustandsänderungen im genannten Sinne, z.B. gemäß dem zweiten Schritt
  - Quadriere den Wert in (der Speicherzelle für) z_1, und schreibe das Ergebnis in (die Speicherzelle für) z_2.

  Dies kann bereits als (recht informeller) imperativer Pseudocode einer ***Zuweisung*** (das ist eine typische „elementare“ Anweisung) aufgefasst werden, in „üblicherer“ Schreibweise z.B.:

  $$z_2 := z_1 * z_1$$
- Eine (in anderem Kontext eventuell relevante) Zustandsänderung
  - Quadriere den Wert in z_1 (und schreibe das Ergebnis wieder in z_1).

  würde beschrieben durch die Zuweisung

  $$z_1 := z_1 * z_1$$

3.5 Typüberprüfung

Typaussagen

Zur ***statischen Typüberprüfung*** (Bestimmung der Typen von Termen und Feststellung eventueller Typisierungsfehler vor der Programmausführung; Alternative in anderen Programmiersprachen: ***dynamische Typüberprüfung*** während der Programmausführung) 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 die ***Typzuweisung*** $\Gamma = \{x_1 : typ_1, \ldots, x_n : typ_n\}$ die Typannahmen bzgl. $x_1, \ldots, x_n$ darstellt (und immer angenommen sei, dass $x_i \neq x_j$ für $i \neq j$ ist).

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

Typinferenz

- Grundlage für die Typinferenz sind ***Typisierungsregeln***, mit deren Hilfe Typaussagen aus ***Typaxiomen*** in schematischer Weise gewonnen werden können.
- Typaxiome sind „elementare" Typaussagen über Konstanten, Namen, Basisfunktionen (...). Beispiele:

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

- 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 $A_1, \ldots, A_k$ 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) $\Gamma \cup \{x : typ_1\} \rhd t : typ_2 \ \vdash\ \Gamma \rhd\ \textbf{fn}\ x \Rightarrow t : typ_1 \to typ_2$

Beachte:

- Die Typzuweisungen in den Prämissen der Regeln müssen jeweils die Eigenschaft haben, dass die daraus gebildeten Typzuweisungen in den Konklusionen gemäß o.a. allgemeiner Festlegung „widerspruchsfrei“ sind.
- Die Regeln sind eigentlich ***Regelschemata***, die jeweils für unendlich viele konkrete Regeln mit konkreten $\Gamma_1, \Gamma_2, \ldots, t_1, t_2, \ldots, typ_1, typ_2, \ldots$ stehen.

Beispiel einer Herleitung einer Typaussage

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)

Prinzipale Typisierung und Instanzierung

- Viele Terme können verschiedene Typen haben, z.B.:

$\{x : \textbf{int}, f : \textbf{int} \to \textbf{int}\} \ \rhd\ f(x) : \textbf{int}$,
$\{x : \textbf{int} * \textbf{int}, f : \textbf{int} * \textbf{int} \to \textbf{bool}\} \ \rhd\ f(x) : \textbf{bool}$,
$\{x : \boldsymbol{\alpha}, f : \boldsymbol{\alpha} \to \boldsymbol{\alpha}\} \ \rhd\ f(x) : \boldsymbol{\alpha}$,
$\{x : \boldsymbol{\alpha}, f : \boldsymbol{\alpha} \to \boldsymbol{\beta}\} \ \rhd\ f(x) : \boldsymbol{\beta}$.

Unter den möglichen Typbestimmungen gibt es eine allgemeinste (***prinzipale Typisierung***; im Beispiel: die letztgenannte): Jede andere erhält man durch Einsetzen von Typen (***Instanzierung***) für die Typvariablen (im Beispiel: α und β).

- Bei der automatischen Typinferenz bestimmt das SML-System zu jedem Term jeweils die prinzipale Typisierung. Instanzierungen werden durch die Anwendung der ***Instanzierungsregel***

$$\Gamma \rhd t : typ \;\vdash\; \Gamma[\alpha_1/typ_1, \ldots, \alpha_m/typ_m] \rhd t : typ[\alpha_1/typ_1, \ldots, \alpha_m/typ_m]$$

nachgebildet. (Die Angabe $[\alpha_1/typ_1, \ldots, \alpha_m/typ_m]$ bezeichnet die simultane Ersetzung jedes Vorkommens der Typvariablen α_i durch typ_i ($i = 1, \ldots, m$) in Γ bzw. typ.)

Unifikation

- Die Bestimmung der prinzipalen Typisierung für einen Term t erfolgt rekursiv nach dem Termaufbau von t: Sie wird bestimmt aus den prinzipalen Typisierungen für die Teilterme von t, indem man auf diese die allgemeinstmögliche Instanzierung anwendet, so dass sie „zusammenpassen“.
- Allgemeine Aufgabenstellung dabei:

  Bestimme zu zwei Typen typ und typ' die allgemeinstmögliche Ersetzung $\sigma = [\alpha_1/typ_1, \ldots, \alpha_m/typ_m]$, so dass $typ\sigma$ und $typ'\sigma$ identisch sind.

Solch ein σ heißt ***allgemeinster Unifikator*** von typ und typ', seine Bestimmung (falls er existiert) heißt ***Unifikation*** und kann durch einen ***Unifikationsalgorithmus*** durchgeführt werden.

3.6 Ausnahmen

Mögliche Programm-Fehler

In einem Programm können drei grundsätzliche Arten von Fehlern auftreten:

- Syntaxfehler: Das Programm ist nicht gemäß den syntaktischen Regeln geschrieben. Diese Fehler werden statisch (vor Ausführung des Programms) erkannt. Dazu gehören in SML insbesondere Typfehler.
- Laufzeitfehler: Fehler, die während der Ausführung eines syntaktisch korrekten Programms auftreten. Beispiel: Aufruf einer Funktion mit einem Argument außerhalb des Definitionsbereichs der Funktion.

- Semantische Fehler: Das Programm löst nicht die gestellte Aufgabe, liefert also nicht für jede mögliche Eingabe die gemäß der Aufgabenstellung gewünschten Ergebnisse (Stichworte: Terminierung und Korrektheit).

Ausnahmen

Um Laufzeitfehler kontrolliert zu behandeln und Programme „robust“ zu gestalten, kann in einer SML-Sitzung mit einer ***Ausnahmedeklaration*** der Form

exception a

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

raise a

(der an beliebiger „Term-Stelle“ stehen kann) 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
```

Standardausnahmen

Viele Ausnahmen sind als ***Standardausnahmen*** bereits vordefiniert.
Beispiele (siehe auch Abschnitte 3.2 und 3.4)

```
- 3 div 0;
uncaught exception divide by zero
- val a = 800000000
  val adopp = 2*a;
val a = 800000000 : int
uncaught exception overflow
```

3.7 Mustervergleich

Fallunterscheidung durch Muster

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):

fun $f\ m_1 = t_1$
$\quad | f\ m_2 = t_2$
$\quad\quad \vdots$
$\quad | f\ m_k = t_k$

$m_1, \ldots, m_k$ 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 (***Mustervergleich***); 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 speziell 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***.
- Arten von Rechenstrukturen:
  - Grundlegende, allgemein verwendbare Rechenstrukturen (mit „grundlegender Standardausrüstung“ an Basisfunktionen; typische derartige Strukturen: in den restlichen Abschnitten dieses Kapitels).

– Anwendungsspezifische Rechenstrukturen (siehe Abschnitt 5.4).

(Beachte: In einer konkreten Programmiersprache sind üblicherweise nicht „alle“ grundlegenden Rechenstrukturen direkt verfügbar; sie müssen dann bei Bedarf in geeigneter Weise realisiert werden.)

Beispiele

1. Die (grundlegende) Rechenstruktur von **bool**

   Abstrakt (gemäß Abschnitt 2.2):

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

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

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

   Abstrakt:

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

- 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 (Menge der n-***Tupel über*** $typ_1, \ldots, typ_n$).

- Signatur:

  $typ_1 * \ldots * typ_n$
  $typ_1, \ldots, typ_n$
  $(\ , \ldots, \ ) : typ_1 \rightarrow typ_2 \rightarrow \ldots \rightarrow typ_n \rightarrow typ_1 * typ_2 * \ldots * typ_n$
  $\#1 : typ_1 * \ldots * typ_n \rightarrow typ_1$
  $\vdots$
  $\#n : typ_1 * \ldots * typ_n \rightarrow typ_n$

- 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***, ***Verbunde***; ebenfalls als Gleichheitstypen):

  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
- p = {Alter=20+5,Name="Meier"};
val it = true : bool
```

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. Beispiele:

```
- 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}
- type komplexeZahl = {re:real,im:real}
- type Punkt = {x:real,y:real}
- type Name = {Vorname:string,Initiale:char,Nachname: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 2.5 polymorphe Funktionen, z.B.:

$$\#1 : {'a1} * \ldots * {'an} \rightarrow {'a1}$$

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

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

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

4.3 Varianten

Die Rechenstruktur der Varianten

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

  $typ_1 \mid \ldots \mid typ_n$

gebildet (Menge der ***Varianten aus*** $typ_1, \ldots, typ_n$). $typ_1, \ldots, typ_n$ müssen dabei nicht alle verschieden sein.

- Signatur:

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

- 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 $typ_1 \mid \ldots \mid typ_n$).

Varianten in SML

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

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

Beispiel:

```
- datatype Geldbetrag = EUR of int | USD of int;
datatype Geldbetrag = EUR of int | USD of int
- val x = EUR(10);
val it = EUR 10 : Geldbetrag
- x = EUR(2+8);
val it = true : bool
- x = USD(10);
val it = false : bool
```

Mustervergleich mit Varianten

- 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 3.7):

  $inj_i\ m$,

  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

- Wichtige, in der Praxis sehr häufig vorkommende Datentypen sind die Mengen typ^* aller (endlichen) Folgen über anderen Datentypen typ. Diese können mit verschiedenen Basisfunktionen zur Bearbeitung ausgestattet werden, im Zusammenhang mit den nachfolgend ausgewählten Funktionen erhält man ***Listentypen***, deren Datenelemente ***Listen*** (***Sequenzen***) (über typ) heißen. Sie bilden ***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):
    $$y :: (x_1, \ldots, x_n) = (y, x_1, \ldots, x_n)$$ (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“):
    $$null(x) = true \Leftrightarrow x = nil.$$

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

- 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**.

- 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.

  (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** $\rightarrow$ **int**)

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

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

```
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** $\rightarrow$ **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

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

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

   $f : typ_1 * typ_2 \to typ_2 \quad \mapsto \quad foldr(f) : typ_2 \to typ_1$ **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.: $foldr : ('a * 'b \to 'b) \to 'b \to 'a$ **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

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

   (D.h.: $filter : ('a \to$ **bool**$) \to 'a$ **list** $\to 'a$ **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 Text x in die Liste der Zeichen von x umwandelt, z.B.:

$explode$("abc") $= (\#$"a", $\#$"b", $\#$"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
```

Anwendungsbeispiele

1. Teilaufgabe im Rahmen von ***Syntaxanalyse***-Algorithmen:
   Für eine Zeichenreihe (über dem ASCII-Alphabet) soll festgestellt werden, ob sie ein alphanumerischer Identifikator gemäß der Syntaxdefinition für SML in Abschnitt 3.3 ist.

   Modellierung: Zeichenreihe als Text (**string**).

Algorithmus:

```
fun buchstabe x = (* true genau dann, wenn x ein Buchstabe ist *)
      (#"A" <= x andalso x <= #"Z") orelse
      (#"a" <= x andalso x <= #"z")
fun erlaubtezeichen nil     = true
  | erlaubtezeichen (y::yt) =
          (* true genau dann, wenn die Liste nur aus Buchstaben,
             Ziffern, Hochkommas und Unterstrichen besteht *)
      (buchstabe(y) orelse
      (#"0" <= y andalso y <= #"9") orelse
      y = #"'" orelse
      y = #"_")
      andalso
      erlaubtezeichen(yt)
fun alphanumidalsliste nil     = false
  | alphanumidalsliste (x::xt) = buchstabe(x) andalso
                                 erlaubtezeichen(xt)
  (* Die eigentliche Loesung: *)
fun alphanumid x = alphanumidalsliste(explode(x))
```

Beispielanwendungen:

```
- alphanumid("ab_1");
val it = true : bool
- alphanumid("ab-1");
val it = false : bool
```

2. ***Türme von Hanoi:***
   n (≥ 1) Spielsteine paarweise verschiedener Größe sind mit nach oben abnehmender Größe zu einem Turm aufgeschichtet. Dieser Turm soll von seinem Platz (1) in einer Folge von Spielzügen auf einen anderen Platz (2) verlegt werden, wobei ein weiterer Platz (3) als „Zwischenablage“ zur Verfügung steht. In jedem Spielzug darf jeweils nur der oberste Stein eines Turms von einem der Plätze (1) – (3) entweder auf einen leeren Platz oder auf einen größeren Stein gelegt werden.

   Modellierung: Spielzug „Verlege Stein von Platz (i) nach Platz (j)“
   als Paar (i, j) aus **int** $\times$ **int**,
   Folge von Spielzügen (Ergebnis) als Liste von Spielzügen.

   Algorithmus (mit Einbettung):

```
fun TVHallg(n,j,k) = (* Folge von Spielzuegen zur Verlegung
                        eines Turms mit n Steinen von Platz
                        j nach Platz k *)
    if n=1 then [(j,k)]
    else let
         val i = 6-j-k  (* freier Platz *)
         in
         TVHallg(n-1,j,i) @ [(j,k)] @ TVHallg(n-1,i,k)
         end
```

```
  (* Die eigentliche Loesung: *)
fun TVH(n) = TVHallg(n,1,2)
```

Beispielanwendung:

```
- TVH(3);
val it = [(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)]
         : (int * int) list
```

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“: Siehe Abschnitte 4.6 und 5.4.

4.5 Reihungen

Die Rechenstruktur der Reihungen

- Die Ausstattung endlicher Folgen von Elementen eines Typs typ mit den nachfolgenden Basisfunktionen ergibt ***Reihungstypen***. Die Elemente heißen ***Reihungen*** (***Vektoren***) (über typ); sie bilden ebenfalls homogene Datenstrukturen.
- Bezeichnung von Reihungstypen: typ **vect** (für typ^*).

- Signatur:

  typ **vect**
  typ
  nat
  $dim : typ$ **vect** $\rightarrow$ **nat**
  $init :$ **nat** $* typ \rightarrow typ$ **vect**
  $get : typ$ **vect** $*$ **nat** $\rightarrow typ$
  $update : typ$ **vect** $*$ **nat** $* typ \rightarrow typ$ **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** $\rightarrow$ **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** $*\,''a \rightarrow$ **bool**)

```
fun enthallg(x,k,a) =  (* Vorbedingung: k > 0 *)
     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 Induktiv definierte Datenmengen

Induktive Definition von Mengen

(Unendliche) Mengen können ***induktiv*** („konstruktiv“, durch Beschreibung der „Konstruktion“ ihrer Elemente) definiert werden.

Allgemeines Schema für die induktive Definition einer Menge M:

a) Gewisse Objekte werden explizit als Elemente von M angegeben.

b) Weitere Elemente $y \in M$ sind gemäß festgelegter Regeln (mit Hilfe gewisser Konstruktoren) aus schon vorhandenen $x_1, \ldots, x_k \in M$ erzeugbar.

(Außer den Elementen gemäß a) und b) soll M keine weiteren Elemente enthalten.)

Beispiel: Induktive Definition der Menge A^* aller endlichen Folgen über einer Menge A:

a) Die leere Folge ε ist Element von A^*.

b) Ist $b \in A$ und $a \in A^*$, so ist $b :: a \in A^*$.
(:: „Vornanhängen“ von b an a, wie in Abschnitt 4.4.)

(Dies entspricht der Erzeugbarkeit von Listen durch nil (d.h. ε) und ::.)

Induktive Datenstrukturen

- Neben Listen sind allgemein auch andere Datentypen, die induktiv definierbar sind und damit eine ***induktive Datenstruktur*** tragen, von Bedeutung.
- Die Signatur einer Rechenstruktur von einem derartigen Typ enthält (typischerweise) mindestens:
  - den damit definierten Typ sowie gegebenenfalls die Typen der in der induktiven Definition noch verwendeten Elemente (Beispiel Listen: typ **list**, typ).
  - die explizit angegebenen Elemente des Typs als Konstanten (Beispiel Listen: nil).
  - Die Konstruktoren zur Erzeugung von Elementen des Typs als weitere Funktionen (Beispiel Listen: ::).

  Häufige zusätzliche typische Bestandteile der Signatur:

  - Typ **bool** und Funktionen zum Test, ob ein Element ein explizit angegebenes Element ist (Beispiel Listen: $null$).
  - Funktionen, die an den induktiven Aufbau der Elemente „angelehnt“ sind, insbesondere: Selektoren für die Daten, aus denen Elemente erzeugt werden können (Beispiel Listen: hd, tl).

  (Zusätzlich können noch weitere Funktionen enthalten sein; Beispiel Listen: @.)

Induktive Datenstrukturen in SML

- In SML sind außer Listentypen keine Datentypen für induktive Datenstrukturen direkt verfügbar.
- Zur Realisierung stehen z.T. verschiedene Möglichkeiten zur Verfügung, insbesondere kann das oben allgemein beschriebene Grundschema der induktiven Mengendefinition direkt nachgebildet werden (siehe Abschnitt 5.4).

Beispiel

Stapel und Schlangen (vgl. Abschnitt 4.4) können nach diesem Schema eigenständig definiert werden, z.B. Stapel mit folgender Signatur:

typ **stack**
typ
bool

$emptyst : typ$ **stack**	(leerer Stapel)
$push : typ * typ$ **stack** $\rightarrow typ$ **stack**	
$top : typ$ **stack** $\rightarrow typ$	
$pop : typ$ **stack** $\rightarrow typ$ **stack**	
$isemptyst : typ$ **stack** $\rightarrow$ **bool**	(Test auf „Leersein“)

Bemerkung

Auch der Datentyp der natürlichen Zahlen könnte („theoretisch“) auf diese Weise (statt als Basistyp) eingeführt werden, z.B. (in „minimaler“ Form) mit folgender Signatur:

indnat

$zero$: **indnat**	(0)
$succ$: **indnat** $\rightarrow$ **indnat**	(***Nachfolgerfunktion*** $x \mapsto x + 1$)

(Beachte: Die Elemente von **indnat** sind $zero, succ(zero), succ(succ(zero)), \ldots$)

Induktive Datenstrukturen und Rekursion

- Wie Listen eignen sich auch andere Datentypen mit induktiver Struktur besonders gut zur Bearbeitung mit rekursiven Funktionen.
- Allgemeines Schema der Definition einer rekursiven Funktion f mit Argumenten aus einem derartigen Datentyp typ (vorausgesetzt sei: die induktive Erzeugung der Elemente des Typs ist eindeutig):
  - $f(x)$ wird für die Konstanten x von typ explizit angegeben.
  - Für jeden Konstruktor, mit dem ein Element y aus Elementen $x_1, \ldots, x_n$ erzeugt wird, wird $f(y)$ unter Rückgriff auf $f(x_1), \ldots, f(x_n)$ definiert.

Solche Definitionen können mit dem SML-Konzept des Mustervergleichs (mit weiteren Arten von Mustern) unter alleiniger Verwendung der Konstanten und der genannten Konstruktoren formuliert werden (bei Listen mit nil und ::). Die Formulierung mit Fallunterscheidungen durch bedingte Terme erfordert üblicherweise auch weitere Basisfunktionen (bei Listen: $null$, hd und tl).

- Beispiel: Bestimmung der Anzahl der Elemente in einem Stapel ($'a$ **stack** $\rightarrow$ **int**). (Vgl. $length$ in Abschnitt 4.4.)

  (a) Mit Mustervergleich:

```
fun lengthstack emptyst     = 0
  | lengthstack (push(_,x)) = 1+lengthstack(x)
```

  (b) Mit bedingtem Term:

```
fun lengthstack(x) = if isemptyst(x) then 0
                     else 1+lengthstack(pop(x))
```

Bemerkungen

- Die in Abschnitt 3.3 behandelte Syntaxdefinition mit BNF-Grammatiken lässt sich als (i. Allg. recht komplexe) induktive Definition (der Menge der gewünschten Zeichenreihen) auffassen. Das erhellt auch noch einmal, warum die Auswertungsfunktion W in Abschnitt 3.4 passenderweise rekursiv definiert ist.
- Der in Abschnitt 3.4 bereits erwähnte Begriff der ***strukturellen Induktion*** lässt sich entsprechend auf beliebige induktive Datenstrukturen übertragen: Eine adäquate fundierte Relation auf induktiv definierten Datenmengen für Induktionsbeweise (und insbesondere für Abstiegsfunktionen für Terminierungsbeweise) ist gegeben durch:

  $x \prec y \Leftrightarrow y$ ist erzeugt „unter Verwendung“ von x.

  Bei Listen heißt das insbesondere:

  $x \prec y \Leftrightarrow x$ ist „Teilliste“ von y (speziell: $x = tl(y)$).

4.7 Binärbäume

Induktive Definition von Binärbäumen

Neben listenartigen Datenstrukturen spielen in vielen Anwendungen ***baumartige*** induktive Datenstrukturen eine wichtige Rolle.

Es sei A eine Menge. Die Menge $A^{\triangle}$ der ***Binärbäume*** über A (als eine „Standardform“ baumartiger Datentypen) ist induktiv definiert wie folgt:

a) $A^\triangle$ enthält den ***leeren Binärbaum*** τ.

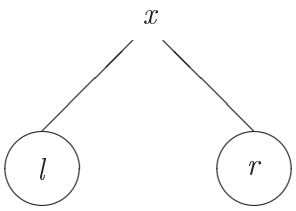
b) Sind $x \in A$ und $xl, xr \in A^\triangle$, so ist das 3-Tupel (x, xl, xr) ein Element von $A^\triangle$.

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 rekursiv 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:



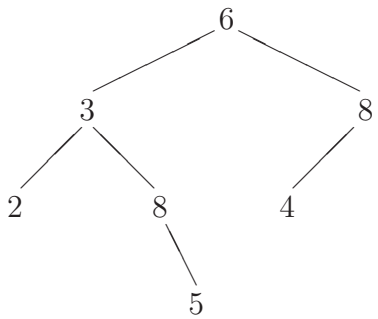
wobei an den Stellen l und r die entsprechenden Darstellungen von xl und xr angesetzt sind. „Zweige“, die auf τ führen, werden weggelassen.

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

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

  typ **bintree**
  typ
  bool
  $emptybt : typ$ **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**
  $isemptybt : typ$ **bintree** $\rightarrow$ **bool**

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

Einige Grundalgorithmen für Binärbäume

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

```
fun knotanz z =
      if isemptybt 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   isemptybt(z1) orelse isemptybt(z2)
      then isemptybt(z1) andalso isemptybt(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 isemptybt 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 isemptybt 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 isemptybt z then nil
      else linsym(left(z)) @ [root(z)] @ linsym(right(z))
```

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

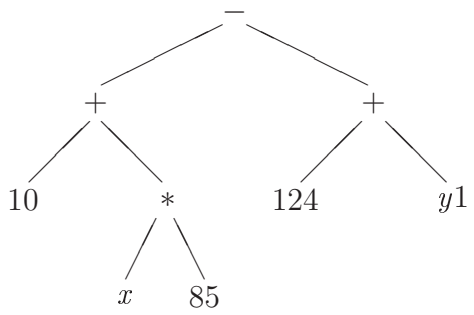
```
fun linnach z =
      if isemptybt z then nil
      else linnach(left(z)) @ linnach(right(z)) @ [root(z)]
```

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 isemptybt b then nil
      else pfumw(left(b)) @ pfumw(right(b)) @ [root(b)]
```

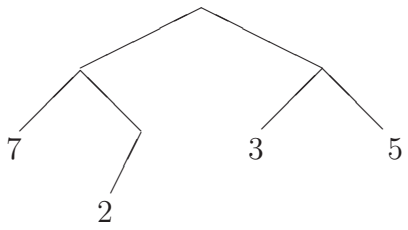
(Typ von $pfumw$: **string bintree** $\rightarrow$ **string list**.)

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***.

  Beispiel:



Kapitel 5
Methodisches Programmieren

5.1 Modularisierung

Beispiel einer komplexen Anwendung

In einer Firma sollen durch ein Programm Quittungen erstellt werden:

Fma. L. Kaufgut *10.01.2006*
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

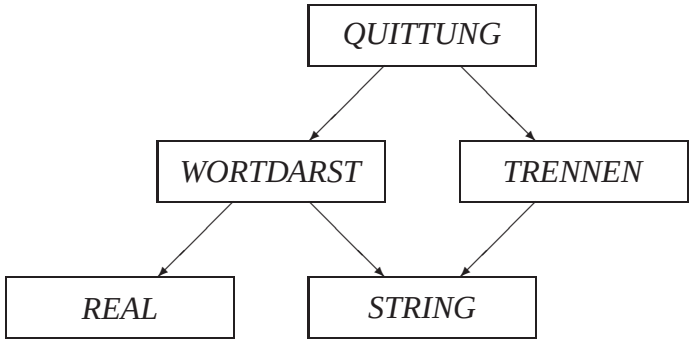
- Ein ***Modul*** ist eine Menge von Datentypen zusammen mit einer Menge von Funktionen auf diesen Typen und eventuell noch weiteren Bestandteilen (die im Folgenden noch besprochen werden).
- Insbesondere ist jede Rechenstruktur ein Modul.
- Im Allgemeinen enthält ein Modul „interne“ Bestandteile, die „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 sind ***verborgen***). 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.
- Erweiterung 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* (mit der Schnittstelle $konvert$) 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
```

  (Beachte die Verwendung von **val** in $WORTDARSTSig$.) Der Modul enthält in seiner Strukturdeklaration außer $konvert$ eventuell noch weitere (Hilfs-) Funktionen. In

  structure $WORTDARST : WORTDARSTSig$

  werden die Bestandteile der Signatur 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.

Beispiel: Modul $Math$ mit vielen **real**-Funktionen ($sqrt, sin, cos, ln, \ldots$).

Anwendung z.B.:

```
- Math.sqrt 1.44;
val it = 1.2 : real
```

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: 1 <= 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

- In Funktionsrümpfen können in einem let-Term auch lokale (***untergeordnete***) Funktionen deklariert werden (vgl. Abschnitt 2.1):

  ...
  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***).

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

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. Stapel (vgl. Abschnitte 4.4 und 4.6):

```
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
```

Rekursive datatype-Deklarationen

- Stapel (in obigem Beispiel als „eingeschränkte Liste“ realisiert) können auch „eigenständig“ als Datentyp mit einer induktiven Datenstruktur gemäß Abschnitt 4.6 angesehen werden.
- Das in Abschnitt 4.6 beschriebene Grundschema für induktive Datenstrukturen kann allgemein ein SML „direkt“ durch (rekursive!) datatype-Deklarationen nachgebildet und realisiert werden:

  Ein solcher Datentyp ist ein Variantentyp (gemäß Abschnitt 4.3) mit den explizit angegebenen Konstanten und den „erzeugenden“ Konstruktoren (als Injektionen), im Beispiel der Stapel:

  datatype $'a$ **stack** $= emptyst \mid push$ **of** $'a * 'a$ **stack**.
- In den (Schnittstellen-Realisierungen von) entsprechenden Moduln können derartige Deklarationen verwendet werden. Die weiteren Funktionen der Signatur werden mit Hilfe der Bestandteile des Variantentyps (rekursiv) definiert.

Weitere Beispiele

1. Stapel (jetzt „eigenständig“ im Sinne von Abschnitt 4.6)

```
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
   datatype 'a stack = emptyst | push of 'a * 'a stack
   exception emptystack
   fun top emptyst     = raise emptystack
     | top (push(x,_)) = x
   fun pop emptyst      = raise emptystack
     | pop (push(_,xt)) = xt
   fun isemptyst emptyst     = true
     | isemptyst (push(_,_)) = false
end
```

2. Binärbäume (vgl. Abschnitt 4.7):

```
signature BINTREESig =
sig
   type 'a bintree
   exception emptytree
   val emptybt   : '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 isemptybt : 'a bintree -> bool
end

structure BINTREE : BINTREESig =
struct
   datatype 'a bintree = emptybt |
                         build of 'a * 'a bintree * 'a bintree
   exception emptytree
   fun root emptybt           = raise emptytree
     | root (build(x,_,_))    = x
   fun left emptybt           = raise emptytree
     | left (build(_,xl,_))   = xl
```

```
    fun right emptybt          = raise emptytree
      | right (build(_,_,xr))  = xr
    fun isemptybt emptybt        = true
      | isemptybt (build(_,_,_)) = false
end
```

Anwendungsspezifische Datentypen

Moduln wie *VECT*, *STACK*, *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:

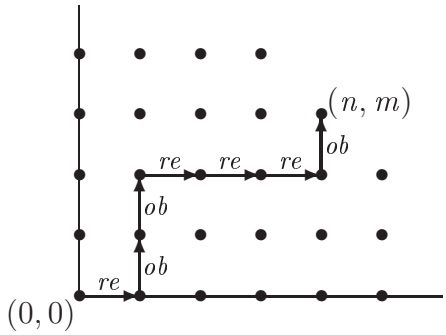
```
signature STACKSig =
sig
   type 'a stack
   exception emptystack
   val emptyst   : 'a stack
   val push      : 'a * 'a stack -> 'a stack
   val top       : 'a stack -> 'a
   val pop       : 'a stack -> 'a stack
   val isemptyst : 'a stack -> bool
(* Axiome:
   isemptyst(emptyst) = true,
   isemptyst(push(x,s)) = 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), $n, m \in \mathbb{N}$ 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:

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*

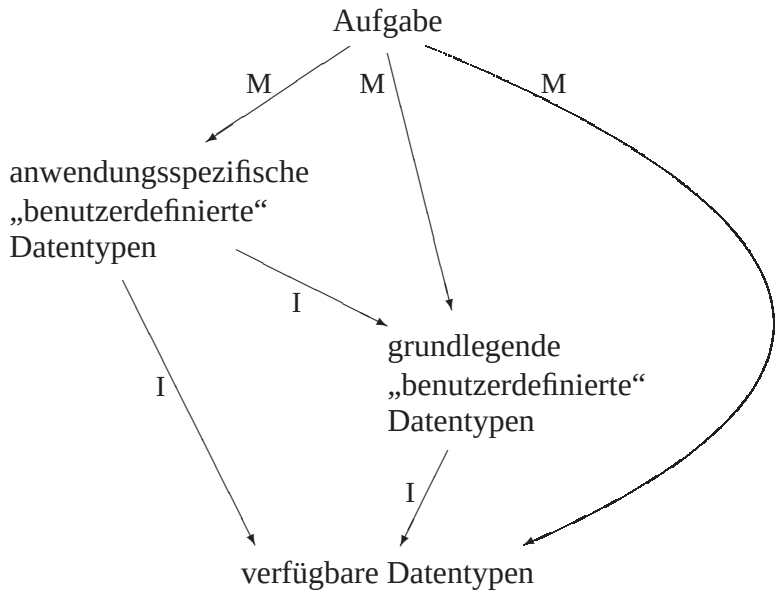
Modellierung der Gesamtheit aller Wege: Als $schritt$ **list set**.

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 typische 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).
- 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 folgende Funktion summiert alle Komponenten einer Liste ganzer Zahlen:

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

Beispiel-Auswertung (gemäß Substitutionsmodell, Schreibweise wie in Kapitel 2):

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

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) $sum0(0,8,2,4) = 0$

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

c) $sum0(3,8,2,4) = \ldots$ (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

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

  $$\left.\begin{aligned} fak(4) &= 4 * fak(3) \\ &= 4 * (3 * fak(2)) \\ &= 4 * (3 * (2 * fak(1))) \\ &= 4 * (3 * (2 * (1 * fak(0)))) \\ &= 4 * (3 * (2 * (1 * 1))) \end{aligned}\right\} \text{Aufruffolge}$$

  $$\left.\begin{aligned} &= 4 * (3 * (2 * 1)) \\ &= 4 * (3 * 2) \\ &= 4 * 6 \\ &= 24 \end{aligned}\right\} \text{Tatsächliche Berechnungen}$$

- Die schrittweisen Berechnungen nach dem letzten Aufruf sind dadurch hervorgerufen, dass im Rumpf der Funktion der rekursive Aufruf noch mit anderen Größen „weiterverarbeitet“ wird (im Beispiel: Multiplikation des 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 $k \in \mathbb{N}$).

  Beispiel-Auswertung:

  $$\left.\begin{aligned} potenz(3,54) &= potenz(3,18) \\ &= potenz(3,6) \\ &= potenz(3,2) \\ &= false \end{aligned}\right\} \text{nur Aufruffolge}$$

  (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.
Oft anwendbares Schema dafür:

- Es wird eine allgemeinere Funktion definiert, 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.

Beispiel

Für die Funktion fak wird die allgemeinere Funktion

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

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

Berechnung von $fakrep$:

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

$fakrep$ ist repetitiv rekursiv. Beispiel-Auswertung:

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

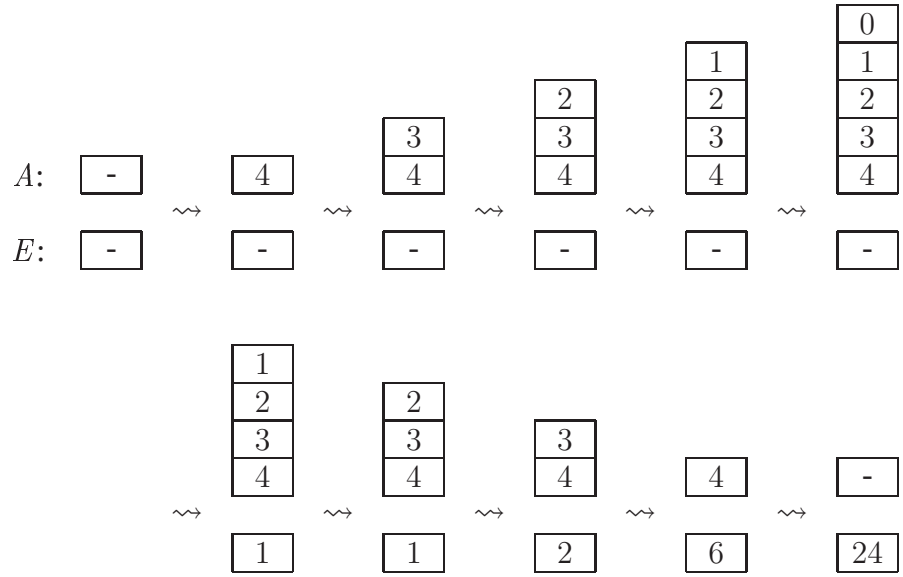
Effizienzvergleich im Beispiel

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 $fakrep$ benötigt allerdings weniger Speicherplatz (und ist auf vielen Rechenanlagen auch effizienter zu realisieren).

Illustration der Speicherplatz-Ersparnis:
Die Auswertung eines Aufrufs einer rekursiven Funktion (hier: $fak(n)$) erfolgt im Rechner nach folgendem Grundschema:

- Die jeweiligen aktuellen Parameter des Aufrufs und der weiteren induzierten Aufrufe (hier: $n, n-1, n-2, \ldots, 0$) werden nacheinander (als „noch unerledigt“) im Speicher abgelegt.
- Anschließend werden diese Eintragungen in umgekehrter Reihenfolge zum „Aufbau“ des Ergebnisses verwendet (hier: mit dem jeweiligen Zwischenergebnis multipliziert).

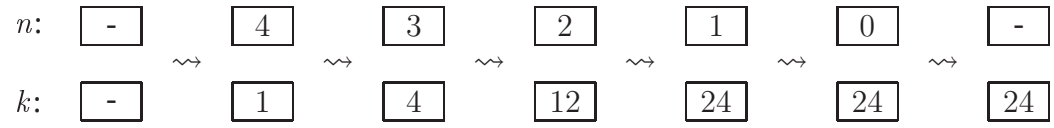
Als Bild für $fak(4)$ (A, E: Speicherstellen für Argumente bzw. Ergebnis):



Beachte: Die (Werte in den) Speicherzellen für die Ablage der Argumente bilden zusammengenommen einen Stapel (Abschnitt 4.4).

Für die Auswertung einer repetitiv rekursiven Funktion genügen Speicherzellen, auf denen die jeweiligen Werte der Parameter (hier: n und k) „protokolliert“ werden.

Als Bild für $fakrep(4)$:



Rekursion und Iteration

- Das Berechnungsschema, das der Auswertung eines Aufrufs $f(a_1, \ldots, a_n)$ einer repetitiv rekursiven Funktion

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

  zugrunde liegt, kann allgemein durch folgende Schritte beschrieben werden:

(1) Schreibe die Eingaben $a_1, \ldots, a_n$ in (die Speicherzellen für) $x_1, \ldots, x_n$.

(2) Solange die „Terminierungsbedingungen“ der Rekursion für die Inhalte der $x_1, \ldots, x_n$ nicht zutreffen, wiederhole folgenden Schritt:

Ersetze die Inhalte von $x_1, \ldots, x_n$ durch die durch den jeweiligen rekursiven Aufruf gegebenen neuen Parameterwerte.

(3) Gib als Ergebnis den aus den Inhalten von $x_1, \ldots, x_n$ gemäß den „Terminierungsfällen“ bestimmten Wert zurück.

- Dieses Schema beschreibt einen imperativen Algorithmus (vgl. Abschnitt 3.4).

  Schreibweise in „typischem“ Pseudocode (angelehnt an übliche imperative Programmiersprachen):

  $x_1 := a_1;$
  $\vdots$
  $x_n := a_n;$
  while Terminierungsbedingungen der Rekursion für $x_1, \ldots, x_n$ nicht erfüllt
  do Besetze $x_1, \ldots, x_n$ mit den neuen Parameterwerten gemäß Rekursion
  end;
  return Ergebnis gemäß Terminierungsfällen der Rekursion

- Charakteristisch für diese imperative Auswertung ist die ***Iteration*** (Wiederholung) der Anweisung „Besetze $x_1, \ldots, x_n$ mit den neuen Parameterwerten“ durch die ***Wiederholungsanweisung*** **while** ... **do** ... **end**.

  Das imperative Konzept der Iteration spiegelt genau das funktionale Konzept der repetitiven Rekursion wider.

  Die Auswertung einer (beliebig) nicht-repetitiven Funktion kann – wie im Beispiel fak angedeutet – durch Iteration unter Verwendung eines stapelartigen „Hilfsspeichers“ (***Keller***) ausgedrückt werden.

Ein weiteres Beispiel

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

$$y +\!+\ m = (y_1 + m, y_2 + m, \ldots, y_n + m).$$

($+\!+$ 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:

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

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 *frep* definiert durch:

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

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

Berechnung von *frep*:

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

frep ist repetitiv rekursiv. Beispiel-Auswertung:

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

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

Aber: Weitergehende Einbettung möglich zu

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

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

Berechnung von frep':

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

frep' ist wieder repetitiv rekursiv. Beispiel-Auswertung:

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

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.) frep' hat somit sogar eine niedrigere asymptotische Zeitkomplexität als f (neben den anderen Effizienzeffekten wie bei frep).

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:

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

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

Einbettung:

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

Dann gilt: $\mathit{fib}(n) = \mathit{fibrep}(n,0,1)$.

Berechnung von fibrep (repetitiv rekursiv):

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

Beispiel-Auswertung:

$$\begin{aligned}
\mathit{fibrep}(4,0,1) &= \mathit{fibrep}(3,0+1,0)\\
&= \mathit{fibrep}(3,1,0)\\
&= \mathit{fibrep}(2,1+0,1)\\
&= \mathit{fibrep}(2,1,1)\\
&= \mathit{fibrep}(1,1+1,1)\\
&= \mathit{fibrep}(1,2,1)\\
&= \mathit{fibrep}(0,2+1,2)\\
&= \mathit{fibrep}(0,3,2)\\
&= 3+2\\
&= 5
\end{aligned}$$

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

6.3 Effiziente Datenstrukturen

Das Suchproblem

Die effiziente Verarbeitungsmöglichkeit von Daten kann ganz erheblich von deren Strukturierung abhängen.

Im Folgenden: Illustration dieses Sachverhalts anhand folgender Standardaufgabe (Suchproblem, vgl. Abschnitte 4.4, 4.5, 4.7), die in der Praxis in vielen Ausprägungen vorkommt.

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:int 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 4.7).

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: $O(log(n))$.

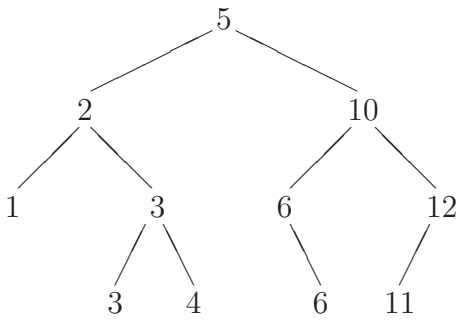
Veranschaulichung der Effizienzverbesserung:
Bei jedem „Suchschritt“ wird die Anzahl der noch zu durchsuchenden Daten halbiert. Beim linearen Suchen wird diese Anzahl nur immer um 1 verringert.

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 von $inssort$:

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

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 von $mergesort$ (schreibe m für $merge$ und s für $mergesort$):

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

Allgemein: Bei Auswertung von $mergesort(x)$ für eine Liste der Länge n ist die Anzahl der Schritte zu der Auswertungsstelle (1) und zwischen $(1),(2),\ldots,(7)$ 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

Zur Erinnerung (Abschnitt 3.3): Die Semantik von Programmen beschreibt deren Bedeutung. Sie kann auf verschiedene Arten angegeben werden.

Die Auswertungsfunktion W (Abschnitt 3.4) definiert eine ***denotationelle*** (***funktionale***) Semantik von SML-Programmen: Einer SML-Funktion wird als Bedeutung

- eine („mathematische“) Funktion (als ***semantisches Modell***)

zugeordnet. Vorteile dieser Semantik: Implementierungsdetails werden „versteckt“, mathematische Beweismethoden (vgl. Abschnitt 2.4) werden verfügbar gemacht.

Für gewisse Zwecke (z.B.: Festlegung von Implementierungsdetails, Effizienzbetrachtungen) ist die denotationelle Semantik zu abstrakt (zu „grob“). Die hierfür geeignetere „feinere“ ***operationelle*** Semantik beschreibt als Bedeutung eines Programms

- den „Ablauf“ von elementaren Schritten bei der Programmausführung.

Eine weitere sehr grobe Semantik-Art ist die ***axiomatische*** Semantik, die speziell für imperative Programme geeignet ist. Die Bedeutung der einzelnen Sprachkonstrukte wird dabei (ähnlich wie in Abschnitt 5.4) beschrieben durch

- charakteristische Eigenschaften.

Denotationelle Semantik von Funktionen

- Die durch die Auswertungsfunktion W (Abschnitt 3.4) beschriebene denotationelle Semantik von Funktionen weist noch einige Ungenauigkeiten auf, insbesondere:
  - Wie sehen die als Bedeutung angegebenen mathematischen Funktionen tatsächlich aus, gibt es sie überhaupt?
- Ein funktionales Programm ist eine Funktion, die syntaktisch durch eine Gleichung definiert ist (in SML: **val** (**rec**) $f = \ldots$). Ihre denotationelle Bedeutung ist (einheitlich) durch die Lösung der Gleichung gegeben. (Wir betrachten der Einfachheit halber keine verschränkt rekursiven Funktionen.) Beispiel (mathematische Notation):

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

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

- Hauptproblem: Rekursive Funktionen, definiert durch Gleichungen der Art

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

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

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

  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:

$g_\omega : \mathbb{N} \to \mathbb{N}^\omega$,

$$g_\omega(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \omega & \text{sonst.} \end{cases}$$

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

Fixpunkt-Funktionale

- Für jede Lösung $g : \mathbb{N} \to \mathbb{N}^\omega$ 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***

  $G : (\mathbb{N} \to \mathbb{N}^\omega) \to (\mathbb{N} \to \mathbb{N}^\omega)$,

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

- 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

Partielle Ordnungen

Definition. Eine binäre Relation $\sqsubseteq$ auf einer Menge 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.

Monotonie

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 $\{f^i(\bot) \mid i \in \mathbb{N}\}$ eine Kette in A.

Vollständigkeit

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 $x = sup\ B$.

Stetigkeit

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.

Kleinste Fixpunkte

Definition. Seien A eine Menge mit partieller Ordnung $\sqsubseteq$ und $f : A \to A$ eine 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 kleinsten Fixpunkt.)

Satz 4. (Fixpunktsatz von Kleene)
Seien A eine cpo mit kleinstem Element, $f : A \to A$ eine stetige Funktion. Das Element

$$x = sup\, \{f^i(\bot) \mid i \in \mathbb{N}\}$$

existiert in A und ist kleinster Fixpunkt von f. (Bezeichnung: $fix(f)$.)

7.3 Denotationelle Semantik von SML-Funktionen

Erweiterung von Datentypen

Zur Anwendung der Ergebnisse von Abschnitt 7.2 zur formalen Semantikdefinition für funktionale (SML-) Programme sei für jeden Datentyp typ 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^U aus Abschnitt 3.4 (bezüglich einer gegebenen Umgebung U).

- 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

Die Semantik $[\![h]\!]$ einer durch

fun $h\ x = t$

deklarierten Funktion h vom Typ $typ_1 \to typ_2$ 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***

  $$f \sqsubseteq g \Leftrightarrow f(a) \sqsubseteq_{fl} g(a) \qquad \text{für alle } a \in typ_1$$

bildet eine cpo mit dem kleinsten Element

$$\Omega : typ_1 \to typ_2^\omega,$$
$$\Omega(a) = \omega \quad \text{für alle } a \in typ_1.$$

- Das Fixpunkt-Funktional

$$F : (typ_1 \to typ_2^\omega) \to (typ_1 \to typ_2^\omega),$$
$$F(f)(a) = [\![t]\!], \quad \text{wobei } [\![t]\!] \text{ bestimmt wird bezüglich der Umgebung, die sich aus der aktuellen Umgebung durch Hinzunahme der Bindungen <}h,f\text{> und <}x,a\text{> ergibt.}$$

ist stetig. Dann ist

$$[\![h]\!] = \mathit{fix}(F),$$

d.h.: $[\![h]\!] = sup\,\{F^i(\Omega) \mid i \in \mathbb{N}\}$.

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

1. `fun h(n) = n+3` (vgl. Abschnitt 7.1).

   Mit $F(f)(n) = n + 3$ erhält man:

$$F^0(\Omega)(n) = \Omega(n) = \omega,$$
$$F^1(\Omega)(n) = F(\Omega)(n) = n + 3,$$
$$F^2(\Omega)(n) = F(F(\Omega))(n) = n + 3,$$
$$\vdots$$
$$F^i(\Omega)(n) = n + 3 \quad \text{für alle } i \geq 1.$$

   Damit: $[\![h]\!](n) = n + 3$.

2. `fun g(n) = if n=0 then 1 else g(n+1)` (für $n \geq 0$, vgl. Abschnitt 7.1)

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

   erhält man:

$$F^0(\Omega)(n) = \Omega(n) = \omega,$$
$$F^1(\Omega)(n) = F(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \Omega(n+1) = \omega & \text{sonst,} \end{cases}$$
$$F^2(\Omega)(n) = F(F(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ F(\Omega)(n+1) = \omega & \text{sonst,} \end{cases}$$

$$\vdots$$

$$F^i(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \omega & \text{sonst} \end{cases} \qquad \text{für alle } i \geq 1.$$

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:

$$F^0(\Omega)(n) = \Omega(n) = \omega,$$

$$F^1(\Omega)(n) = F(\Omega)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * \Omega(n-1) = \omega & \text{sonst,} \end{cases}$$

$$F^2(\Omega)(n) = F(F(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ \omega & \text{sonst,} \end{cases}$$

$$F^3(\Omega)(n) = F(F^2(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F^2(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ 2, & \text{falls } n = 2 \\ \omega & \text{sonst,} \end{cases}$$

$$F^4(\Omega)(n) = F(F^3(\Omega))(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n * F^3(\Omega)(n-1) & \text{sonst} \end{cases}$$

$$= \begin{cases} 1, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ 2, & \text{falls } n = 2 \\ 6, & \text{falls } n = 3 \\ \omega & \text{sonst} \end{cases}$$

   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:

$$[\![fak]\!](n) = n!$$