Informatik I

Folien zum Skriptum

Informatik I – Einführung in Algorithmen
und in die Programmierung

von François Bry

2001, 2002, 2004

1. Einleitung

Ziel:

Einführung zentraler Begriffe wie
„Spezifikation“, „Algorithmus“, „Funktion“, „Rekursion“, „Programm“, usw.

Inhalt:

1.1. Spezifikation, Algorithmus und Programm — Begriffserläuterung am Beispiel der Multiplikation

Einer der bekanntesten Algorithmen ist das Verfahren zur Multiplikation zweier natürlicher Dezimalzahlen.

```
4 2 3   x    2  4  3  1
            -----------
            (1)
             7  2  9  3

          4  8  6  2  0
      (1)(1)
       9  7  2  4  0  0
    -----------------
   (1)(1)(1)(1)(1)
    1  0  2  8  3  1  3
```

Wie kann dieses Verfahren im allgemeinen Fall beschrieben werden?

1.1.1. Informelle Spezifikation des Multiplikationsalgorithmus

Seien Multiplikator und Multiplikand die natürlichen Zahlen, die multipliziert werden sollen:

```
Multiplikator x  Multiplikand
              ---------------
                      Produkt
```

Gegeben: Multiplikationstabelle für die Multiplikation einstelliger Zahlen („das kleine 1x1“)

[einstellig = besteht nur aus einer Ziffer]

Multiplikationsalgorithmus (informell)

Fall 1: `Multiplikator` ist eine Ziffer

Fall 1.1: `Multiplikand` ist eine Ziffer

Ermittle aus der Multiplikationstabelle:

`Ergebnis` = `Multiplikator x Multiplikand`

Liefere `Ergebnis`

Fall 1.2: `Multiplikand` ist keine Ziffer.

Es gilt also: `Multiplikand` = `Z Rest`,
mit `Z` Ziffer und `Rest` natürliche Zahl

Annahme: `Produkt1` = `Multiplikator x Rest` ist bekannt.

Dann ist:

`Produkt2` = `Multiplikator x Z` (aus der Multiplikationstabelle)

Sei n die Länge von `Rest` (d.h. die Anzahl der Ziffern):

$$\texttt{GeschobenProdukt2} = \texttt{Produkt2}\ \underbrace{\texttt{0 ... 0}}_{n \text{ Nullen}}$$

Dann gilt: `Produkt = Produkt1 + GeschobenProdukt2`

Multiplikationsalgorithmus (informell) (II)

Anmerkung:
Annahme in Fall 1.2 unproblematisch, da `Produkt2` mit dem selben Verfahren berechenbar

Fall 1.2 lässt sich damit auch wie folgt zusammenfassen:

`Multiplikand` ist keine Ziffer.

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass: `Multiplikand` $=$ `Z Rest`

Berechne (gleicher Algorithmus, Fall 1): `Produkt1 = Multiplikator x Rest`

Ermittle aus der Multiplikationstabelle: `Produkt2 = Multiplikator x Z`

Berechne
`GeschobenProdukt2 = Produkt2` *gefolgt von* n *Nullen* [$n =$ Länge von `Rest`]

`Ergebnis = Produkt1 + GeschobenProdukt2`

Multiplikationsalgorithmus (informell) (III)

Fall 2: `Multiplikator` ist keine Ziffer (enthält also mindestens 2 Ziffern).

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:

```
Multiplikator = Z Rest
```

Berechne (mit dem Algorithmus Fall 1):

```
Produkt1 = Z x Multiplikand
```

Berechne (mit demselben Algorithmus Fall 1 oder 2, je nach Länge von `Rest`):

```
Produkt2 = Rest x Multiplikand
```

Berechne

`GeschobenProdukt1 = Produkt1` *gefolgt von* n *Nullen*,
wobei n Länge von Rest

Berechne

```
Ergebnis = GeschobenProdukt1 + Produkt2
```

Liefere `Ergebnis`

Multiplikationsalgorithmus: Diskussion

Bezeichner `Z`, `Rest`, `Produkt1`, `Produkt2` und `Ergebnis`

- in beiden Fälle verwendet
- aber mit unterschiedlicher Bedeutung

Ist das zulässig?

$\rightarrow$ ja, da Geltungsbereiche dieser Bezeichner (d.h. die Fälle) sich nicht überlappen!

[*Anmerkung:* Vergleichbares gibt es in allen Lebensbereichen $\rightsquigarrow$ Homonyme]

Ist der Wert eines Bezeichners wie `Produkt1` eindeutig, wenn derselbe Algorithmus Bezug auf sich selbst (oder auf Teilalgorithmen von sich) nimmt?

$\rightarrow$ ja, weil Geltungsbereiche dieser Bezeichner (d.h. die Fälle) sich auch bei solchen „Wiederverwendungen" nicht überlappen

1.1.2. Beispiel einer Anwendung des Falles 1

`Multiplikator` = 2, `Multiplikand` = 94

Der Multiplikationsalgorithmus verläuft wie folgt:

```
2 x    9 4
   -------
         8
   + 1 8 0  (180 ist 2 x 9 gefolgt von 1 Null)
   -------
     1 8 8
```

Der Ablauf im Detail ist wie folgt:

Beispiel (II)

Fall 1.2 trifft zu (`Multiplikand = 94` ist keine Ziffer).

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass: `Multiplikand = Z Rest`

Also `Z = 9`, `Rest = 4`

Berechne (Algorithmus Fall 1): `Produkt1 = Multiplikator x Rest`

Also: `Produkt1 = 2 x 4`

> Begin Nebenberechnung von `2 x 4` (mit Algorithmus Fall 1):
>
> `Multiplikator = 2`, `Multiplikand = 4`
>
> Der Fall 1.1 trifft zu (`Multiplikand = 4` ist eine Ziffer)
>
> Multiplikationstabelle liefert: `Ergebnis = Multiplikator x Multiplikand`
> d.h. `Ergebnis = 8`
>
> Liefere `Ergebnis`

Beispiel (III)

Fortsetzung der Hauptberechnung:

`Produkt1 = 8` (= `Ergebnis` der Nebenberechnung)

Aus der Multiplikationstabelle: `Produkt2 = Multiplikator x Z = 2 x 9 = 18`

Berechne

`GeschobenProdukt2 = Produkt2` *gefolgt von* n *Nullen*

wobei n Länge von `Rest` (d.h. $n = 1$)

Also: `GeschobenProdukt2 = 180`

Berechne

`Ergebnis = Produkt1 + GeschobenProdukt2`

Also: `Ergebnis = 8 + 180 = 188`

Liefere `Ergebnis`

1.1.3. Rekursion und Terminierung

- Algorithmen, die Bezug auf sich selbst (oder auf Teile von sich) nehmen, heißen „rekursiv“

  → „Rekursion“ — eine zentrale Technik der Informatik

- rekursive Algorithmen wirken gelegentlich befremdlich bzw. gewöhnungsbedürftig

  aber: Rekursion ist eine natürliche und eigentlich wohlvertraute Beschreibungstechnik für Verfahren

Beispiel: Zeichnen eines Vierecks auf dem Boden:

Zeichne eine Seite wie folgt:
  Gehe 3 Schritte nach vorne und zeichne dabei eine Linie;
  Wende Dich dann um 90 Grad nach rechts.
  Wenn Du nicht am Startpunkt stehst, dann rufe denselben Algorithmus auf
  (d.h. zeichne eine Seite unter Anwendung des oben geschilderten Verfahrens).

Rekursion und Terminierung (II)

Dieses Beispiel eines rekursiven Algorithmus stört uns nicht, weil

1. das Verfahren überschaubar ist,
2. die Terminierungsbedingung leicht erkennbar ist.

Überprüfung der Terminierung ist bei rekursiven Algorithmen oft nicht trivial!

im Beispiel:
Ohne die Bedingung „wenn Du nicht am Startpunkt stehst“

- würde der Viereck-Algorithmus den Zweck erfüllen, ein Viereck zu zeichnen,
- aber nicht terminieren.

Frage: Terminiert der Multiplikationsalgorithmus aus Abschnitt 1.1.1?

Antwort: $\rightsquigarrow$ siehe die folgende „kommentierte“ Fassung des Algorithmus

Terminierung des Multiplikationsalgorithmus

„Beweis“:

Fall 1: `Multiplikator` ist eine Ziffer

Fall 1.1: `Multiplikand` ist eine Ziffer.

`Ergebnis` = `Multiplikator x Multiplikand`

Liefere `Ergebnis`

** Fall 1.1 terminiert offenbar.

Fall 1.2: `Multiplikand` ist keine Ziffer.

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:
`Multiplikand` = `Z Rest`

** Die Ermittlung von Z und Rest terminiert offenbar.
** Rest ist echt kürzer als Multiplikand (*)

Terminierung des Multiplikationsalgorithmus (II)

Berechne (gleicher Algorithmus, Fall 1): `Produkt1 = Multiplikator x Rest`

** Wegen (*) werden die Multiplikanden der rekursiven Anwendungen des
** Falls 1 immer echt kürzer, so dass letztendlich der (terminierende!) Fall 1.1
** eintreten wird. Die obige rekursive Berechnung terminiert also.

Ermittle aus der Multiplikationstabelle: `Produkt2 = Multiplikator x Z`

** Terminiert offenbar.

Berechne:
`GeschobenProdukt2 = Produkt2` *gefolgt von* n *Nullen* [$n =$ Länge von `Rest`]

** Terminiert offenbar.

Berechne: `Ergebnis = Produkt1 + GeschobenProdukt2`

** Unter der Annahme, dass ein terminierender Algorithmus zur Addition
** zweier natürlichen Zahlen zur Verfügung steht, terminiert diese Berechnung

Liefere `Ergebnis`

Terminierung des Multiplikationsalgorithmus (III)

Fall 2: `Multiplikator` ist keine Ziffer (enthält also mindestens 2 Ziffern)

Sei `Z` eine Ziffer und `Rest` eine natürliche Zahl, so dass:

`Multiplikator = Z Rest` (**)

** Terminiert offenbar.

Berechne (mit dem Algorithmus Fall 1): `Produkt1 = Z x Multiplikand`

** Der Fall 1 terminiert (oben festgestellt).

Berechne (mit demselben Algorithmus Fall 1 oder 2, je nach Länge von `Rest`):

`Produkt2 = Rest x Multiplikand`

** Wegen (*) und (**) ist bei jeder rekursiven Anwendung entweder der
** Multiplikan, oder der Multiplikator immer echt kürzer als der Multiplikand
** oder Multiplikator der vorherigen Anwendung, so dass letztendlich der
** (terminierende!) Fall 1.1 eintreten wird. Die obige rekursive Berechnung
** terminiert also.

Terminierung des Multiplikationsalgorithmus (IV)

Berechne

`GeschobenProdukt1 = Produkt1` *gefolgt von* n *Nullen* [n Länge von Rest]

** Terminiert offenbar.

Berechne

`Ergebnis = GeschobenProdukt1 + Produkt2`

** Unter der Annahme, dass ein terminierender Algorithmus zur Addition
** zweier natürlichen Zahlen zur Verfügung steht, terminiert diese Berechnung

Liefere Ergebnis.

1.1.4. Kritik an der (informellen) Algorithmusbeschreibung

1. nicht ganz präzise:

   Geltungsbereiche von Variablen (wie `Z` und `Rest`) nicht explizit festgelegt

   $\leadsto$ unklar, was genau gemeint ist

2. wenig geeignet für einen präzisen Beweis der Terminierung oder der Korrektheit des Verfahrens

$\Rightarrow$ formale Spezifikationen zur Beschreibung solcher Verfahren (Algorithmen)

1.1.5. Zum Begriff „Algorithmus“

„Algorithms have been around since the beginning of time and existed well before a special word had been coined to describe them. Algorithms are simply a set of step by step instructions, to be carried out quite mechanically, so as to achieve some desired result. [...] Algorithms are not confined to mathematics [...]. The Babylonians used them for deciding points of law, Latin teachers used them to get the grammar right, and they have been used in all the cultures for predicting the future, for deciding medical treatment, or for preparing food. Everybody today uses algorithms of some sort or another, often unconsciously, when following a recipe, using a knitting pattern, or operating household gadgets. [...] Today, principally because of the influence of computing, the idea of finiteness has entered into the meaning of algorithm as an essential element, distinguishing it from vaguer notions such as process, method, or technique.“

aus: Jean-Luc Chabert et al. A history of algorithms - From the pebble to the microchip.
Springer Verlag, ISBN 3-540-63369-3, 1999

Zum Begriff „Algorithmus“ (II)

Herkunft des Wortes „Algorithmus“:

- aus dem (arabischen) Namen eines berühmten iranischen Mathematikers (1. Hälfte des 9. Jahrhunderts nach Christus)

  Muhammad ibn Musa al-Khwarzem (Muhammad Sohn des Musa von Khwarzem)

  [*Khwarzem*: Region in Zentralasien südlich des Aral-Meeres]

Der Titel eines Buches (al-Mukhtaar f Hisb al-Jabr wa l-Muqbala) dieses Mathematikers lieferte übrigens das Wort „Algebra“.

Zum Begriff „Algorithmus“ (III)

Definition (Intuitive Definition des Algorithmusbegriffs)

Ein Algorithmus ist ein Verfahren mit einer *präzisen* (d.h. in einer genau festgelegten Sprache formuliert), *endlichen Beschreibung* unter Verwendung *effektiver* (d.h. tatsächlich ausführbarer) *elementarer Verarbeitungsschritte*.

Zu jedem Zeitpunkt der Abarbeitung des Algorithmus benötigt der Algorithmus nur *endlich viele Ressourcen*.

Zum Begriff „Algorithmus“ (IV)

Wichtige Eigenschaften von Algorithmen:

- Ein Algorithmus heißt *terminierend*, wenn er für alle zulässigen Schrittfolgen stets nach endlich vielen Schritten endet.
- Er ist *deterministisch*, wenn in der Auswahl der Verarbeitungsschritte keine Freiheit besteht.
- Er ist *determiniert*, wenn das Resultat eindeutig bestimmt ist.
- Er wird *sequenziell* genannt, wenn die Schritte stets hintereinander ausgeführt werden.
- Er heißt *parallel* (*nebenläufig*), wenn gewisse Verarbeitungsschritte nebeneinander (im Prinzip gleichzeitig) ausgeführt werden.

1.1.6. Formale Spezifikation eines Algorithmus

zunächst Einführung der Begriffe „totale“ und „partielle Funktion“:

Definition (Totale und partielle Funktion):

- Eine totale Funktion, kurz Funktion, $f : A \to B$ (von A nach/in B) ist eine Teilmenge des Kartesischen Produkts $A \times B$, so dass es für jedes $a \in A$ ein eindeutiges $b \in B$ gibt mit $(a, b) \in f$.

  Eindeutig bedeutet: wenn $(a, b_1) \in f$ und $(a, b_2) \in f$, dann $b_1 = b_2$.

- Eine partielle Funktion $f : A \to B$ ist eine Teilmenge des Kartesischen Produkts $A \times B$, so dass es $A' \subset A$ gibt mit:
  - $A' = \{\ a \mid$ es gibt $b \in B : (a, b) \in f\ \}$
  - $\{\ (a, b) \mid (a, b) \in f\ \}$ ist eine totale Funktion von A' in B.

Formale Spezifikation eines Algorithmus (II)

daraus ergeben sich auch die folgenden alternativen Bezeichnungsweisen:

- totale Funktion: linkstotale und rechtseindeutige binäre (2-stellige) Relation.
- partielle Funktion: rechtseindeutige binäre Relation

Missverständnisse durch verschiedene sprachliche Konventionen:

- Mathematik: “Funktion“ meint üblicherweise „totale Funktion“
- Informatik: “Funktion“ meint üblicherweise „partielle Funktion“

Formale Spezifikation eines Algorithmus (III)

Formale Spezifikation des Multiplikationsalgorithmus (aus Abschnitt 1.1.1)

1. Schritt: Festlegung, welche Typen von Daten verwendet werden
(lässt sich aus der informellen Beschreibung erkennen)

`digit`: (Ziffer)

Ein „digit“ ist ein Element aus $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

`integer`: (ganze Zahl; hier nur nichtnegative ganze, also natürliche Zahlen)

Ein „integer“ ist eine beliebige, nicht-leere, endliche Sequenz von „digits“ (führende Nullen sind also *nicht* ausgeschlossen).

`boolean`: (Wahrheitswert)

`true` oder `false`

Formale Spezifikation eines Algorithmus (IV)

2. Schritt: Bestimmung und formale Spezifikation der Funktionen

```
function
(integer,integer) ---> integer
integer_mult(multiplicator, multiplicand) =
   if one_digit_integer(multiplicator)
   then (* Fall 1 *) integer_digit_mult(multiplicator, multiplicand)
   else (* Fall 2 *)
        (let (*  Einfuehrung von Abkuerzungen fuer bessere Lesbarkeit  *)
             product1 = integer_digit_mult(head(multiplicator), multiplicand)
             product2 = integer_mult(tail(multiplicator), multiplicand)
         in
         integer_add(shift(tail(multiplicator), product1), product2)
        )
end
```

Formale Spezifikation eines Algorithmus (V)

```
function
(digit, integer) ---> integer
integer_digit_mult(multiplicator, multiplicand) =
     if one_digit_integer(multiplicand)
     then (* Fall 1.1 *)
          digit_mult(multiplicator, multiplicand)
     else (* Fall 1.2 *)
          (let product1 = integer_digit_mult(multiplicator, tail(multiplicand))
               product2 = digit_mult(multiplicator, head(multiplicand))
           in
           integer_add(product1, shift(tail(multiplicand), product2))
          )
end
```

Formale Spezifikation eines Algorithmus (VI)

dabei verwendete (als vorhanden angenommene) Funktionen:

`one_digit_integer: integer` $\rightarrow$ `boolean`

liefert true, wenn das Argument ein 1-stelliger `integer`-Wert ist, sonst false.

`digit_mult: (digit, digit)` $\rightarrow$ `integer`

realisiert die „Multiplikationstabelle“

`head: integer` $\rightarrow$ `digit`

liefert das am weitesten links stehende `digit` eines `integer`-Werts

`tail: integer` $\rightarrow$ `integer`

entfernt das am weitesten links stehende `digit` des Arguments

`integer_add: (integer, integer)` $\rightarrow$ `integer`

liefert die Summe zweier `integer`-Werte

`shift: (integer, integer)` $\rightarrow$ `integer`

fügt an das 2. Argument so viele Nullen an, wie das 1. Argument `digits` enthält

Formale Spezifikation eines Algorithmus (VII)

Anmerkungen:

- die Funktion `tail` ist nicht total

  weil sie für `integer`, die nur aus einer Ziffer bestehen, nicht definiert ist

  (die leere Sequenz ist nach Annahme kein `integer`-Wert)

- Der Formalismus, in dem `integer_mult` und `integer_digit_mult` spezifiziert sind, erinnert an eine Programmiersprache (— ist aber keine)

- Geltungsbereiche der Variablen sind durch Klammerung präzise definiert

Die folgenden Begriffe werden später noch (ausführlich) eingeführt:

- Funktionsdefinition und -aufruf
- Formale Parameter und (Aufruf-)Parameter
- Globale und lokale Variablen
- Datentyp

1.1.7. Eigenschaften eines Algorithmus: Partielle und totale Korrektheit

„Korrektheit" — eine wünschenswerte Eigenschaft eines Algorithmus

- aber was bedeutet, „ein Algorithmus ist korrekt" ?

  . . . er darf nie ein falsches Ergebnis liefern

- reicht das für Korrektheit aus?

  nein — damit wäre ein Algorithmus, der nie ein Ergebnis liefert (also nicht terminiert), immer korrekt, z.B.

```
function
(integer, integer) ---> integer
mult(x, y) = (mult(y, x))
```

Partielle und totale Korrektheit (II)

Man unterscheidet zwischen „partieller“ und „totaler“ Korrektheit:

1. Partielle Korrektheit:

   Liefert der betrachtete Algorithmus ein Ergebnis, so ist es das erwartete/richtige Ergebnis

   (Bsp.: liefert `mult(a, b)` einen Wert `c`, so gilt `c = a * b`)

2. Totale Korrektheit:

   Der Algorithmus terminiert für alle möglichen Eingaben und liefert jeweils das erwartete/richtige Ergebnis

   (Bsp.: für alle natürliche Zahlen `a` und `b` terminiert `mult(a,b)` und liefert das Produkt `a * b`)

1.1.8. Beweisprinzip der vollständigen Induktion

Viele (Korrektheits-) Beweise in der Informatik basieren auf vollständiger Induktion (vgl. auch den (informellen!) Terminierungsbeweis im Abschnitt 1.1.3)

Definition Definition (Beweisprinzip der vollständigen Induktion):

Sei $f : \mathbb{N} \rightarrow M$ eine (totale) Funktion von den natürlichen Zahlen in eine Menge M. Sei B die Menge $\{f(n) \mid n \in \mathbb{N}\}$.
Um zu zeigen, dass jedes Element b von B eine Eigenschaft E besitzt, genügt es zu zeigen:

1. Induktionsbasis: $f(0)$ besitzt die Eigenschaft E.
2. Induktionsschritt: Sei n eine beliebige natürliche Zahl. Wenn $f(n)$ die Eigenschaft E besitzt, so auch $f(n+1)$.

1.1.9. Programm

Wann sprechen wir von einer (formalen) Spezifikation, wann von einem Programm?

Eine formale Spezifikation eines Algorithmus, wie im Abschnitt 1.1.6 gegeben, ist kein Programm:

1. Manche verwendete Funktionen, wie `integer_add`, sind nur (formal oder informell) spezifiziert, aber nicht implementiert.
2. Die verwendeten Datenstrukturen sind nur spezifiziert, nicht implementiert, d.h. nicht unter Anwendung einer Programmiersprache auf Speicherstrukturen eines Computers abgebildet.

Ein Programm ist in einer (implementierten) Programmsprache formuliert und kann auf einem Rechner (mittels eines Interpreters oder Compilers) ausgeführt werden.

Programm (II)

Warum erst formal spezifizieren, wenn man dann den Algorithmus „ausprogrammieren" muss?

Formale Spezifikationen sind wichtig und voreiligen Programmentwürfen vorzuziehen:

1. *ad hoc*–Programmierung führt sehr oft zu fehlerhafter Software.
2. Jede Programmiersprache besitzt Eigenschaften, die manchmal die Entwicklung eines einfachen Algorithmus erschweren.
3. Die vollständige und präzise Spezifikation der Datentypen einer Anwendung erlaubt oft erst eine gute Implementierung in einer Programmiersprache.
4. Programmiersprachen wechseln, Algorithmen bleiben.

1.1.10. Eigenschaften eines Algorithmus: Zeit- und Speicherplatzkomplexität

Komplexität eines Algorithmus: Zeit- und Speicherbedarf

Der Algorithmus `integer_digit_mult` (Abschnitt 1.1.1) läuft wie folgt ab:

```
multiplicator x multiplicand
      (digit)      (integer)
              --------------
                    product1  = integer_digit_mult(multiplicator,
                                                   tail(multiplicand))
                               [ product2 =
                                   digit_mult(multiplicator,
                                              head(multiplicand)) ]
 + shift(tail(multiplicand),
                   product2)
  --------------------------
                      result
```

Zeit- und Speicherplatzkomplexität (II)

Länge von `product1` $\leq$ Länge von `tail(multiplicand)` $+ 1$
$=$ Länge von `multiplicand`

Länge von `product2` ≤ 2

Länge von `shift(tail(multiplicand), product2)`
$\leq 2 +$ Länge von `multiplicand` - 1
$< 2 +$ Länge von `multiplicand`

Länge von `result` $\leq\ 2 +$ Länge von `multiplicand`

max. Speicherbedarf: $\leq\ 1 + 2$ * Länge von `multiplicand`

Der Speicherbedarf der Funktion `integer_digit_mult` wächst also linear $(1 + 2x)$ in der Länge $(= x)$ ihres 2. Arguments.

Zeit- und Speicherplatzkomplexität (III)

Die Ermittlung des Zeitbedarfs verlangt, dass Zeiteinheiten festgelegt werden. z.B.:

1 Look-up („Nachschlagen“) in der Multiplikationstabelle	$=$	1 Zeiteinheit
Addition zweier digits	$=$	1 Zeiteinheit

Dann ergibt sich für die Funktion `integer_digit_mult` der folgende Zeitbedarf:

- so viele Look-ups in der Multiplikationstabelle wie `multiplicand` lang ist
- Wegen der Überträge:
  Anzahl der Additionen zweier digits $\leq$ (Länge von `multiplicand` - 1)

Der Zeitbedarf der Funktion `integer_digit_mult` wächst also höchstens linear $(2x - 1)$ in der Länge $(= x)$ ihres 2. Arguments.

1.2. Was ist Informatik?

Ursprung des Begriffs „Informatik":

- Wortschöpfung aus „Information" und „Elektronik"
  (1962 vorgeschlagen von dem (französischen) Ingenieur Philippe Dreyfus)
- im Englischen: „computing science" bzw. „computer science"

Einige mögliche Definitionen für „Informatik":

> **Definition** (Informatik) [DUDEN Informatik]:
>
> Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Computern.

Was ist Informatik? (II)

Definition (Informatik)
[Gesellschaft für Informatik (GI) e.V., www.gi-ev.de;
Studien- und Forschungsführer Informatik, Springer-Verlag]:

Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.

Definition (Computer Science)
[Association for Computing Machinery, www.acm.org]:

Computer Science is the systematic study of algorithms and data structures, specifically (1) their formal properties, (2) their mechanical and linguistic realizations, and (3) their applications.

Was ist Informatik? (III)

Wichtig: Informatik und Programmierung sind nicht das selbe!

Die Informatik setzt die Programmierung voraus, ist aber viel mehr als nur Programmierung.

Die Grundstudiumsvorlesungen geben einen Überblick in die Hauptbereiche der Informatik

$\rightsquigarrow$ damit kann die Frage „Was ist Informatik?“ immer weiter beantwortet werden

Die Informatik entwickelt sich (immer noch) extrem schnell — wie kaum ein anderes Fach.

Neue Anwendungsbereiche prägen dabei die Informatik mit.
(siehe z.B. Bioinformatik, Medieninformatik, . . .)

1.3. Die Programmiersprache der Vorlesung

Diese Vorlesung ist eine Einführung in die Informatik und in die Programmierung (bzw. deren Grundkonzepte).

- Dazu ist eine gut konzipierte Programmiersprache vorteilhaft.
- Programmiersprachen aus der Industrie haben erhebliche Mängel, die das (ordentliche) Erlernen der Programmierung wesentlich erschweren.
- hier als Programmiersprache gewählt: SML (Standard ML)
  - stammt aus der Forschung
  - SML wird seit Ende der 70er Jahre entwickelt
  - die heutige Form stammt im wesentlichen aus dem Jahr 1986
    - alt genug, so dass frühe Entwurfsmängel beseitigt werden konnten
    - jung genug, so dass moderne Programmiersprachenkonzepte berücksichtigt wurden

Die Programmiersprache der Vorlesung (II)

SML ist eine *Funktionale Programmiersprache*

→ ein SML-Programm kann als Sammlung von Gleichungen, die Funktionen (im mathematischen Sinne) beschreiben, verstanden werden

Nicht alle Programmiersprachen sind funktional.

Weitere sogenannte Berechnungsparadigmen, auf denen Programmiersprachen beruhen, sind:

- das imperative Paradigma,
- das logische Paradigma,
- das objektorientierte Paradigma.

(Manche Programmiersprachen kombinieren Aspekte aus verschiedenen Paradigmen.)

Die Programmiersprache der Vorlesung (III)

Ziel/Vision:

Während des Grundstudiums sollten Sie möglichst alle Paradigmen und (damit) mehrere Programmiersprachen kennenlernen.

Zum Beispiel in den Vorlesungen

- Informatik 1: funktionale Programmierung — am Beispiel SML
- Informatik 2: imperative und objektorientierte Programmierung — am Beispiel Java
- [Informatik 3: maschinennahe Programmierung / Assembler]

Die Programmiersprache der Vorlesung (IV)

Wichtig:

Wir wollen Sie nicht zu Experten in einer oder mehreren bestimmten Programmiersprachen ausbilden:

- Sie sollen die allen Programmiersprachen gemeinsamen Konzepte kennenlernen und verstehen.
- Damit sollen Sie in die Lage versetzt werden, sich jede (auch zukünftige) Programmiersprache in kürzester Zeit anzueignen.

  Es gibt zu viele Programmierspachen, als dass Sie sie während eines Studiums lernen könnten!

1.4. Inhaltsverzeichnis der Vorlesung

1.5. Literatur

zur Programmiersprache SML:

- das Skriptum zur Vorlesung
- weitere Literatur ist im Skriptum angegeben (und auf der www-Seite zur Vorlesung)

zur Einführung in die Informatik:

- das Skriptum zur Vorlesung
- weitere Literatur ist auf der www-Seite zur Vorlesung angegeben

Empfehlung:
Schauen Sie sich verschiedene Bücher (auch und gerade englischsprachige) an; vergleichen und prüfen Sie, welche Bücher für Sie geeignet sind — und kaufen Sie keine Bücher „auf Verdacht“!

2. Einführung in die Programmierung mit SML

Einführung in die (funktionale) Programmierung am Beispiel von SML

SML = „Standard ML“

SML ist eine Verfeinerung und Weiterentwicklung der Sprache ML (Meta Language)

SML und ML bezeichnen heute eigentlich die selbe Programmiersprache

[darüber hinaus gibt es noch einige leicht abweichende ML-Dialekte]

Überblick über den Inhalt dieses Kapitels

- 2.1 Antipasti
- 2.2 Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
- 2.3 Präzedenz- und Assoziativitätsregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
- 2.4 Namen, Bindungen und Deklarationen
- 2.5 Fallbasierte Definition einer Funktion
- 2.6 Definition von rekursiven Funktionen
- 2.7 Wiederdeklaration eines Namens - Statische Bindung - Umgebung
- 2.8 Totale und partielle Funktionen (Fortsetzung)
- 2.9 Kommentare
- 2.10 Die Standardbibliothek von SML
- 2.11 Beispiel: Potenzrechnung

2.1. Antipasti

[Antipasti — Sammlung von kleinen schmackhaften Appetitanregern]

- SML wird mit dem Linux-Kommando `sml` aufgerufen.
- Eine SML-Sitzung wird mit `^D` beendet (`Ctrl` + `D` bzw. `Strg` + `D` gleichzeitig).
- SML bietet eine interaktive, d.h. dialogorientierte, Benutzerschnittstelle mit einer sogenannten Treiberschleife:
  1. „`-`“ am Anfang der Zeile zeigt an, dass der Benutzer einen Ausdruck eingeben kann.
  2. Das Ende des Ausdrucks wird mit „`;`“ gekennzeichnet, und die Auswertung des Ausdruckes mit „enter“ (bzw. „return“ / Zeilenumbruch) anfordern.
  3. SML wertet dann den Ausdruck aus und liefert den ermittelten Wert in einer neuen Zeile.

Beispiel einer SML-Sitzung

```
linux% sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit

- 2;
val it = 2 : int

- ~1;
val it = ~1 : int

- ~(~2);
val it = 2 : int

-  2 * 3 + 1;
val it = 7 : int

- (2 * 3) + 1;
val it = 7 : int
```

Beispiel einer SML-Sitzung (II)

```
- 2 * (3+1);
val it = 8 : int

- 4 div 2;
val it = 2 : int

- 5 div 2;
val it = 2 : int

- 100 mod 4;
val it = 0 : int

- 012;
val it = 12 : int

- Ctrl+D
linux%
```

2.1.1. Der Datentyp „ganze Zahl“

- „`int`“ bezeichnet den (Daten-)Typ „integer“ oder „ganze Zahl“.
- In SML besitzt jeder Ausdruck einen (Daten-)Typ.
- Typen sind wichtige Bestandteile von (modernen) Programmiersprachen.
- „`it`“ bezeichnet den unbenannten Wert des Ausdrucks, dessen Auswertung angefordert wird.

  [*später:* Bindung von Werten an Namen]

Der Datentyp „ganze Zahl“ (II)

ganze Zahlen in SML:

- führende Nullen sind zulässig; z.B.

  `012` ist eine andere Notation für `12`,   `~0012` ist eine andere Notation für `~12`.

  [syntaktisch verschiedene Ausdrücke können den selben Wert repräsentieren!]

- vordefinierte Operationen über ganze Zahlen: `+`, `-`, `*`, `div`, `mod` (alle infix notiert)

- „Präzedenzen“ der vordefinierten Operationen müssen beachtet werden:

  `2 * 3 + 1` steht z.B. für `(2 * 3) + 1`.

- *Vorsicht:*

  `~` (Vorzeichen für negative Zahlen) und `-` (Subtraktion) sind nicht austauschbar.

- `~` ist ein unärer (d.h. einstelliger) Operator (oder Operation).

  `-`, `+`, `*`, `div` und `mod` sind binäre (d.h. zweistellige) Operatoren.

2.1.2. Gleichheit für ganze Zahlen

Zum Vergleich von ganzen Zahlen bietet SML die vordefinierte Funktion „=“:

```
- 2 = 2;
val it = true : bool

- 2 = 3;
val it = false : bool
```

Eine Funktion, die wie = als Wert entweder `true` oder `false` liefert, wird *Prädikat* oder auch *Test* genannt.

2.1.3. Der Datentyp „Boole'scher Wert“

- Die Werte der Ausdrücke `2 = 2` und `2 = 3` sind sogenannte „Wahrheitswerte“ oder Boole'sche Werte.

- Es gibt zwei Boole'sche Werte:

  `true` (wahr) und `false` (falsch).

- vordefinierte Operationen über Boole'sche Ausdrücke:

  `not` (Negation): unär; präfix notiert

  `andalso` (Konjunktion) und `orelse` (Disjunktion): binär; infix notiert

Der Datentyp „Boole'scher Wert“ (II)

```
- true;
val it = true : bool

- not true;
val it = false : bool

- not (not false);
val it = false : bool

- true andalso not false;
val it = true : bool

- false orelse true;
val it = true : bool

- not (2 = 3);
val it = true : bool
```

Der Datentyp „Boole'scher Wert" (III)

Der Ausdruck `not not false` kann nicht ausgewertet werden, weil er von SML wie `(not not) false` verstanden wird.

`(not not) false` ist aus zwei Gründen inkorrekt:

1. Die Teilausdrücke `(not not)` und `false` sind nicht mit einer Operation verbunden.

   `(not not) false` ist also genauso sinnlos wie etwa `2 4`.

2. Der Teilausdruck `(not not)` ist inkorrekt gebildet, weil die erste Negation auf keinen Booles'schen Ausdruck angewandt wird.

   `(not not)` ist genauso sinnlos wie etwa `(~ ~)`.

   [Übrigens ist `~ ~ 5` aus den gleichen Gründen inkorrekt.]

2.1.4. Gleichheit für Boole'sche Werte

vordefinierte Funktion =:

```
- true = not (not true);
  val it = true : bool

- false = (false andalso not true);
  val it = true : bool
```

Gleichheit für Boole'sche Werte (II)

Vergleich von Wahrheitswerten mit `=` ist fast immer schlechter Programmierstil.

Häufig werden unnötig komplizierte Konstruktuionen verwendet:

```
if Bedingung = true then Ausdruck else Ausdruck'
```

Aber: `Bedingung = true` hat den selben Wert wie `Bedingung`; einfacher ist

```
if Bedingung then Ausdruck else Ausdruck'
```

Ähnliches gilt für die folgenden gleichwertigen Ausdrücke:

```
if Bedingung = false then Ausdruck else Ausdruck'
```

```
if not Bedingung then Ausdruck else Ausdruck'
```

```
if Bedingung then Ausdruck' else Ausdruck
```

2.1.5. Überladen

Gleichheit für Boole'sche Ausdrücke und für ganze Zahlen:

- syntaktisch identischer Operator (=)
- aber zwei grundverschiedene Funktionen, weil ihre Argumente verschiedene Typen besitzen
  - $\rightsquigarrow$ SML betrachtet die Typen der Argumente, um zu entscheiden, welche Gleichheitsfunktion angewendet werden soll
- der Operator = ist „überladen" (*overloaded*)

Überladen: Derselbe Name (Bezeichner) wird zur Bezeichnung unterschiedlicher Operationen oder Funktionen verwendet, die vom System unterschieden werden.

Überladen von Bezeichnern ist nicht ungefährlich und sollte nur in klar abgegrenzten Fällen verwendet werden.

Überladen (II)

weitere überladene Operatoren in SML: + und *
(Addition und Multiplikation für ganze Zahlen als auch für reelle Zahlen):

```
- 2 + 3;
val it = 5 : int

- 2.1 + 3.3;
val it = 5.4 : real

- 2 * 3;
val it = 6 : int

- 2.1 * 3.3;
val it = 6.93 : real
```

Überladen (III)

Achtung:

- in der Mathematik sind ganze Zahlen ($\mathbb{Z}$) ein Untertyp (Teilmenge) der reellen Zahlen ($\mathbb{R}$)
- ganze Zahlen und reelle Zahlen stellen verschiedene (Daten-)Typen dar

  (in SML wie in den meisten anderen Programmiersprachen)
- im Rechner werden ganze Zahlen und reelle Zahlen völlig unterschiedlich repräsentiert

  $\leadsto$ unterschiedliche Implementierungen der arithmetischen Operationen

Überladen (IV)

Der Versuch, eine ganze Zahl und eine reelle Zahl zu addieren, führt folglich zu einer Fehlermeldung:

```
- 2 + 4.83;
stdIn:10.1-10.9 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    2 + 4.83
```

- Wegen des ersten Operanden 2 wird + als Addition für ganze Zahlen interpretiert.
- Da `4.83` keine ganze Zahl ist, wird ein Typfehler gemeldet.

2.1.6. Weitere Typen

SML bietet weitere häufig benötigte Typen, wie z.B.

- „Zeichen“ (wie `a`, `b`, `c`, usw.)
  und
- „Zeichenfolge“ (wie `dieser Text`)

[→ Kapitel 6]

Ferner ermöglicht SML die Definition von eigenen, maßgeschneiderten Typen, wie etwa

- eine beliebige Notenskala oder
- die Tage der Woche in einer beliebigen Sprache

[→ Kapitel 9]

2.1.7. Vergleichsfunktionen für ganze und für reelle Zahlen

Vordefinierte überladene Prädikate für ganze Zahlen und für reelle Zahlen:

<	(echt kleiner)
>	(echt größer)
<=	(kleiner gleich)
>=	(größer gleich)

Nur für ganze Zahlen zusätzlich: <> (Negation der Gleichheit)

Vorsicht:
= und <> sind für reelle Zahlen nicht zulässig.

Vergleichsfunktionen für ganze und für reelle Zahlen (II)

Für reelle Zahlen bietet SML die Funktion `Real.compare(x, y)` an:

```
- Real.compare(1.0,7.0);
val it = LESS : order

- Real.compare(100.0,1.0);
val it = GREATER : order

- Real.compare(1.0,1.0);
val it = EQUAL : order
```

Achtung:
Die Funktion `Real.compare` ist kein Prädikat, da sie keine Boole'schen Werte liefert.

[den Ergebnistyp `order` behandeln wir erst später]

Vergleichsfunktionen für ganze und für reelle Zahlen (III)

SML bietet auch die Gleichheitsfunktion `Real.==` für reelle Zahlen, die den Typ `order` nicht verwendet:

```
- Real.==(2.5, 2.5);
val it = true : bool

- Real.==(2.5, 3.0);
val it = false : bool
```

Vorsicht:

Aufgrund der internen Repräsentation reeller Zahlen ist diese Gleichheitsfunktion von der mathematischen Gleichheit zu unterscheiden!

2.1.8. Weitere nützliche Funktionen für ganze Zahlen

`Int.abs`: Betrag einer ganzen Zahl

```
- Int.abs(~4);
val it = 4 : int
```

`Int.min`: Minimum zweier ganzen Zahlen

```
- Int.min(5,2);
val it = 2 : int
```

`Int.max`: Maximum zweier ganzen Zahlen

```
- Int.max(3,5);
val it = 5 : int
```

Weitere nützliche Funktionen für ganze Zahlen (II)

`Int.sign`: „Vorzeichen“ einer ganzen Zahl

```
- Int.sign(0);
val it = 0 : int

- Int.sign(~5);
val it = ~1 : int

- Int.sign(6);
val it = 1 : int
```

2.2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung

Präzisierung der Begriffe

- Ausdrücke
- Werte
- Typen
- (polymorphe) Typüberprüfung

und ihrer Zusammenhänge

2.2.1. Ausdrücke, Werte und Typen

SML (genauer: das SML-System) wertet Ausdrücke aus.

Ein Ausdruck kann

- atomar sein — wie `2`, `4` oder `false` — oder
- zusammengesetzt sein — wie `2 + 4` und `not (false andalso true)`.

Jeder korrekt gebildete Ausdruck hat einen Typ:

- ein Typ ist eine Menge von Werten;
  z.B. die Menge der ganzen Zahlen
- ein Ausdruck hat (meistens) auch einen Wert;
  dieser Wert ist ein Element des Typs des Ausdrucks
- manche Ausdrücke haben keinen Wert;
  z.B. `1 div 0` (da hier eine nicht-totale Funktion verwendet wird)

Ausdrücke, Werte und Typen (II)

- Atomare Ausdrücke wie `true` und `false` kann man oft mit ihren Werten identifizieren.
- *Aber:* `02` und `2` sind verschiedene atomare Ausdrücke, die den selben Wert haben.

  (da führende Nullen bei ganzen Zahlen erlaubt sind)
- Atomare Ausdrücke, wie `div` oder `orelse`, deren Werte Funktionen sind, sind prinzipiell von ihren Werten zu unterscheiden.
- Zusammengesetzte Ausdrücke sind damit auf keinen Fall mit ihren Werten identisch.

$\rightsquigarrow$ üblicherweise wird strikt zwischen Ausdrücken und Werten unterschieden!

Ausdrücke, Werte und Typen (III)

Auch Operationen (und allgemein Funktionen) haben Typen, z.B.

- die Funktion +
  - erhält als Argumente zwei (atomare oder zusammengesetzte) Ausdrücke vom Typ „ganze Zahl" und
  - liefert einen Wert ebenfalls vom Typ „ganze Zahl".
- die Gleichheit für ganze Zahlen ist eine Funktion,
  - die als Argumente zwei Ausdrücke vom Typ „ganze Zahl" erhält und
  - einen Wert vom Typ „Boole'scher Wert" liefert.

Man schreibt:

$$\begin{array}{lclcl} + & : & (\texttt{int, int}) & \rightarrow & \texttt{int} \\ = & : & (\texttt{int, int}) & \rightarrow & \texttt{bool} \end{array}$$

Bei Bildung zusammengesetzter Ausdrücke muss immer auf die Typen der verwendeten Operationen und der eingesetzten Teilausdrücke geachtet werden.

2.2.2. Typen in Programmiersprachen

Es gibt zwei grundlegende Ansätze bezügl. Typen in Programmiersprachen:

- schwach typisierte Programmiersprachen (z.B. Prolog, Lisp)
- stark (oder streng) typisierte Programmiersprachen (z.B. Pascal, Modula)

Schwache Typisierung:

Ausdrücke wie `8.0 + 1` (Summe einer reellen und einer ganzen Zahl) werden akzeptiert:

Bei der Auswertung wird automatisch die ganze Zahl `1` in eine reelle Zahl umgewandelt. [$\rightsquigarrow$ (automatische) „Typanpassung“]

Stark Typisierung:

Für jeden Namen (Bezeichner) muss ein Typ explizit angegeben werden.

Notwendige Typanpassungen müssen selbst programmiert werden.

Typen in Programmiersprachen (II)

Für Typanpassungen zwischen reellen und ganzen Zahlen bietet SML die vordefinierte Funktionen `real` und `round`:

```
- real(1);
val it = 1.0 : real

- round(8.12);
val it = 8 : int

- round(8.99);
val it = 9 : int

- round(8.0);
val it = 8 : int
```

[`round` dient nicht nur der Typanpassung, sondern auch zum Auf- bzw. Abrunden]

Typen in Programmiersprachen (III)

- SML verfolgt einen Mittelweg zwischen schwach typisierten und stark typisierten Programmiersprachen:

  $\leadsto$ die sogenannte „polymorphe Typüberprüfung“ (*polymorphic type checking*)

  Anstatt immer die explizite Angabe von Typen zu verlangen, ermittelt SML — wenn möglich — die Typen der Bezeichner selbst.

2.3. Präzedenz- und Assoziativitätsregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken

Schon betrachtet:

- `2 * 3 + 1` steht für den Ausdruck `(2 * 3) + 1`
- `not not false` steht für den (inkorrekt gebildeten) Ausdruck `(not not) false`

Dahinter stehen zwei Begriffe:

- Präzedenzen und
- Assoziativitätsregeln

für Operatoren.

Präzedenzen und Assoziativitätsregeln

Präzedenzen von Operatoren legen implizite Klammerung bei unzureichend oder nicht geklammerten Ausdrücken fest.

- Man sagt, `*` bindet stärker als `+`; daher steht z.B. `2 * 3 + 1` für `(2 * 3) + 1`.

  [Obwohl diese Annahme üblich ist, könnte eine Programmiersprache genauso auf der Annahme beruhen, dass `*` weniger stark als `+` bindet.]

Assoziativitätsregeln legen fest, ob fehlende Klammerungen von links oder rechts her einzusetzen sind:

- z.B. ob `2 + 3 + 4` für `(2 + 3) + 4` oder für `2 + (3 + 4)` steht.
- `+` und `*` sind in SML linksassoziativ, d.h. `2 + 3 + 4` steht für `(2 + 3) + 4`.

In vielen Fällen liefern beide Klammerungen den selben Wert, aber nicht immer:

Operation „-“ (Subtraktion):	linksassoziativ	$\Rightarrow$	`10 - 7 - 1` hat den Wert `2`
	rechtsassoziativ	$\Rightarrow$	`10 - 7 - 1` hat den Wert `4`

In manchen Sprachen bestimmen die Assoziativitätsregeln auch die Auswertungsreihenfolge.

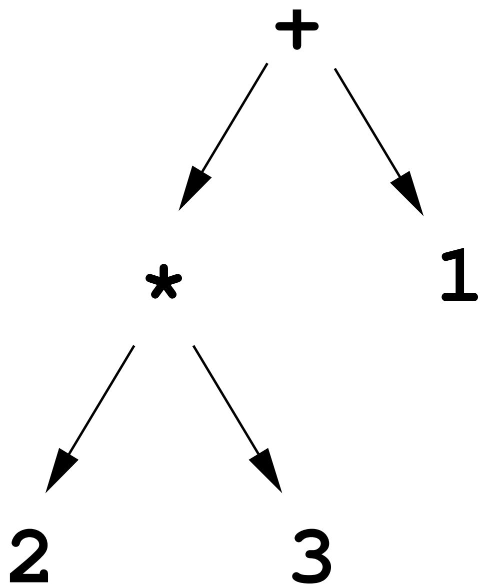
Präzedenzen und Assoziativitätsregeln (II)

Ausdrücke, die SML zur Auswertung weitergereicht werden, sind *linear*, weil sie aus einer Folge von Zeichen bestehen: z.B. `(2 * 3) + 1` und `2 * 3 + 1`

SML analysiert zunächst die Syntax solcher Ausdrücke, bevor die Ausdrücke ausgewertet werden.

Die Syntaxanalyse von `(2 * 3) + 1` führt zur Bildung einer baumartigen Struktur:

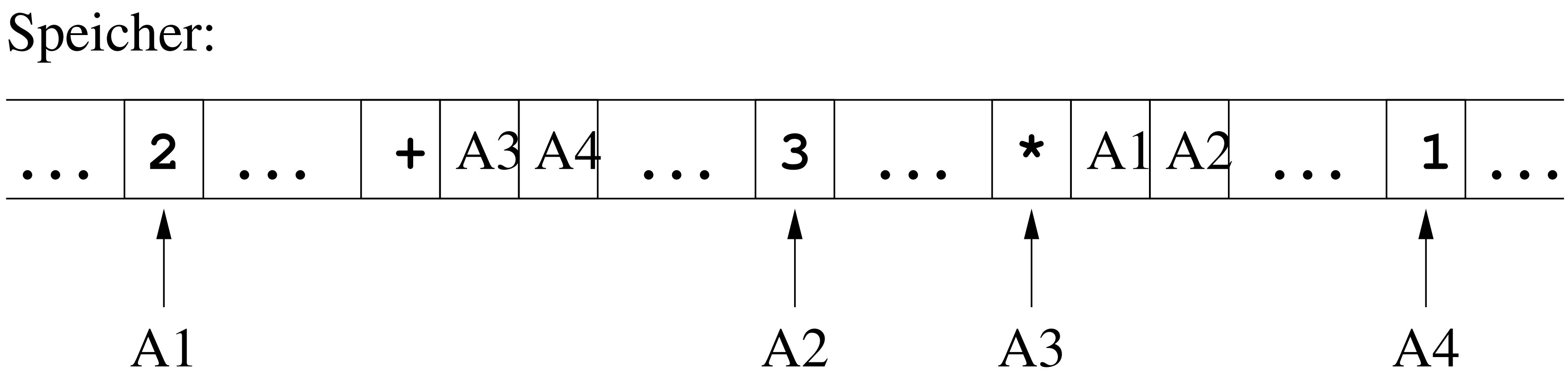
(*)



Präzedenzen und Assoziativitätsregeln (III)

Dabei stellen die gerichteten Kanten im (linear angeordneten) Speicher Zeiger (d.h. hier Speicheradressen A1, . . . , A4) dar:

(**)



Präzedenzen und Assoziativitätsregeln (IV)

Die Syntaxanalyse ist aus zwei Gründen notwendig:

1. Sie ermöglicht die Auslegung (Interpretation) unvollständig geklammerter Ausdrücke — wie etwa `4 + 5 + 6`.
2. Sie ersetzt die sogenannte „konkrete Syntax“ von Ausdrücken, d.h. die vom Programmierer verwendete Darstellung, durch die sogenannte „abstrakte Syntax“, d.h. die Repräsentation im Speicher durch „Bäume“ à la (**), die von SML zur Auswertung verwendet wird.

Da `2 * 3 + 1` in SML für `(2 * 3) + 1` steht, führt die Analyse von `2 * 3 + 1` zur Bildung des selben Baumes wie für `(2 * 3) + 1`.

Die baumartige Repräsentation von Ausdrücken entspricht der Auswertung der Ausdrücke.

2.4. Namen, Bindungen und Deklarationen

- Mit einer „Deklaration“ kann ein Wert an einen „Namen“ gebunden werden.

- Mögliche Werte, die an Namen gebunden werden können, sind u.a.
  - Konstanten und
  - Funktionen.

$\leadsto$ Kostantendeklaration

$\leadsto$ Funktionsdeklaration

2.4.1. Konstantendeklaration — Wertdeklarationen

```
- val zwei = 2;
val zwei = 2 : int
```

Damit kann nun der Namen `zwei` genauso wie die Konstante 2 verwendet werden:

```
- zwei + zwei;
val it = 4 : int

- zwei * 8;
val it = 16 : int
```

Allgemein spricht man auch von einer Wertdeklaration. [daher `val` (*value*)]

- Alle Konstantendeklarationen sind Wertdeklarationen,
- aber nicht alle Wertdeklarationen sind Konstantendeklarationen — siehe unten.

2.4.2. Funktionsdeklaration

```
- fun zweimal(x) = 2 * x;
val zweimal = fn : int -> int
```

Anstelle des Wertes der Funktion, die an den Namen `zweimal` gebunden wird, gibt SML die Kurzmitteilung „`fn`“ (für Funktion) aus.

Der Wert des Namens `zweimal` ist die Funktion, die als Eingabe eine ganze Zahl erhält und das Doppelte dieser Zahl als Ausgabe liefert.

Nachdem eine Funktion deklariert wurde, kann sie aufgerufen werden:

```
- zweimal(8);
val it = 16 : int

- zweimal(zweimal(8));
val it = 32 : int
```

Funktionsdeklaration (II)

Neben dem Platzhalter „`fn`“ liefert SML den Typ der deklarierten Funktion.

Der Typ `int -> int` wird wie folgt ermittelt:

- Da 2 eine ganze Zahl ist, steht die überladene Operation `*` für die Multiplikation ganzer Zahlen.
- Folglich muss `x` vom Typ „ganze Zahl“ sein (daher: `int ->`).
- Da `*` die Multiplikation ganzer Zahlen ist, ist der von `zweimal` berechnete Wert eine ganze Zahl (daher: `-> int`).

Anstatt von *Funktionsdeklaration* spricht man auch von *Funktionsdefinition*.

Die folgenden Deklarationen sind gleichwertig:

```
- fun zweimal (x) = 2 * x;

- fun zweimal x = 2 * x;
```

2.4.3. Funktion als Wert — Anonyme Funktion

Für SML ist eine Funktion ein Wert.

$\leadsto$ Verwendung des Deklarationskonstrukts `val` möglich

Die Funktion `zweimal` kann z.B. wie folgt definiert werden:

```
val zweimal = fn x => 2 * x;
```

Hier passiert folgendes:

- Der Teil `fn x => 2 * x` definiert eine *anonyme Funktion*.
  [`fn` wird oft „lambda“ (λ) ausgesprochen.]
- Diese anonyme Funktion wird an den Namen `zweimal` gebunden.

Vorsicht: Verwechseln Sie die SML-Konstrukte `fn` und `fun` nicht!

2.4.4. Formale und aktuelle Parameter einer Funktion

In der *Funktionsdeklaration*

```
fun zweimal(x) = 2 * x;
```

ist `x` ein *formaler Parameter* (der Funktionsdeklaration oder Funktionsdefinition).

Im *Funktionsaufruf* `zweimal(8)` ist 8 der *aktuelle Parameter* (des Funktionsaufrufes).

Formale Parameter haben eine ähnliche Bedeutung wie Pronomen in natürlichen Sprachen:

Um zweimal von ETWAS zu berechnen, multipliziere ES mit 2.

2.4.5. Rumpf oder definierender Teil einer Funktionsdeklaration

Der *Rumpf* (auch: definierender Teil) einer Funktionsdeklaration ist der Teil nach dem Zeichen „=“.

Im Falle der folgenden Deklaration der Funktion `zweimal`

```
fun zweimal(x) = 2 * x;
```

ist der Rumpf: `2 * x`.

2.4.6. Namen, Variablen und Bezeichner

Die Begriffe

- Name
- Variable
- Bezeichner

haben dieselbe Bedeutung.

2.4.7. Typ-Constraints

Da `x + x = 2 * x` ist, könnte man die Funktion `zweimal` wie folgt definieren:

```
- fun zweimal(x) = x + x;
```

Dies wird aber nicht von allen SML-Systemen als korrekt angenommen:

- Der Typ des formalen Parameters `x` ist nicht eindeutig ($\rightarrow$ `int` oder `real`).
- Manche Systeme nehmen an, dass `x` den Typ `int` hat, weil sie im Zweifel + als Addition von ganzen Zahlen annehmen.
- Andere SML-Systeme machen keine solche Annahme und verwerfen die vorangehende Funktionsdeklaration als inkorrekt.

Typ-Constraints (auch: Typisierungsausdrücke) ermöglichen, die fehlende Information anzugeben.

Typ-Constraints (II)

Mit

```
- fun zweimal x: int = x + x;
```

wird der Typ des Ergebnisses (des berechneten und gelieferten Wertes) angegeben;

mit

```
- fun zweimal(x: int) = x + x;
```

wird der Typ des Parameters angegeben;

mit

```
- fun zweimal(x: int): int = x + x;
```

werden sowohl der Typ des Ergebnis als auch der Typ des Parameters angegeben.

Typ-Constraints (III)

Mit einem Typ-Constraint wird die folgende Funktion für reelle Zahlen definiert:

```
- fun reell_zweimal x:real = x + x;
val reell_zweimal = fn : real -> real

- val pi = 3.1416;
val pi = 3.1416 : real

-  reell_zweimal pi;
val it = 6.2832 : real
```

[Vor und nach dem „ : “ sind in einem Typ-Constraint Leerzeichen zulässig.]

2.4.8. Syntax von Namen

SML unterscheidet zwischen „alphabetischen“ und „symbolischen Namen“.

$\longrightarrow$ aber kein Unterschied in der Verwendung !

Alphabetische Namen fangen mit einem (kleinen oder großen) Buchstaben an, dem endlich viele (auch null) Buchstaben (`a ...z A ...Z`), Ziffern (`0 1 2 ...9`), Unterstriche (`_`) und Hochkommata (single quote: `'`) folgen.

Symbolische Namen sind (endliche) Folgen der folgenden Zeichen:

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

Syntax von Namen (II)

Beispiel:

```
- val @#!@@@ = 12;
val @#!@@@ = 12 : int

- fun $@#? x = 5 * x;
val $@#? = fn : int -> int
```

Vorsicht:
Die folgenden symbolischen Namen haben in SML eine vordefinierte Bedeutung:

`:   |   =   =>   ->   #`

2.4.9. Dateien laden (einlesen)

Bis jetzt haben wir nur die interaktive Eingabe von Deklarationen betrachtet.

Umfangreiche sowie häufig verwendete Funktionsdeklarationen sollte man in einer Datei speichern.

Eine Datei, die z.B. `meine_datei.sml` heißt, kann wie folgt geladen werden:

```
- use("meine_datei.sml");
val it = () : unit
```

Dabei ist `()` (gesprochen „unity“) der einzige Wert eines besonderen Datentyps namens `unit`.

`unit` wird für Funktionen verwendet, die eigentlich keinen Wert berechnen, sondern einen *Nebeneffekt* bewirken (wie das Laden von Funktionsdeklarationen aus einer Datei).

2.5. Fallbasierte Definition einer Funktion

In Deklarationen bzw. Definitionen werden häufig *Fallunterscheidungen* benötigt.

SML bietet dafür zwei Möglichkeiten:

- if-then-else ($\rightarrow$ 2 Fälle)
- Pattern Matching ($\rightarrow$ beliebig viele Fälle)

2.5.1. if-then-else

Eine Funktion `Vorzeichen` kann z.B. wie folgt definiert werden:

```
fun Vorzeichen(x : int) = if x > 0 then 1
                          else if x < 0 then ~1
                               else 0;
```

Das Konstrukt `if Test then E1 else E2` stellt die Anwendung einer wie folgt definierten Funktion auf `Test` dar:

```
(fn true => E1 | false => E2)
```

[`if Test then E1 else E2` entspricht somit `(fn true => E1 | false => E2)(Test)`.]

if-then-else (II)

Der `else`-Teil von if-then-else-Ausdrücken muss immer angegeben werden.
(im Gegensatz zu vielen (imperativen) Programmiersprachen)

- Ohne else-Teil hat z.B. `if B then A` keinen Wert,
  wenn die Bedingung `B` den Wert `false` hat.

  $\longrightarrow$ in der funktionalen Programmierung unmöglich!

In einem SML-Ausdruck `if B then A1 else A2` müssen `A1` und `A2` denselben Typ haben.

2.5.2. Pattern Matching (Musterangleich)

In der Definition der obigen anonymen Funktion sind zwei Aspekte bemerkenswert:

1. „|“ drückt eine Alternative aus.
2. Die Ausdrücke `true` und `false` stellen Muster (*patterns*) dar.

   „Matcht“ der Wert des aktuellen Parameters mit dem ersten Muster, so wird der Wert des Ausdrucks `E1` geliefert. Ansonsten wird getestet, ob der Wert des aktuellen Parameters mit dem zweiten Muster „matcht“.

Es können mehr als zwei Fälle in der Deklaration vorkommen:

- Die Muster werden sequenziell in der Reihenfolge der Definition probiert, bis eines mit dem Wert des aktuellen Parameters „matcht“.
- Das Muster _ (wildcard) stellt einen Fangfall dar, d.h. matcht mit jedem möglichen Wert des aktuellen Parameters.

  Das Wildcard-Symbol wird nicht im Rumpf eines Falles (also hinter =>) verwendet.

Pattern Matching (II)

Das folgende Prädikat liefert `true`, wenn es auf eine ganze Zahl angewandt wird, die eine (nicht-negierte) Ziffer ist:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
              | 5 => true
              | 6 => true
              | 7 => true
              | 8 => true
              | 9 => true
              | _ => false;
```

Vorsicht: Pattern sind keine Tests wie etwa (`x > 0`), sondern mögliche Werte des Parameters.

[Pattern Matching wird auf Deutsch auch „Angleich" oder „Musterangleich" genannt.]

2.6. Definition von rekursiven Funktionen

am Beispiel der Berechnung der Summe der n ersten natürlichen Zahlen:

- rekursive Definition
- effiziente (nicht-rekursive) Definition
- Induktionsbeweis (für die Summenformel)
- alternativer (nicht-induktiver) Beweis
- Terminierungsbeweis

2.6.1. Rekursive Berechnung der Summe der n ersten natürlichen Zahlen

Funktion `summe`, die zu jeder natürlichen Zahl `n` die Summe aller natürlichen Zahlen von 0 bis einschließlich `n` liefert; z.B. wie folgt definiert:

$$\mathtt{summe}(\mathtt{n}) = \begin{cases} 0 & \text{falls } \mathtt{n} = 0 \\ \mathtt{n} + \mathtt{summe}(\mathtt{n} - 1) & \text{falls } \mathtt{n} > 0 \end{cases}$$

was in SML in einer der folgenden Weisen programmiert werden kann:

```
fun summe(n) = if n = 0 then 0 else n + summe(n-1);
```

oder

```
val rec summe = fn 0 => 0 | n => n + summe(n-1);
```

Man beachte das Wort `rec`!
Bei rekursiven Funktionen muss `rec` nach `val` angegeben werden.

2.6.2. Effiziente Berechnung der Summe der n ersten natürlichen Zahlen

`summe` kann auch wie folgt definiert werden:

$$\mathtt{summe}(\mathtt{n}) = \frac{\mathtt{n} * (\mathtt{n} + 1)}{2}$$

$\leadsto$ wesentlich effizientere Berechnung, weil für jedes `n` nur drei Grundoperationen benötigt werden

Diese Definition kann in SML wie folgt formuliert werden:

```
fun summe(n) = n * (n + 1) div 2;
```

2.6.3. Induktionsbeweis

Warum gilt $\texttt{summe}(\texttt{n}) = \frac{\texttt{n}*(\texttt{n}+1)}{2}$?

Dies kann durch vollständige Induktion gezeigt werden.

Beweis:

Induktionsbasis: Für $\texttt{n} = 0$ gilt die Gleichung $\texttt{summe}(\texttt{n}) = \frac{\texttt{n}*(\texttt{n}+1)}{2}$ offenbar, weil $\frac{0*(0+1)}{2} = 0$.

Induktionsschritt:

Induktionsannahme $(*)$: Für ein $k \in \mathbb{N}$ gelte: $\texttt{summe}(\texttt{k}) = \frac{\texttt{k}*(\texttt{k}+1)}{2}$.

Nun zeigen wir, dass die Gleichung auch für die Nachfolgerzahl $k+1$ gilt:

$$\begin{aligned} \texttt{summe}(\texttt{k}+1) &= \texttt{k}+1+\texttt{summe}(\texttt{k}) & \overset{(*)}{=} \texttt{k}+1+\frac{\texttt{k}*(\texttt{k}+1)}{2} \\ &= \frac{2(\texttt{k}+1)+\texttt{k}*(\texttt{k}+1)}{2} & = \frac{(\texttt{k}+2)*(\texttt{k}+1)}{2} \end{aligned}$$

qed.

2.6.4. Alternativer Beweis

Beweis:

Sei $n \in \mathbb{N}$.

Fall 1: n ist gerade.

Gruppiere die Zahlen von 1 bis n zu Paaren: $(n, 1)$, $(n-1, 2)$, $(n-2, 3)$, $\ldots$.

Das letzte Paar ist $((n/2) + 1, n/2)$. Die Summe jedes Paares ist $n + 1$ und es gibt $n/2$ Paare, also $\texttt{summe}(n) = n * (n + 1)/2$.

Fall 2: n ist ungerade.

Gruppiere die Zahlen von 0 bis n zu Paaren: $(n, 0)$, $(n-1, 1)$, $(n-2, 2)$, $\ldots$.

Das letzte Paar ist $(\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor)$. Die Summe jedes Paares ist n und es gibt $(n + 1)/2$ Paare, also $\texttt{summe}(n) = n * (n + 1)/2$.

qed.

2.6.5. Terminierungsbeweis

Bei der Programmierung rekursiver Funktionen treten häufig Denk- oder Programmierungsfehler auf:

$\rightsquigarrow$ Terminierung dann nicht gesichert

Zum Beispiel:

```
fun s(n) = n + s(n+1);
```

Die Terminierung einer rekursiven Funktion wie `summe` kann unter Anwendung der Beweistechnik der vollständigen Induktion gezeigt werden.

Wir zeigen, dass für alle (positiven) ganzen Zahlen `n` der Aufruf `summe(n)` terminiert.

Terminierungsbeweis (II)

Beweis:

Basisfall: `summe(0)` terminiert, weil der Aufruf `summe(0)` den Wert `0` liefert.

Induktionsschritt:

Annahme: Für eine natürliche Zahl $\mathtt{k} \in \mathbb{N}$ terminiere der Aufruf `summe(k)`.

Zu zeigen ist, dass der Aufruf `summe(k + 1)` terminiert:

`summe(k + 1)` = `k + 1 + summe(k)` (gemäß Funktionsdeklaration)

Da der Aufruf `summe(k)` terminiert, terminiert auch `summe(k + 1)`.

qed.

Beweisen oder Testen?

Durch Testen kann man Fehler finden, aber im Allgemeinen nicht ausschließen
(da man nicht alle, i.d.R. unendliche viele, Fälle testen kann).
Daher sind Beweise unabdingbare Bestandteile guter Programmentwicklung.

2.7. Wiederdeklaration eines Namens — Statische Bindung — Umgebung

- Deklaration $\longrightarrow$ Wiederdeklaration
- statische versus dynamische Bindung
- Umgebung

2.7.1. Wiederdeklaration eines Namens

Betrachten wir die folgende Sitzung:

```
- val zwei = 2;
val zwei = 2 : int

- fun zweimal(n) = zwei * n;
val zweimal = fn : int -> int

- zweimal(9);
val it = 18 : int

- val zwei = 0;
val zwei = 0 : int

- zweimal(9);
val it = 18 : int

- fun zweimal'(n) = zwei * n;
val zweimal' = fn : int -> int

- zweimal'(9);
val it = 0 : int
```

Die Bindung eines Wertes (Konstante oder Funktion) an einen Namen darf durch eine neue Deklaration verändert werden.

$\longrightarrow$ Wiederdeklaration (*redeclaration*/*redefinition*)

2.7.2. Statische und dynamische Bindung

Die Wiederdeklaration eines Namens gilt *nicht* für Deklarationen, die diesen Namen vor der Wiederdeklaration verwendet haben.

> So steht `zwei` für `2` in der Deklaration der Funktion `zweimal`,
> für `0` in der Deklaration der Funktion `zweimal'`.

Die Bindung in SML ist eine „statische Bindung“ (oder „lexikalische Bindung“).

Würde die Wiederdeklaration eines Namens `N` Einfluss auf Funktionen haben, deren Rümpfe sich auf `N` beziehen, so würde man von einer „dynamischen Bindung“ sprechen.

Die Wiederdeklaration von Namen und ihre Behandlung durch SML ist eine große Hilfe bei der Entwicklung von Programmen, die viele Namen verwenden.

2.7.3. Umgebung

Das SML-System verwaltet mit jeder Sitzung und jeder eingelesenen Datei, d.h. Programm, eine geordnete Liste von Gleichungen der Gestalt Name = Wert, die *Umgebung* heißt.

Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N = W` am Anfang der Umgebung.

Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. So gilt immer als Wert eines Namens `N` derjenige Wert, der bei der letzten Deklaration von `N` angegeben wurde.

Kommt ein Name `A` im Wertteil `W` einer Deklaration `val N = W` oder `val rec N = W` oder `fun N = W` vor, so wird der Wert von `A` ermittelt und in `W` anstelle von `A` eingefügt, bevor der Eintrag für `N` in der Umgebung gespeichert wird. So verändert eine spätere Wiederdeklaration von `A` den Wert von `N` nicht.

2.8. Totale und partielle Funktionen (Fortsetzung)

Der Unterschied zwischen totalen und nichttotalen (also partiellen) Funktionen ist für die Programmierung vom Belang.

Die rekursive Funktion `summe` mit Typ `int` $\rightarrow$ `int`

```
fun summe(n) = if n = 0 then 0 else n + summe(n-1);
```

ist über den ganzen Zahlen nicht total (z.B. terminiert `summe(~25)` nicht).
Über den natürlichen Zahlen ist diese Funktion aber total.

Es ist wichtig zu wissen, über welchem Bereich eine programmierte Funktion total ist, damit die Funktion auch nur mit entsprechenden Parametern aufgerufen wird.

Oft werden Eigenschaften der Funktion nur bezüglich dieses Bereiches angegeben.

2.9. Kommentare

In SML sind Kommentare beliebige Texte, die

- mit `(*` anfangen und
- mit `*)` enden.
- Geschachtelte Kommentare sind erlaubt.

Beispiel:

```
(*
fun Vorzeichen(x : int) = if x > 0 then 1
                          else if x < 0 then ~1
                               else (* x = 0 *) 0;
*)
```

Klare und präzise Kommentare sind in jedem Programm unabdingbar.
Es ist naiv anzunehmen, dass ein Programm selbsterklärend sei.

2.10. Die Standardbibliothek von SML

Manche der vordefinierten Funktionen von SML wie `Real.compare` sind in sogenannten Modulen programmiert, d.h. in Programmen, andere wie + sind Teile des SML-Systems. Die SML-Bezeichnung für Module ist „Struktur“ (structure).

Eine Funktion `F`, die in einem Modul `M` definiert ist, wird außerhalb dieses Moduls als `M.F` bezeichnet — und aufgerufen.

Die Standardbibliothek stellt eine Sammlung von Modulen (Strukturen) für herkömmliche Typen wie reelle Zahlen dar. Die Module der Standardbibliothek werden vom SML-System automatisch geladen. Das Laden von anderen Modulen muss aber vom Programmierer explizit angefordert werden.

Siehe „The Standard ML Basis Library“ unter

http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/

2.11. Beispiel: Potenzrechnung

am Beispiel der Potenzfunktion:

- einfache (direkte) rekursive Funktionsdefinition
- Terminierungsbeweis
- Zeitbedarf der einfachen rekursiven Definition
- eine effizientere (rekursive) Definition
- Zeitbedarf der effizienteren Definition
- weitere Verbesserung der Implementierung (Ausblick)

2.11.1. Einfache Potenzrechnung

Es sei die folgende Funktion in SML zu programmieren:

$$\begin{array}{llcl} \texttt{potenz:} & \mathbb{Z} \times \mathbb{N} & \rightarrow & \mathbb{Z} \\ & (a, b) & \mapsto & a^b \end{array}$$

[Die Potenz ist übrigens keine vordefinierte Funktion in SML.]

Die folgenden Gleichungen liefern die Grundlage für ein rekursives Programm:

$$\begin{array}{ll} a^b = 1 & \text{falls } b = 0 \\ a^b = a * a^{b-1} & \text{andernfalls} \end{array}$$

Daraus folgt die folgende Implementierung in SML:

```
fun potenz(a, b) = if b = 0 then 1 else a * potenz(a, b - 1);
```

2.11.2. Terminierungsbeweis für die einfache Potenzrechnung

Wir beweisen induktiv, dass für alle $(a, b) \in (\mathbb{Z} \times \mathbb{N})$ der Aufruf von `potenz(`a`, `b`)` terminiert.

Beweis:

Sei a eine beliebige ganze Zahl.

Basisfall: $b = 0$. Nach Funktionsdeklaration terminiert der Aufruf und liefert 1.

Induktionsschritt:

Annahme: Für ein gegebenes $b \in \mathbb{N}$ terminiere der Aufruf `potenz(`a`, `b`)`.

Nach Definition berechnet der Aufruf `potenz(`a`, `$b + 1$`)` den Wert von a `* potenz(`a`, `b`)` — er terminiert also nach Induktionsannahme.

qed.

2.11.3. Zeitbedarf der einfachen Potenzberechnung

Der Zeitbedarf wird als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt.

Dies stellt eine Vergröberung dar, weil Multiplikationen kleiner Zahlen weniger Zeit verlangen als Multiplikationen großer Zahlen.

[Solche vergröbernden Annahmen sind bei Schätzungen des Zeitbedarfs üblich.]

- Berechnung von `potenz(a, 0)` bedarf keiner Multiplikation
- Berechnung von `potenz(a, b+1)` bedarf einer Multiplikation mehr als Berechnung von `potenz(a, b)`

  $\leadsto$ Berechnung von `potenz(a, b)` benötigt insgesamt `b` Multiplikationen

$\Rightarrow$ Zeitbedarf der Funktion `potenz` ist *linear im zweiten Argument*

2.11.4. Effizientere Potenzrechnung

Ist b gerade mit $b = 2k$, so gilt: $a^b = a^{2k} = (a^k)^2$.

Für gerade natürliche Zahlen b kann die b-Potenz einer ganzen Zahl a also mit weniger als b Multiplikationen berechnet werden.

Diese Beobachtung führt zur folgenden Funktionsdeklaration:

```
fun potenz'(a, b) = if b = 0
                    then 1
                    else if gerade(b)
                         then quadrat(potenz'(a, b div 2))
                         else a * potenz'(a, b - 1);
```

wobei die Hilfsfunktionen `gerade` und `quadrat` wie folgt deklariert werden:

```
fun gerade(a) = (a mod 2 = 0);
fun quadrat(a : int) = a * a;
```

2.11.5. Zeitbedarf der effizienteren Potenzberechnung

Der Zeitbedarf wird wieder als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt.

[Rechenzeiten für die Aufrufe des Prädikats `gerade` werden vernachlässigt.]

So geschätzt ist die Rechenzeit abhängig von `b` und unabhängig von `a`.

Sei also $rz(\texttt{b})$ die Rechenzeit eines Aufrufes von `potenz'(a, b)`
(für eine beliebige ganze Zahl `a` und für eine natürliche Zahl `b`):

$$\begin{array}{rcl} rz(2^{\texttt{b}}) & = & rz(2^{\texttt{b}-1}) + 1 \\ rz(0) & = & 0 \end{array}$$

Es gilt also:

$$rz(2^{\texttt{b}}) = \texttt{b}$$

Zeitbedarf der effizienteren Potenzberechnung (II)

Für die Potenzen von 2 ist rz die Umkehrung der Funktion $\texttt{b} \mapsto 2^{\mathrm{b}}$,
d.h. der Logarithmus zur Basis 2 ($\log_2$).

- Diese Beobachtung liefert keinen präzisen Wert für Zahlen, die keine Potenzen von 2 sind.
- Für große Zahlen ist der Zeitbedarf von `potenz'` viel geringer als der Zeitbedarf von `potenz`.

  Für wachsende Werte von `b` vergrößert sich sehr schnell der Berechnungszeitabstand zwischen `potenz'` und `potenz`.

`b`	`potenz(a,b)`	`potenz'(a,b)`
1	1 Multiplikation	1 Multiplikation
10	10 Multiplikationen	5 Multiplikationen
100	100 Multiplikationen	9 Multiplikationen
1000	1000 Multiplikationen	15 Multiplikationen
⋮	⋮	⋮

2.11.6. Bessere Implementierung der effizienteren Potenzrechnung

Die folgende Implementierung der effizienteren Potenzrechnung ist auch möglich:

```
fun potenz''(a, b) = if b = 0
                       then 1
                       else if gerade(b)
                              then potenz''(quadrat(a), b div 2)
                              else a * potenz''(a, b - 1);
```

Der innere `then`–Fall der Funktion `potenz''` ist „endrekursiv“
(aber der `else`–Fall nicht).

Endrekursion kann vom System effizienter umgesetzt werden ($\leadsto$ Details später)

Der Zeitbedarf von `potenz''` ist der selbe wie von `potenz'`!

3. Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)

- 3.1 Auswertung von Ausdrücken
- 3.2 Auswertung in applikativer und in normaler Reihenfolge
- 3.3 Verzögerte Auswertung
- 3.4 Auswertung der Sonderausdrücke
- 3.5 Funktionale Variablen versus Zustandsvariablen
- 3.6 Funktionale Programmierung versus Imperative Programmierung

3.1. Auswertung von Ausdrücken

- Arten von Ausdrücken
- Die Auswertung von Ausdrücken als Algorithmus
- rekursiver Auswertungsalgorithmus
- Analyse des Auswertungsalgorithmus
- Durchführungsbeispiel
- Substitutionsmodell

3.1.1. Arten von Ausdrücken

Nicht alle Ausdrücke haben denselben Zweck:

- *Konstanten*- und *Funktionsdeklarationen* binden Werte (Konstanten oder Funktionen) an Namen; z.B.:

```
val zwei = 2;

fun quadrat(x: int) = x * x;
```

- *Funktionsanwendungen* wenden Funktionen auf Werte an:

z.B. `quadrat(3 + zwei)`,
`quadrat(3) + quadrat(2)`

Von dem Zweck eines Ausdrucks hängt ab, wie der Ausdruck ausgewertet wird.

Auswertung von Ausdrücken (II)

Wir unterscheiden verschiedene Arten von Ausdrücken:

- *Funktionale Ausdrücke* sind Konstanten und Funktionsanwendungen:
  - *atomar*:

    z.B. `3`, `2`, `true`, `false`, `zwei`,
  - *zusammengesetzt*:

    z.B. `quadrat(3 + 2)`, `3 * 2`, `not false`, `not false <> true`
- *Sonderausdrücke* werden mit den folgenden vordefinierten Konstrukten gebildet:
  - `val` und `fun`, die zur Wertdeklaration dienen
  - `if-then-else`, `case` und das *Pattern Matching* zur Fallunterscheidung
  - die Boole'schen Operatoren `andalso` und `orelse`

[weitere SML-Konstrukte zur Bildung von Sonderausdrücken folgen später]

3.1.2. Die Auswertung von Ausdrücken als Algorithmus

Frage: wie soll die Auswertung von Ausdrücken spezifiziert werden?

Formalisierung notwendig, weil:

- es sich um eine (symbolische) Berechnung handelt,
- die Auswertung von Ausdrücken auf einem Computer durchgeführt werden soll.

$\leadsto$ Auswertung von Ausdrücken als Algorithmus (formal) beschreiben

Anmerkungen:

- also Zurückführung auf schon Bekanntes — nach dem Prinzip des Occam'schen Messers
- Wichtig:
  Durchführung der Auswertung auf einem Computer ist nicht der Hauptgrund für die algorithmische Spezifikation.

  Die Spezifikation dient dem (menschlichen!) Verständnis!

3.1.3. Die Auswertung von Ausdrücken als rekursive Funktion

Skizze des Auswertungsalgorithmus:

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Werte die Teilausdrücke von `A` aus.
2. Wende die Funktion, die sich als Wert des am weitesten links stehenden Teilausdrucks ergibt, auf die Werte an, die sich als Werte aus der Auswertung der restlichen Teilausdrücke ergeben.

$\rightsquigarrow$ rekursiver Algorithmus

Die Auswertung von Ausdrücken als rekursive Funktion (II)

Annahmen des Auswertungsalgorithmus:

- Vereinfachung (aber keine prinzipielle Einschränkung):
  Präfixschreibweise für alle Funktionen,
  z.B. `+(1,2)` statt `1 + 2` (keine korrekte SML-Syntax!)
- Gewisse Funktionen stehen zur Verfügung, ohne dass man sie definieren muss:
  → sogenannte Systemfunktionen (Addition, Multiplikation, . . .)

  Andere Funktionen können definiert werden, z.B.:

  ```
  fun quadrat(x) = *(x, x)
  ```

  Dies ist „syntaktischer Zucker“ für:

  ```
  val quadrat = fn(x) => *(x, x)
  ```

  In der aktuellen Umgebung hat der Name `quadrat` als Wert also die Funktion `fn(x) => *(x, x)`.

Die Auswertung von Ausdrücken als rekursive Funktion (III)

Auswertungsalgorithmus:

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Falls `A` atomar ist, dann:
   (a) Falls `A` vordefiniert ist, liefere den vordefinierten Wert von `A`
   (dieser kann auch eine Systemfunktion sein)
   (b) sonst (der Wert `W` von `A` ist durch eine Gleichung in der Umgebung definiert):
   Liefere `W` als Wert von `A`.
   (`W` kann auch eine Funktion der Form $\mathtt{fn}(\mathtt{F}_1,\ldots,\mathtt{F}_\mathtt{k})$ `=> R` sein.)

Die Auswertung von Ausdrücken als rekursive Funktion (IV)

2. $\mathtt{A}$ ist zusammengesetzt: d.h. $\mathtt{A}$ hat die Form $\mathtt{B}(\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n})$ mit $\mathtt{n} \geq 0$:

   Werte die Teilausdrücke $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ aus.
   Seien $\mathtt{W}_1, \ldots, \mathtt{W}_\mathtt{n}$ die Werte der Teilausdrücke $\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$.

   (a) Falls der Wert von $\mathtt{B}$ eine Systemfunktion ist, dann:
   Wende sie auf $(\mathtt{W}_1, \ldots, \mathtt{W}_\mathtt{n})$ an.
   Liefere den dadurch erhaltenen Wert als Wert von $\mathtt{A}$.

   (b) Falls der Wert von $\mathtt{B}$ eine Funktion der Form $\mathtt{fn}(\mathtt{F}_1, \ldots, \mathtt{F}_\mathtt{n}) => \mathtt{R}$ ist, dann:
   Werte $\mathtt{R}$ in der erweiterten Umgebung aus
   (aktuelle Umgebung erweitert um Gleichungen: $\mathtt{F}_1 = \mathtt{W}_1, \ldots, \mathtt{F}_\mathtt{n} = \mathtt{W}_\mathtt{n}$)
   Liefere den dadurch erhaltenen Wert als Wert von $\mathtt{A}$
   (die Umgebung ist nun wieder die ursprüngliche).

Die Auswertung von Ausdrücken als rekursive Funktion (V)

Ist Auswertung von `B` im Fall 2 nicht umständlich und kompliziert?
Betrachten wir die verschiedenen Unterfälle.

- Der SML-Typ `unit` ermöglicht es, dass $\texttt{n} = \texttt{0}$ im Fall 2 vorkommen kann.

  Die folgende Funktionsdeklaration ist möglich:

  ```
  - fun f() = 1;
  val f = fn : unit -> int
  ```

  `f()` ist die Anwendung der Funktion namens `f` auf `()` (unity).

  Für andere funktionale Programmiersprachen kann im Fall 2 des Auswertungsalgorithmus die Einschränkung $\texttt{n} \geq \texttt{1}$ statt $\texttt{n} \geq \texttt{0}$ notwendig sein.

Die Auswertung von Ausdrücken als rekursive Funktion (VI)

- Sei `A` ein zusammengesetzter Ausdruck A mit `abs` (d.h. `B` ist `abs`).

  `abs` ist (nur!) der Name der einstelligen Systemfunktion (vom Typ `int -> int`) zur Berechnung des Absolutbetrags eines Wertes.

  Also muss `B` zunächst „ausgewertet“ werden;
  die sich als Wert ergebende Systemfunktion kann dann angewendet werden.

- Sei `A` der zusammengesetzte Ausdruck `quadrat(2)` (d.h. `B` ist `quadrat`).

  (mit `quadrat` als dem Namen, der in der aktuellen Umgebung die Funktion `fn(x) => *(x, x)` als Wert hat)

  Die Auswertung von `B` liefert also `fn(x) => *(x, x)`.

  Dann wird der Rumpf `*(x, x)` ausgewertet
  (wofür die Umgebung um die Gleichung `x = 2` erweitert wurde).

Die Auswertung von Ausdrücken als rekursive Funktion (VII)

- Sei `A` der Ausdruck `(if n>0 then quadrat else abs)(5)`
  (d.h. `B` = `if n>0 then quadrat else abs` ist selbst zusammengesetzt).

  Hier muß zunächst `B` ausgewertet werden, um die Funktion zu erhalten, die auf `5` angewendet werden soll.

Der vorgestellte Auswertungsalgorithmus definiert eine Funktion, die

- als Eingabeparameter einen Ausdruck und eine Umgebung erhält und
- als Wert den Wert des Ausdrucks liefert.

Die Auswertungsfunktion ist rekursiv! (siehe Fall 2 und 2(b))

3.1.4. Unvollständigkeit des obigen Algorithmus

Der vorgeschlagene Auswertungsalgorithmus ist nicht vollständig:

- Programmfehler werden nicht behandelt.
- Typen werden nicht berücksichtigt.
- Einige Konstrukte von SML können nicht behandelt werden.
- Die Behandlung der Umgebung ist nur unpräzise erläutert.

Unvollständigkeit des obigen Algorithmus (II)

Offenbar können die folgenden konkreten Fehler auftreten:

- In 1(b) tritt ein Fehler auf, wenn `A` ein nichtdeklarierter Name ist, d.h. für den es keine Gleichung in der Umgebung gibt.
- In 2(a) und 2(b) tritt ein Fehler auf, wenn die Anzahl `n` der aktuellen Parameter $\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ mit der Stelligkeit des Wertes von `B` nicht übereinstimmt.

  z.B. wenn der folgende Ausdruck ausgewertet werden soll:

  ```
  quadrat(3, 5)
  ```

$\leadsto$ Erkennung und Behandlung solcher Fehler sowie Berücksichtigung von Typen verlangt Ergänzung des obigen Auswertungsalgorithmus !

(wird später behandelt)

3.1.5. Zweckmäßigkeit des obigen Algorithmus

Ist es zulässig bzw. sinnvoll, den Algorithmus als rekursive Funktion zu spezifizieren, wo er doch selbst u.a. zur Auswertung von rekursiven Funktionen dienen soll?

Dreht sich eine solche Spezifikation nicht „im Kreis“ ?
(sozusagen wie ein *Perpetuum Mobile* !?!)

Wir bedienen uns hier unserer Intuition von der Auswertung von (rekursiven) Funktionen, um zu verstehen, wie (rekursive) Funktionen ausgewertet werden.

Nicht anders gehen z.B. Linguisten vor, wenn sie in einer Sprache die Grammatik derselben Sprache erläutern.

$\leadsto$ also kein prinzipiell unsinniges Vorgehen !

Zudem können wir uns von der Durchführbarkeit des Auswertungsalgorithmus überzeugen.

3.1.6. Beispiel einer Durchführung des Auswertungsalgorithmus

Seien folgende Deklarationen gegeben:

```
val zwei = 2;
fun quadrat(x) = *(x, x);
```

so dass die Umgebung also aus den beiden folgenden Gleichungen besteht:

```
quadrat = fn(x) => *(x, x)
zwei    = 2
```

Auszuwerten sei der Ausdruck `A`: `quadrat(zwei)`

(im Folgenden beziehen sich die Nummern auf die entsprechenden Fälle des Algorithmus.)

Beispiel . . . (II)

2. `A` ist zusammengesetzt: `B` ist `quadrat`, $\mathtt{A}_1$ ist `zwei`, $\mathtt{n} = 1$.
   Werte den Teilausdruck `B` aus;
   Nebenrechnung, in der `A` der Ausdruck `quadrat` ist:

   1. `A` ist atomar.
   (b) `A` ist nicht vordefiniert.
   Als Wert von `quadrat` wird aus der Umgebung `fn(x) => *(x,x)` geliefert.

   Ende der Nebenrechnung; Wert von `B` ist `fn(x) => *(x,x)`.
   Werte den Teilausdruck $\mathtt{A}_1$ aus;
   Nebenrechnung, in der `A` der Ausdruck `zwei` ist:

   1. `A` ist atomar.
   (b) `A` ist nicht vordefiniert.
   Als Wert von `zwei` wird aus der Umgebung die natürliche Zahl 2 geliefert.

   Ende der Nebenrechnung; Wert von $\mathtt{A}_1$ ist 2.

Beispiel . . . (III)

(b) Der Wert von `B` ist keine Systemfunktion, sondern eine Funktion `fn(x) => *(x,x)`.
Die erweiterte Umgebung besteht aus der aktuellen Umgebung und der zusätzlichen Gleichung `x`=2.
Werte `*(x,x)` in dieser erweiterten Umgebung aus;
Nebenrechnung, in der `A` der Ausdruck `*(x,x)` ist.

2. `A` ist zusammengesetzt,
   `B` ist `*`, $\mathtt{A}_1$ ist `x`, $\mathtt{A}_2$ ist `x`, $\mathtt{n} = 2$.
   Werte den Teilausdruck `B` aus;
   Nebenrechnung, in der `A` der Ausdruck `*` ist

   1. `A` ist atomar.

   (a) `A` ist vordefiniert.
   Wert von `*` ist die zweistellige Multiplikationsfunktion, also eine Systemfunktion.

   Ende der Nebenrechnung; Wert von `B` ist die Multiplikationsfunktion.

Beispiel . . . (IV)

Werte den Teilausdruck $\mathtt{A}_1$ aus;

Nebenrechnung, in der `A` der Ausdruck `x` ist:

1. `A` ist atomar.

(b) `A` ist nicht vordefiniert.
Als Wert von `x` wird aus der (erweiterten) Umgebung die natürliche Zahl 2 geliefert.

Ende der Nebenrechnung; Wert von $\mathtt{A}_1$ ist 2

Genauso: Wert von $\mathtt{A}_2$ ist 2

(a) Der Wert von `B` ist eine Systemfunktion, nämlich die Multiplikationsfunktion.
Wende sie auf `(2, 2)` an
Der dadurch erhaltene Wert ist `4`.

Ende der Nebenrechnung, Wert von `*(x,x)` ist `4`.

Der dadurch erhaltene Wert von `A` ist also `4`

(die Umgebung ist nun wieder die ursprüngliche, ohne die Gleichung `x=2`).

Anmerkungen zum Auswertungsbeispiel

In Implementierungen wird üblicherweise nicht die textuelle Repräsentation der vom Benutzer definierten Funktionen (wie etwa `fn(x) => *(x,x)` mit dem Namen `quadrat`) verwendet.

Statt dessen wird ein Verweis (Zeiger, Speicheradresse) auf einen „Funktionsdeskriptor“ verwendet, der folgendes enthält:

- die formalen Parameter,
- die Typen und
- den Funktionsrumpf.

Als „Wert“ von `quadrat` wird dann intern die Speicheradresse dieses Funktionsdeskriptors geliefert.

Einen Teil dieser Angaben benutzt das SML-System, um z.B. die Ausgabe `fn: int -> int` für den Typ der Funktion `quadrat` zu ermitteln.

Anmerkungen zum Auswertungsbeispiel (II)

Das Symbol `fn` (oder die Speicheradresse für einen Funktionsdeskriptor) stellt also nur einen Vermerk dar.

- Eine solche Vermerktechnik ist notwendig, weil ein Programm nur Bezeichner (also *Symbole*) bearbeitet.
- Die Begriffe „Funktion“ und „Funktionsanwendung“ werden erst durch einen Auswertungsalgorithmus verwirklicht – durch symbolische Berechnungen!

Computer führen Berechnungen penibel Schritt für Schritt nach einem vorgebenen präzise festgelegten Verfahren durch:

$\leadsto$ Fehler oder Ungenauigkeiten im Programm führen damit fast zwangsläufig zu falschen Ergebnissen !

3.1.7. Substitutionsmodell

Der Auswertungsalgorithmus kann auf einer höheren Abstraktionsebene wie folgt erläutert werden:

> Um einen Ausdruck `A` auszuwerten, werden alle Teilausdrücke von `A` durch ihre Definitionen ersetzt, wobei
>
> - die formalen Parameter von Funktionsdefinitionen durch die aktuellen Parameter der Funktionsanwendung ersetzt werden,
> - vordefinierte Konstanten gemäß ihrer Definition durch ihre Werte ersetzt werden und
> - vordefinierte Funktionen gemäß ihrer Definition ersetzt werden.

Diese sehr abstrakte Beschreibung der Auswertung wird *Substitutionsmodell* genannt.

Das Substitutionsmodell ist viel weniger präzise als der vorangehende Algorithmus — insbesondere wird die Reihenfolge der Auswertungsschritte offen gelassen.

3.2. Auswertung in applikativer und in normaler Reihenfolge

- Auswertungsreihenfolge
- applikative Reihenfolge
- normale Reihenfolge

3.2.1. Auswertungsreihenfolge

Die Reihenfolge der Auswertung der Teilausdrücke eines zusammengesetzten Ausdruck wird

- im Auswertungsalgorithmus festgelegt,
- im Substitutionsmodell *nicht* festgelegt.

Frage: Spielt die Auswertungsreihenfolge eine Rolle ?

Betrachten wir z.B. den Ausdruck `quadrat(2 + 1)`.

Der Auswertungsalgorithmus geht wie folgt vor:

- Da es ein zusammengesetzter Asudruck ist, werden die Teilausdrücke `quadrat` ($\leadsto$ Verweis `fn`) und `2 + 1` ($\leadsto$ Wert `3`) ausgewertet.
- Dann erst wird die zu `quadrat` gehörende Funktion auf den aktuellen Parameter `3` angewendet (was zu `3 * 3` führt).

Auswertungsreihenfolge (II)

Der Auswertungsalgorithmus wertet also

- zunächst die aktuellen Parameter (Operanden) einer Funktionsanwendung aus,
- bevor er die Funktion auf die Werte dieser Parameter anwendet.

Achtung:

Der Auswertungsalgorithmus legt aber nicht fest, in welcher Reihenfolge die Teilausdrücke $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ ausgewertet werden (z.B. von links nach rechts, von rechts nach links, . . .)

Auswertungsreihenfolge (III)

Auswertung des Ausdrucks `quadrat(quadrat(2))`:

mit dem Auswertungsalgorithmus:

0. `quadrat(quadrat(2))`
1. `quadrat(2 * 2)`
2. `quadrat(4)`
3. `4 * 4`
4. `16`

Eine alternative Reihenfolge:

0. `quadrat(quadrat(2))`
1. `quadrat(2) * quadrat(2)`
2. `(2 * 2) * (2 * 2)`
3. `4 * 4`
4. `16`

$\rightsquigarrow$ anderer Ablauf, aber (hier!) dasselbe Ergebnis !

3.2.2. Auswertung in applikativer Reihenfolge

Die Auswertung entsprechend dem vorgestellten Auswertungsalgorithmus — also Parameterauswertung vor Funktionsanwendung — hat die folgenden Namen, die alle dasselbe bezeichnen:

- Auswertung in applikativer Reihenfolge
- Inside-out–Auswertung
- Call-by-value–Auswertung
- strikte Auswertung

Auswertung in applikativer Reihenfolge (II)

Anmerkung:

„Call-by-value“ hat für nicht-funktionale Programmiersprachen eine andere Bedeutung:

- Es bezeichnet Prozedur-/Funktionsaufrufe, bei denen die Werte der Aufrufparameter an die Prozedur/Funktion übergeben werden (als Kopien), aber nicht die Speicheradressen der Originalwerte.

- Als Alternative dazu gibt es „Call-by-reference“, wobei nur die Speicheradressen als Referenzen auf die Werte übergeben werden.

3.2.3. Auswertung in normaler Reihenfolge

Die Auswertung in der anderen Reihenfolge — also Funktionsanwendung vor Parameterauswertung — hat die folgenden Namen:

- Auswertung in normaler Reihenfolge
- Outside-in–Auswertung
- Call-by-name–Auswertung

Anerkung:
In einigen alten Programmiersprachen bezeichnete *call by name* die Weitergabe eines Bezeichners ohne Festlegung eines Wertes noch einer Speicheradresse, also eine rein textuelle Zeichenfolgenersetzung.

Auswertung in normaler Reihenfolge (II)

Wenn alle Funktionsanwendungen terminieren, liefern beide Auswertungsreihenfolgen (applikativ und normal) immer dasselbe Ergebnis.

Bei nichtterminierenden Funktionsanwendungen ist dies nicht immer der Fall.

Beispiel:
Funktion `null`, die für jede ganze Zahl den Wert `0` liefert,
und eine nichtterminierende Funktion `f`:

```
fun null(x : int) = 0;
fun f(x : int) = f(x + 1);
```

Die Auswertung des Ausdrucks `null(f(1))`

- in normaler Reihenfolge liefert den Wert `0`,
- in applikativer Reihenfolge terminiert nicht.

3.2.4. Vorteil der applikativen Reihenfolge gegenüber der normalen Reihenfolge

Am Beispiel des Ausdrucks `quadrat(quadrat(2))` erkennt man folgenden Vorteil der applikativen Reigenfolge:

- Ein Parameter (oder Operand) wird bei einer Auswertung in applikativer Reihenfolge nur einmal ausgewertet,

- bei einer Auswertung in normaler Reihenfolge jedoch unter Umständen mehrmals.

$\leadsto$ Vermeidung redundanter Auswertungen

3.3. Verzögerte Auswertung

Die applikative Auswertungsreihenfolge ist aber nicht immer die vorteilhafteste.

Betrachten wir hierzu noch einmal die einstellige konstante Funktion `null`:

```
fun null(x : int) = 0;
```

3.3.1. Vorteil der normalen Reihenfolge gegenüber der applikativen Reihenfolge

Vergleich der Auswertung des Ausdrucks `null(quadrat(quadrat(quadrat(2))))`:

in applikativer Reihenfolge:

```
0. null(quadrat(quadrat(quadrat(2))))
1. null(quadrat(quadrat(2 * 2)))
2. null(quadrat(quadrat(4)))
3. null(quadrat(4 * 4))
4. null(quadrat(16))
5. null(16 * 16)
6. null(256)
7. 0
```

in normaler Reihenfolge:

```
0. null(quadrat(quadrat(quadrat(2))))
1. 0
```

Vorteil der normalen Reihenfolge . . . (II)

Die Durchführung der Auswertung der Parameter (oder Operanden) vor der Funktionsanwendung ist nur dann von Vorteil, wenn alle Parameter von der Funktion „verwendet“ werden.

- Die normale Reihenfolge vermeidet die Auswertung von Parametern, wenn diese danach nicht mehr verwendet werden.
- Die applikative Reihenfolge kann also zu überflüssigen Berechnungen führen.

Die verzögerte Auswertung

Die verzögerte Auswertung (*lazy evaluation*, auch *Call-by-need*) bringt Aspekte der Auswertung in applikativer Reihenfolge und in normaler Reihenfolge zusammen.

Grundidee der verzögerten Auswertung:

- Wie bei der Auswertung in normaler Reihenfolge führt die verzögerte Auswertung die Funktionsanwendung vor der Auswertung der Parameter (oder Operanden) durch.
- Bei der Funktionsanwendung werden aber alle bis auf ein Vorkommen eines Parameters durch einen Verweis auf ein einziges dieser Vorkommen ersetzt, damit dieser Parameter nicht mehrmals ausgewertet wird.

Die verzögerte Auswertung (II)

Auswertung des Ausdrucks `quadrat(quadrat(2))`:

```
quadrat(quadrat(2))

quadrat(2)  *  .

(2  *  .)  *  .

4    *    .

16
```

Die Verweise („Pfeile“) verhindern, dass mehrfach vorkommende Teilausdrücke mehrfach ausgewertet werden.

Die verzögerte Auswertung (III)

Die verzögerte Auswertung von `null(quadrat(quadrat(quadrat(2))))`:

```
null(quadrat(quadrat(quadrat(2))))

0
```

Analyse:

- Wie die Auswertung in applikativer Reihenfolge vermeidet die verzögerte Auswertung die mehrfache Auswertung von Parametern.

- Wie die Auswertung in normaler Reihenfolge vermeidet die verzögerte Auswertung die Auswertung von Parametern, die zur Auswertung einer Funktionsanwendung nicht notwendig sind.

Die verzögerte Auswertung (IV)

$\Rightarrow$ Die verzögerte Auswertung hat also die Vorteile der beiden Grundansätze zur Auswertung und vermeidet deren Nachteile.

Die verzögerte Auswertung scheint also die beste Auswertungsform zu sein.

Anmerkung:

Das Substitutionsmodell ist nicht zur Formalisierung der verzögerten Auswertung geeignet, weil die verzögerte Auswertung auf einer Datenstruktur (Zeiger bzw. Graph) beruht, die im Substitutionsmodell nicht vorhanden ist.

3.3.2. Verweis

Was ist aber ein Verweis (auch Zeiger, *pointer* oder Referenz genannt)?

- in einer abstrakten Sichtweise:

  $\longrightarrow$ vergleichbar mit einer gerichteten Kante in einem Graphen

  Folgt man einen Verweis (einer gerichteten Kante) gelangt man zu den referenzierten Objekt (Ausdruck, Wert; Knoten im Graph).

- in einer implementierungsnahen Sichtweise:

  $\longrightarrow$ entspricht einer (Speicher-) Adresse

  Die Adresse gibt an, wo das referenzierte Objekt (im Speicher) zu finden ist

Im Grunde liegt der Verwendung von Verweisen ein Berechnungsmodell zugrunde:

Nach diesem Berechnungsmodell wird bei der Auswertung (von Ausdrücken) der Ausdruck, auf den ein Verweis zeigt, an der Stelle des Ursprungs des Verweises eingefügt.

3.3.3. Auswertungsreihenfolge von SML

SML verwendet die Auswertung in applikativer Reihenfolge, da die die verzögerte Auswertung auch Nachteile hat:

- Die verzögerte Auswertung verlangt eine ziemlich komplizierte — also zeitaufwendige — Verwaltung von Verweisen (Zeigern).
- Die verzögerte Auswertung lässt imperative (oder prozedurale) Befehle wie Schreibbefehle nur schwer zu, weil sie wie die Auswertung in normaler Reihenfolge den Zeitpunkt der Ausführung solcher Befehle schwer vorhersehbar macht.

  (Dies wird bei Einführung der imperativen Befehle von SML näher betrachtet.)
- In manchen Fällen verlangt die verzögerte Auswertung viel mehr Speicherplatz als die Auswertung in applikativer Reihenfolge.

3.4. Auswertung der Sonderausdrücke

- Wertdeklarationen
- `if-then-else`
- Pattern Matching / `case-of`
- Boole'sche Operatoren `andalso` und `orelse`
- Infixoperatoren und Präzedenzen
- erweiterter Auswertungsalgortihmus

3.4.1. Wertdeklarationen (`val` und `fun`)

- (einfache) Wertdeklaration `val N = A`:

  Es wird die Gleichung `N = A` in die Umgebung eingefügt.

- Funktionsdeklaration der Form `val N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt.
  (also wie oben)

- rekursive Funktionsdeklaration der Form `val rec N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt.

  Zusätzlich werden Vorkehrungen getroffen, dass die Umgebung von `A` auch diese Gleichung für `N` enthält.

- Funktionsdeklaration der Form `fun N P = A`:

  wie bei der rekursiven Funktionsdeklaration (vorheriger Fall)

Wertdeklarationen (II)

Wird eine Gleichung zu der Umgebung hinzugefügt, so wird sie auf Korrektheit überprüft:

- Eine Deklaration wie z.B. `val zwei = zwei` wird dabei abgelehnt.

  Die verwendeten Namen im Rumpf (hier: `zwei`) müssen für die Auswertung einer (einfachen) Wertdeklaration bekannt sein.

Anmerkung:
Gleichungen aus Deklarationen werden in der Praxis in einen anderen Formalismus übersetzt, bevor sie in die Umgebung hinzugefügt werden.
Diese Übersetzung ist aber zum Verständnis der Auswertung unwesentlich.

3.4.2. if-then-else

- Die Schreibweise eines Ausdrucks `if A1 then A2 else A3` entspricht nicht der üblichen Schreibweise von Funktionen.

  Diese Schreibweise ist sogenannter „syntaktischer Zucker“
  (syntaktische Verschönerung, die die Lesbarkeit erhöht).

- In Präfixschreibweise könnte man auch `if-then-else(A1, A2, A3)` schreiben.

  (dies ist in SML aber keine korrekte Schreibweise, wir werden es aber gelegentlich zur Veranschaulichung verwenden)

if-then-else (II)

Lässt sich die Auswertung von `if-then-else`–Ausdrücken wie normale Funktionsanwendungen behandeln?

- Das würde Auswertung in applikativer Reihenfolge bedeuten:

  Zur Auswertung eines Ausdrucks `if A1 then A2 else A3` müssten zunächst alle drei Teilausdrücke `A1`, `A2` und `A3` ausgewertet werden.

Dieser Ansatz ergibt aber wenig Sinn, wie wir am Beispiel der Fakultätsfunktion feststellen können:

$$
(*) \quad \begin{array}{ll} \mathrm{n}! = 1 & \text{falls } \mathrm{n} = 0 \\ \mathrm{n} * (\mathrm{n} - 1)! & \text{andernfalls} \end{array}
$$

Die Fakultätsfunktion ist auf den natürlichen Zahlen definiert und damit eine partielle Funktion auf den ganzen Zahlen.

if-then-else (III)

In SML kann die Fakultätsfunktion wie folgt implementiert werden:

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

oder, in der Präfixschreibweise,

(***) `fun fak(n) = if_then_else( n=0, 1, n*fak(n-1) );`

Da die Funktion nicht total ist, haben einige korrekt gebildete Ausdrücke keinen Wert.

Dies gilt z.B. für `fak(~1)`, da die entsprechende Auswertung nicht terminiert.

Die Auswertung von `if A1 then A2 else A3` in applikativer Reihenfolge würde bedeutet:

- Werte zunächst alle drei Teilausdrücke `A1`, `A2` und `A3` aus.
- Wende anschließend die (System-) Funktion `if-then-else` darauf an.

if-then-else (IV)

Diese applikative Auswertungsreihenfolge führt zu Problemen, wie die folgende Auswertung von `fak(0)` zeigt (hier in Präfixschreibweise):

```
fak(0)
if_then_else(0=0, 1, 0*fak(0-1) )
if_then_else(true,1, 0*fak(~1) )
if_then_else(true,1, 0*if_then_else(~1=0 ,1, ~1*fak(~1-1) ) )
if_then_else(true,1, 0*if_then_else(false,1, ~1*fak(~2) ) )
if_then_else(true,1, 0*if_then_else(false,1, ~1*if_then_else(...)))
...
```

$\leadsto$ Diese Auswertung terminiert nicht!

if-then-else (V)

Das Grundprinzip der applikativen Reihenfolge, zunächst immer alle Teilausdrücke auszuwerten, führt zur Nichtterminierung bei rekursiv definierten Funktionen werden.

Daher muss ein Ausdruck `if A1 then A2 else A3` als Sonderausdruck mit eigenem Auswertungsmechanismus behandelt werden:

Werte von den drei Teilausdrücken `A1`, `A2`, `A3` zuerst nur `A1` aus.

- Hat `A1` den Wert `true`,
  dann (und nur dann) wird `A2` ausgewertet (und `A3` wird nicht ausgewertet).
  Der Wert von `A2` wird als Wert des gesamten Ausdrucks geliefert.
- Hat `A1` den Wert `false`,
  dann (und nur dann) wird `A3` ausgewertet (und `A2` wird nicht ausgewertet).
  Der Wert von `A3` wird als Wert des gesamten Ausdrucks geliefert.

if-then-else (VI)

- Dies ist ein sehr präziser (und einfacher) Algorithmus.
- SML wertet `if-then-else`–Ausdrücke in dieser Weise aus.

  (wie die meisten Programmiersprachen)
- Ohne diese Sonderbehandlung wären rekursive Definitionen nicht möglich!

  (obwohl sie — wie die Fakultätsfunktion zeigt — benötigt werden)

3.4.3. Pattern Matching

„Pattern Matching“ haben wir in Beispielen bereits kennengelernt:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
              | 5 => true
              | 6 => true
              | 7 => true
              | 8 => true
              | 9 => true
              | _ => false;
```

```
(fn true => E1 | false => E2)
```

Pattern Matching (II)

SML bietet das folgende Konstrukt an, das `if_then_else` in gewissem Sinn verallgemeinert:

```
case A of A1 => B1
        | A2 => B2
          ...
        | An => Bn
```

Die Auswertung eines solchen Ausdrucks geschieht wie folgt.

- Zunächst wird nur der Ausdruck `A` ausgewertet.
- Der Wert von `A` wird dann nacheinander mit den Mustern `A1`, . . . , `An` „gematcht“.
  Ist `Ai` das erste Pattern, das mit dem Wert von `A` „matcht“, so wird `Bi` ausgewertet und dessen Wert als Wert des `case`-Ausdrucks geliefert.

[Das Pattern Matching in Deklarationen wird ähnlich ausgewertet.]

Pattern Matching (III)

Das `case`-Konstrukt könnte als Grundlage zur Auswertung von anderen Sonderausdrücken dienen.

Zum Beispiel könnte ein Ausdruck `if A1 then A2 else A3` auch so ausgewertet werden:

> Werte von den drei Teilausdrücken `A1`, `A2`, `A3` zunächst gar keinen aus.
> Konstruiere einen neuen Ausdruck:
> ```
> case A1 of true => A2 | false => A3
> ```
> Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des `if-then-else`–Ausdrucks.

Andere Sonderausdrücke, können in ähnlicher Weise auf `case` zurückgeführt werden.

In der Praxis werden Sonderausdrücke allerdings selten auf andere zurückgeführt, sondern durch maßgeschneiderte Auswertungsalgorithmen behandelt.

3.4.4. Die Boole'schen Operatoren `andalso` und `orelse`

Boole'sche Konjunktionen $\mathrm{A}_1 \wedge \mathrm{A}_2$ und Disjunktionen $\mathrm{A}_1 \vee \mathrm{A}_2$ können prinzipiell in zwei verschiedenen Weisen ausgewertet werden:

1. Auswertungsmöglichkeit:

Die Teilausdrücke A_1 und A_2 werden zunächst ausgewertet und der Wert der Konjunktion oder Disjunktion wird entsprechend der folgenden Wahrheitstabelle aus den Werten von A_1 und von A_2 ermittelt:

A_1	A_2	$\mathrm{A}_1 \wedge \mathrm{A}_2$	$\mathrm{A}_1 \vee \mathrm{A}_2$
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Die Boole'schen Operatoren `andalso` und `orelse` (II)

2. Auswertungsmöglichkeit:

Konjunktion ($\wedge$):

Zunächst wird nur der erste Teilausdruck A_1 ausgewertet. Ist der Wert von A_1 true, so wird auch A_2 ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert. Ist der Wert von A_1 false, so wird false als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert (und A_2 wird nicht ausgewertet).

Disjunktion ($\vee$):

Zunächst wird nur der erste Teilausdruck A_1 ausgewertet. Ist der Wert von A_1 true, so wird true als Wert des Ausdrucks $A_1 \vee A_2$ geliefert (und A_2 wird nicht ausgewertet). Ist der Wert von A_1 false, so wird A_2 ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \vee A_2$ geliefert.

Die Boole'schen Operatoren `andalso` und `orelse` (III)

Der zweite Ansatz kann auf andere Sonderausdrücke zurückgeführt werden.

Die Auswertung eines Ausdrucks der Form $\mathrm{A}_1 \wedge \mathrm{A}_2$ könnte so aussehen:

> Werte von den Teilausdrücken A_1, A_2 zunächst gar keinen aus. Konstruiere einen neuen Ausdruck
>
> **if** $\mathtt{A}_1$ **then** $\mathtt{A}_2$ **else false**
>
> Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des ursprünglichen Ausdrucks.

Die Boole'schen Operatoren `andalso` und `orelse` (IV)

Welcher der beiden Auswertungsansätze ist besser geeignet?

- Der erste Ansatz entspricht der üblichen logischen Sichtweise.

  Die Kommutativität der Operatoren $\wedge$ and $\vee$ bleibt auch durch die Auswertung erhalten (d.h., $\mathrm{A1} \wedge \mathrm{A2}$ wird in jedem Fall mit demselben Ergebnis ausgewertet wie $\mathrm{A2} \wedge \mathrm{A1}$).
- Der zweite Ansatz vermeidet die Auswertung des zweiten Operanden, wenn der erste Operand bereits das Ergebnis bestimmt.

SML wählt (wie viele andere Programmiersprachen) den 2. Ansatz.

Die SML-Bezeichnungen „`andalso`“ und „`orelse`“ statt „`and`“ und „`or`“ sollen unterstreichen, dass diese Operatoren auf die zweite Weise ausgewertet werden.

Die Boole'schen Operatoren `andalso` und `orelse` (V)

Wie kann man herausfinden, wie eine Programmiersprache die Boole'schen Operationen auswertet?

In SML reicht dazu der folgende kleine Test (hier für die Konjunktion):

```
fun endlose_berechnung(n) : bool = endlose_berechnung(n + 1);

false andalso endlose_berechnung(0);
```

Terminiert er, dann wertet die Programmiersprache die Konjunktion nach dem 2. Ansatz aus, andernfalls nach dem 1. Ansatz.

3.4.5. Infixoperator-Deklarationen und Präzedenzen

Wenn man die Boole'schen Operatoren selbst definiert, möchte man sie vermutlich gern in Infixschreibweise verwenden.

Zu diesem Zweck bietet SML sogenannte Infixoperator-Deklarationen:

```
infix /\;
fun A1 /\ A2 = if A1 then A2 else false;

infix \/;
fun A1 \/ A2 = if A1 then true else A2;
```

Eine Infixoperator-Deklaration kann mit einer Präzedenz von 0 bis 9 ergänzt werden:

```
infix 6 /\;
```

Fehlt die Präzedenz, dann gilt der voreingestellte (default) Wert 0. Je größer die Präzedenz, desto stärker bindet der Operator.

Infixoperator-Deklarationen und Präzedenzen (II)

Die Deklaration `infix n Op;` definiert den Operator `Op` als linksassoziativ.

Die Deklaration `infixr n Op;` definiert den Operator `Op` als rechtsassoziativ.

Beispiel:

Die Funktion `potenz'` (Abschnitt 2.11) kann wie folgt als rechtsassoziativer Infixoperator deklariert werden:

```
infixr 8 **;
fun a ** b = if b = 0 then 1
                      else if gerade(b)
                              then quadrat(a ** (b div 2))
                              else a * (a ** (b - 1));
```

3.4.6. Erweiterter Auswertungsalgorithmus mit Behandlung von Sonderausdrücken

Für die verschiedenen Sonderausdrücke muss der ursprüngliche Auswertungsalgorithmus erweitert werden:

- Für jeden Sonderausdruck wird ein „Sonderalgorithmus“ als Wert des entsprechenden Bezeichners angenommen.

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. `A` ist atomar:

   (a) `A` ist vordefiniert: liefere den vordefinierten Wert von A.
   (Dieser kann auch ein Sonderalgorithmus oder eine Systemfunktion sein).

   (b) Andernfalls (der Wert von `A` ist in der Umgebung durch die Gleichung $\mathtt{A} = \mathtt{W}$ definiert):

   Liefere `W` als Wert von `A`.
   (wobei `W` auch eine Funktion $\mathtt{fn}(\mathtt{F}_1, \ldots, \mathtt{F}_\mathtt{k})$ `=> R` sein kann)

Erweiterter Auswertungsalgorithmus . . . (II)

2. `A` ist zusammengesetzt und hat die Form $\texttt{B}(\texttt{A}_1, \ldots, \texttt{A}_\texttt{n})$ mit $\texttt{n} \geq \texttt{0}$:

   Werte den Teilausdruck `B` aus.

   (a) Der Wert von `B` ist ein Sonderalgorithmus:

   Wende ihn auf $\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}$ an. Liefere den Ergebniswert als Wert von `A`.

   (b) Der Wert von `B` ist kein Sonderalgorithmus:

   Werte die Teilausdrücke $\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}$ aus; seien $\texttt{W}_1, \ldots, \texttt{W}_\texttt{n}$ die entsprechenden Werte.

   i. Der Wert von `B` ist eine Systemfunktion:

   Wende sie auf $(\texttt{W}_1, \ldots, \texttt{W}_\texttt{n})$ an. Liefere den Ergebniswert als Wert von `A`.

   ii. Der Wert von `B` ist eine Funktion der Form $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_\texttt{n})$ `=> R`:

   Werte `R` in der erweiterten Umgebung aus, die aus der aktuellen Umgebung und den zusätzlichen Gleichungen $\texttt{F}_1 = \texttt{W}_1$, . . . , $\texttt{F}_\texttt{n} = \texttt{W}_\texttt{n}$ besteht.
   Liefere den Ergebniswert als Wert von `A`.
   (die Umgebung ist nun wieder die ursprüngliche)

3.5. Funktionale Variablen versus Zustandsvariablen

Alle Variablen (Namen, Bezeichner), die wir bislang verwendet haben, waren ausschließlich

- funktionale Variablen (auch „logische Variablen“ genannt)

Viele Programmiersprachen verwenden eine andere Art von Variablen:

- „Zustandsvariablen“

SML bietet sowohl funktionale wie Zustandsvariablen an.

3.5.1. Funktionale Variablen

Hauptmerkmal funktionaler Variablen:

- Bindung *eines* Werts an eine funktionale Variable (Bindung eines Werts an einen Namen), die nicht verändert werden kann.

  Eine funktionale Variable kann also als eine benannte Speicherzelle (bzw. benannter Speicherbereich) zur Speicherung eines einzigen Wertes angesehen werden.

```
- val z = 2;
val z = 2 : int

- fun quadrat(x : int) = x * x;

- val z = 5;
val z = 5 : int
```

Die zweite Deklaration einer Variablen namens `z` ist eine „Wiederdeklaration eines Namens“.

Funktionale Variablen (II)

Wiederdeklaration:

- Die erste Speicherzelle namens `z` mit dem Wert `2` bleibt erhalten.
- Eine zweite Speicherzelle erhält den Namen `z` und als Inhalt den Wert `5`.

Namenskonflikte nach Wiederdeklaration einer Variable werden durch die folgende einfache Regel gelöst, deren Wirkung auf Variablen *Überschatten* heißt:

> Nach einer Wiederdeklaration einer Variablen mit Namen `N` gilt der Name `N` nur noch für die zuletzt deklarierte Variable mit Namen `N`.

Welche Auswirkungen hat dies auf Deklarationen, die diesen Namen verwenden?

Bei Programmiersprachen mit statischer (oder lexikalischer) Bindung, werden solche Deklarationen nicht beeinflusst.

Funktionale Variablen (III)

Die Konzepte

- Überschatten und
- statische (oder lexikalische) Bindung

machen eine funktionale Programmiersprache *referenztransparent*:

Der Wert eines Ausdrucks (dieser Sprache) wird nicht verändert, wenn „das Gleiche durch das Gleiche“, d.h. ein Teilausdruck durch seine Definition, ersetzt wird.

Anders ausgedrückt:

Eine Sprache ist genau dann referenztransparent, wenn syntaktisch gleiche Ausdrücke in der gleichen Umgebung stets auch gleiche Werte haben.

3.5.2. Zustandsvariablen

Eine andere Art von Variablen: *Zustandsvariablen*

grundlegende Idee:

- benannten Speicherzelle (oder eines Speicherbereiches) mit *veränderbarem* Inhalt
- *oder anschaulich*: etikettierte (beschriftete) Schublade
  Der Name entspricht dem Etikett, der Schubladeninhalt lässt sich verändern.

Anwendungsbeispiel für Zustandsvariablen: ein Bankkonto

- Eine Zustandsvariable `kto` habe als Inhalt zunächst den Wert 100 (EURO)
- Anwendung von Operationen, um z.B.
  den Kontostand um 25 EURO auf 125 EURO zu erhöhen (`einzahlen 25`),
  dann um 55 EURO auf 70 EURO zu verringern (`abheben 55`).

Mit funktionalen Variablen ist eine solche Veränderung von Inhalten nicht möglich.

Zustand, Zustandsänderung und Zuweisung

- Da Zustandsvariablen veränderbar sind, hängen ihre Inhalte vom jeweiligen Zeitpunkt ab.

- Ein *Zustand* einer Menge von Zustandsvariablen, oder allgemeiner eines Systems, das auf Zustandsvariablen beruht, ist eine Menge von Bindungen (Variablenname, Variableninhalt), die zu einem Zeitpunkt gelten.

- Eine *Zustandsänderung* ist eine Veränderung des Inhalts einer Zustandsvariablen, also auch des Zustandes.

- Zustandsänderungen erfordern eine Operation, mit der der Inhalt einer Zustandsvariablen verändert werden kann.

  Diese Operation heißt üblicherweise *Zuweisung*.

- In vielen Programmiersprachen wird sie „`:=`“ notiert, auch in SML.

Zwei Arten von Zustandsvariablen

Der Begriff der Zustandsvariable ist ziemlich natürlich und einfach.

Da Operationen auf Zustandsvariablen in zwei verschiedenen Weisen ausgedrückt werden können, müssen wir aber genauer unterscheiden:

- *Zustandsvariablen mit expliziter Dereferenzierung*
- *Zustandsvariablen ohne explizite Dereferenzierung*

Zustandsvariablen mit expliziter Dereferenzierung werden auch *Referenzen* genannt.

Referenzen und Dereferenzierung

Bei Zustandsvariablen mit expliziter Dereferenzierung steht der Variablennamen stets für eine (symbolische) Speicheradresse (also für eine Art Schubladenetikett).

$\rightsquigarrow$ Sprachkonstrukt notwendig, um den Inhalt der Schublade zu bezeichnen

- In SML sind Zustandsvariablen Referenzen.
- Der Operator „`!`“ bezeichnet die Operation der *Dereferenzierung*.
- Ist `v` eine SML-Referenz, so bezeichnet `!v` den Inhalt der Referenz.

In SML wird `kto` als Referenz mit anfänglichem Wert von `100` wie folgt deklariert:

```
- val kto = ref 100;
```

Den Kontostand kann man dann so erhalten:

```
- !kto;
val it = 100 : int
```

Referenzen und Dereferenzierung (II)

Achtung:

Wir unterscheiden hier bewusst zwischen den Bezeichnungen „Wert“ und „Inhalt“.

- Der Ausdruck `!kto` hat den Wert `100`.
- Der Ausdruck `kto` dagegen hat *nicht* den Wert `100`.

  Der Wert von `kto` ist eine (symbolische) Referenz auf eine Speicherzelle, die den Inhalt `100` hat.
- Der Wert von `kto` ist das „Etikett der Schublade“;
  der Wert von `!kto` ist das, was in der Schublade drin ist.
- Dies äußert sich auch in den Typen:

  `!kto` hat den Typ `int`, aber `kto` hat den Typ `int ref`.

Referenzen und Dereferenzierung (III)

Unter Verwendung des Dereferenzierungsoperators „`!`“ und der Zuweisung „`:=`“ (ausgesprochen: *becomes*) wird der Konto(zu)stand wie folgt verändert:

```
- kto := !kto + 25;
val it = () : unit

- !kto;
val it = 125 : int

- kto := !kto - 55;
val it = () : unit

- !kto;
val it = 70 : int
```

Sequenzierung

Da der Zeitpunkt einer Zustandsveränderung für den Zustand von Belang ist, muss die Reihenfolge von Zustandsveränderungen festgelegt werden.

Das Konstrukt „`;`“ drückt eine Reihenfolge oder *Sequenz* aus.
(in SML wie in vielen anderen Programmiersprachen)

Der Ausdruck `(A1; A2)` wird so ausgewertet:

- Zuerst wird `A1` ausgewertet, aber sein Wert wird ignoriert.
- Danach wird `A2` ausgewertet; der Wert Wert von `A2` wird als Wert des Ausdrucks `(A1; A2)` geliefert.

Zwischen den Klammern können beliebig viele durch „`;`“ getrennte Ausdrücke stehen, die dann in der gegebenen Reihenfolge ausgewertet werden, wobei nur der Wert des letzten zurückgeliefert wird.

Zustandsvariablen ohne explizite Dereferenzierung

In den frühen Jahren der Informatik haben sich Zustandsvariablen etabliert, die keiner expliziten Dereferenzierung bedürfen:

Je nach Kontext, in dem ein Variablenname `v` vorkommt, steht er mal für eine Referenz (d.h. eine symbolische Speicheradresse) mal für den Wert, der Inhalt der Speicheradresse `v` ist.

Ist `kto` eine solche Zustandsvariable, so wird der Konto(zu)stand wie folgt verändert:

```
kto := kto + 25;
kto := kto - 55;
```

- Links vom Zuweisungsoperator „`:=`“ bezeichnet `kto` hier eine Speicheradresse,
- rechts einen Wert (implizite Dereferenzierung).

Viele Programmiersprachen verwenden solche Zustandsvariablen;
z.B. Scheme, Pascal, Java, ...

Gleichheit

Die Gleichheit zwischen Zustandsvariablen hängt von der Art der Zustandsvariablen ab.

Für Referenzen (Zustandsvariablen mit expliziter Dereferenzierung) bedeutet die Gleichheit natürlich die Gleichheit der Referenzen (also der Speicherzellen):

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref 5;
val w = ref 5 : int ref

- v = w;
val it = false : bool

- !v = !w;
val it = true : bool
```

Gleichheit (II)

Für Zustandsvariablen ohne explizite Dereferenzierung bezieht sich die Gleichheit üblicherweise auf die Inhalte.
Die könnte so aussehen (in einer Phantasiesprache):

```
- declare v = pointer to 5;
v = pointer to 5 : pointer to int;

- declare w = pointer to 5;
w = pointer to 5 : pointer to int;

- w = v;
val it = true : bool
```

Viele Sprachen mit expliziter Dereferenzierung verwenden die Bezeichnung „pointer“ (*Zeiger*) für (symbolische) Speicheradressen.

Alias-Problem

Referenzen und Zeiger ermöglichen, dieselbe Zustandsvariable in verschiedener Weise zu benennen. In SML etwa:

```
- val w = ref 5;
val w = ref 5 : int ref

- val z = w;
val z = ref 5 : int ref
```

`w` und `z` sind Referenzen auf dieselbe Speicherzelle (mit Inhalt `5`).

Man beachte den Unterschied, wenn `z` als `val z = ref 5` deklariert worden wäre!

Alias-Problem (II)

Das Arbeiten mit solchen Zustandsvariablen kann leicht unübersichtlich werden — etwa, wenn die vorangehende Sitzung wie folgt fortgesetzt wird:

```
- w := 0;
val it = () : unit;

- w;
val it = ref 0 : int ref
```

Da `z` dieselbe Referenz wie `w` bezeichnet, verändert die Zuweisung `w := 0` nicht nur `w`, sondern auch `z` (genauer: den Inhalt).

In großen Programmen kann die Verwendung solcher „Aliase“ zur Unübersichtlichkeit und Unverständlichkeit führen.

[Mit funktionalen Variablen ist so etwas nicht möglich!]

Zyklische Referenzierung

Betrachten wir die SML-Sitzung:

```
-  val a = ref (fn(x : int) => 0);
val a = ref fn : (int -> int) ref

- !a(12);
val it = 0 : int
```

`!a` ist eine Funktion, die jede ganze Zahl auf 0 abbildet.

```
- val b = ref (fn(x : int) => if x = 0 then 1 else x * !a(x - 1));

- !b(0);
val it = 1 : int

- !b(3);
val it = 0 : int
```

`!b` ist eine Funktion, die 0 auf 1 und jede andere ganze Zahl auf 0 abbildet.

Zyklische Referenzierung (II)

```
- a := !b;
val it = () : unit

- !b(0);
val it = 1 : int

- !b(3);
val it = 6 : int
```

Nach der Zuweisung `a := !b` ist `!b` nun die rekursive Fakultätsfunktion.

In Zusammenhang mit komplexen Datenstrukturen ($\rightarrow$ später!) können zyklische Referenzierungen nützlich sein.

3.5.3. Zustandsvariablen in SML: Referenzen

[Zusammenfassung der vorherigen Konzepte, soweit sie in SML verfügbar sind]

- In SML sind Zustandsvariablen Referenzen, also (symbolische) Speicheradressen.
- Ist `v` eine SML-Referenz eines Objektes vom Typ `t`, so ist `t ref` der Typ von `v` (siehe die vorangehenden Beispielsitzungen).
- Der Referenzierungsoperator von SML ist „`ref`”:

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

Zustandsvariablen in SML: Referenzen (II)

- Mit dem Dereferenzierungsoperator „`!`“ und den vorangehenden Deklarationen lässt sich der folgende Ausdruck bilden und auswerten:

```
- !w(!v);
val it = 0 : int
```

- Der Sequenzierungsoperator von SML ist „`;`“.

  Der Wert einer Sequenz von Ausdrücken `(A1 ; A2 ; ...; An)` ist der Wert des letzten Ausdrucks, d.h. der Wert von `An`.

- Der Zuweisungsoperator von SML ist „`:=`“ (becomes).

  Der Wert einer Zuweisung ist „`()`“ (unity).

Zustandsvariablen in SML: Referenzen (III)

SML liefert Referenzen, druckt sie aber nicht aus. Anstelle einer Referenz druckt SML das Symbol `ref` gefolgt von ihrem Inhalt:

```
- val v = ref 5;
val v = ref 5 : int ref

- v;
val it = ref 5 : int ref

-  val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- w;
val it = ref fn : (int -> int) ref
```

Dieses Druckverhalten von SML sollte nicht dazu verleiten, die Referenzen von SML für Zustandsvariablen ohne explizite Dereferenzierung zu halten!

Zustandsvariablen in SML: Referenzen (IV)

- Die Referenz einer Referenz ist selbstverständlich möglich:

```
-  val w = ref (fn x:int => 0);
val w = ref fn : (int -> int) ref

- val z = ref w;
val z = ref (ref fn) : (int -> int) ref ref

- !(!z)(9);
val it = 0 : int
```

- Die Gleichheit für Referenzen in SML ist „=“.

3.6. Funktionale Programmierung versus Imperative Programmierung

- Programmierung mit Zustandsvariablen nennt man *imperative Programmierung*.

- Die imperative Programmierung stellt ein Programmierparadigma dar, das sich wesentlich von der funktionalen Programmierung unterscheidet.

- Programmiersprachen wie SML bieten die Möglichkeit, beide Programmierparadigmen zu vermischen.

  Jedoch ist das Hauptparadigma solcher Sprachen nur eines von beiden Paradigmen.

Im Folgenden:
einige Unterschiede zwischen funktionaler und imperativer Programmierung

3.6.1. Überschatten versus Zustandsänderung

Überschatten und *Zustandsänderungen* sind zwei unterschiedliche Techniken:

- Überschatten verändert den Wert einer bereits existierenden Variable nicht, sondern verwendet deren Name für eine neue Speicherzelle.
- Eine Zustandsänderung erstellt keine neue Variable, sondern verändert den Inhalt einer bereits vorhandenen Speicherzelle.

Moderne imperative Programmiersprachen kennen auch das Überschatten.

3.6.2. Funktion versus Prozedur

- Eine *Funktion* kann auf Argumente angewandt werden und liefert einen Wert als Ergebnis, ohne dabei Zustandsveränderungen zu verursachen.

  Eine Funktion ist also referenztransparent.

- *Prozedur* ist ein Oberbegriff, der sowohl Funktionen umfasst als auch Programme, die Zustandsveränderungen verursachen.

  Eine Prozedur, die keine Funktion ist (also Zustandsveränderungen verursacht), liefert je nach Programmiersprache entweder überhaupt keinen Wert als Ergebnis oder einen uninteressanten Wert wie () in SML.

Funktion versus Prozedur (II)

- In vielen Programmiersprachen kann man Prozeduren definieren, die keine Funktionen sind, weil sie Zustandsveränderungen verursachen, die aber Werte als Ergebnis liefern, als seien sie Funktionen.

  Das ist oft schlechter Programmierstil!

  (konstruiertes?) Beispiel:

  Ein Programm `ln`, das den Logarithmus zu einer Basis $b = e$ berechnet (e ist die Euler'sche Zahl) und das beim Aufruf `ln(`e`)` nicht nur den Wert `1.0` liefert, sondern nebenbei noch den Wert der Basis b verändert!

- Prozeduren, die Werte liefern, die also in Ausdrücken an derselben Stelle wie Funktionen vorkommen können, werden häufig — fälschlich bzw. irreführend — Funktionen genannt.

3.6.3. Unzulänglichkeit des Substitutionsmodells zur Behandlung von Zustandsvariablen

- Mit Zustandsvariablen und Zustandsveränderungen ist die Referenztransparenz durchbrochen.
- Der Wert eines Ausdrucks hängt nun vom Zeitpunkt ab, zu dem er ausgewertet wird.

$\leadsto$ Das Substitutionsmodell ist hier nicht mehr sinnvoll anwendbar!
(siehe nachfolgendes Beispiel)

Zur Beschreibung von Zustandsvariablen wird ein anderes, wesentlich komplizierteres Berechnungsmodell benötigt: das *Umgebungsmodell* (wird später behandelt)

Unzulänglichkeit des Substitutionsmodells . . . (II)

In der folgenden Sitzung hat der Ausdruck `!kto` verschiedene Werte (bei gleichbleibender Umgebung):

```
- val kto = ref 100;
val kto = ref 100 : int ref

- kto := ! kto + 50;
val it = () : unit

- kto;
val it = ref 150 : int ref

-  kto := ! kto + 50 ;
val it = () : unit

- kto;
val it = ref 200 : int ref
```

3.6.4. Rein funktionale Programme und Ausdrücke

- Funktionale Programme und Ausdrücke ohne Zustandsvariablen werden *rein funktional* genannt.

- Zustandsvariablen erschweren Beweise von Programmeigenschaften (wie z.B. Terminierung), weil zusätzlich zu den Deklarationen die Veränderungen des Programmzustands berücksichtigt werden müssen.

  Dafür müssen die zeitlichen Programmabläufe berücksichtigt werden, wozu sogenannte „temporallogische“ Formalismen verwendet werden.

3.6.5. Nebeneffekte

Die Zustandsveränderungen, die sich aus der Auswertung von nicht rein funktionalen Ausdrücken oder Programmen ergeben, werden

- *Nebenwirkungen* oder
- *Nebeneffekte* (*side effects*)

genannt.

3.6.6. Reihenfolge der Parameterauswertung

Weder der Auswertungsalgorithmus noch das Substitutionsmodell legen die Auswertungsreihenfolge der aktuellen Parameter $\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ eines Ausdrucks $\mathtt{B}(\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n})$ fest.

Die aktuellen Parameter können

- von links nach rechts (also in der Reihenfolge $\mathtt{A}_1$, $\mathtt{A}_2$ bis $\mathtt{A}_\mathtt{n}$) oder
- von rechts nach links (also in der Reihenfolge $\mathtt{A}_\mathtt{n}$, $\mathtt{A}_{\mathtt{n}-1}$, . . . , $\mathtt{A}_1$) oder
- in irgendeiner anderen Reihenfolge ausgewertet werden.

Für rein funktionale Programme beeinflusst die Reihenfolge der Auswertung der aktuellen Parameter das Ergebnis nicht.

Dies ist aber anders, wenn einige der aktuellen Parameter nicht rein funktional sind, weil ihre Auswertung Nebeneffekte hat, so dass der Zustand am Ende der Parameterauswertung von der Reihenfolge dieser Auswertung abhängt.

4. Prozeduren zur Abstraktionsbildung

- 4.1 Die „Prozedur": Ein Kernbegriff der Programmierung
- 4.2 Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
- 4.3 Prozeduren versus Prozesse
- 4.4 Ressourcenbedarf — Größenordnungen
- 4.5 Beispiel: Der größte gemeinsame Teiler

4.1. Die „Prozedur": Ein Kernbegriff der Programmierung

- Programmzerlegung durch Prozeduren
- Vorteile von Prozeduren
- Funktion vs. Prozedur
- Funktionen und Prozeduren in SML

4.1.1. Prozeduren zur Programmzerlegung

Die formale Spezifikation des Multiplikationsalgorithmus (Abschnitt 1.1.6), d.h. die Funktion `integer_mult`, nimmt Bezug auf weitere Funktionen:

`integer_digit_mult`	Multiplikation einer natürlichen Zahl mit einer einstelligen natürlichen Zahl
`one_digit_integer`	Test, ob eine natürliche Zahl einstellig ist
`digit_mult`	die Multiplikationstabelle
...	...

Diese Funktionen stellen Teilalgorithmen dar, die in der Spezifikation des Multiplikationsalgorithmus verwendet werden.

Prozeduren zur Programmzerlegung (II)

Die Verwendung von Teilalgorithmen zur Spezifikation eines Algorithmus ist eine natürliche Vorgehensweise:

- Zur Lösung eines Problems lohnt es sich in der Regel, Teilprobleme zu erkennen, zu formalisieren und getrennt vom Hauptproblem zu lösen.
- Aus diesem natürlichen Ansatz ist der Begriff „Prozedur“ entstanden, der ein Kernbegriff der Programmierung ist.
- Prozeduren sind Teilprogramme, die Teilaufgaben lösen.
- Jede Programmiersprache ermöglicht die Programmierung von Prozeduren. (auch die sogenannten „niederen“ Maschinensprachen)
- Die Vorteile von Prozeduren sind vielseitig.
- Sie werden im Folgenden an Hand der formalen Spezifikation des Multiplikationsalgorithmus erläutert.

4.1.2. Vorteile von Prozeduren

1. Prozeduren ermöglichen die Zerlegung eines Programms in übersichtliche Teilprogramme.
   - Die Verwendung der Funktion `integer_digit_mult` macht die Spezifikation der Funktion `integer_mult` übersichtlicher.
   - Jedes einzelne Teilprogramm kann leichter als das Gesamtprogramm spezifiziert und auf Korrektheit überprüft werden.

2. Prozeduren ermöglichen Mehrfachverwendung von identischen Programmteilen an verschiedenen Stellen im Programm.
   - Die Funktion `integer_digit_mult` wird zweimal in der Definition der Funktion `integer_mult` verwendet.
   - Die Erkennung mehrfachverwendbarer Teile erleichtert sowohl die Spezifikation als auch die Implementierung und die Wartung von Programmen.

Vorteile von Prozeduren (II)

3. Prozeduren ermöglichen die Verwendung von sogenannten „lokalen Variablen“ mit präzise abgegrenzten Geltungsbereichen.

   - *Beispiel:* `integer_digit_mult` verwendet die „lokalen Variablen“ `product1` und `product2` als Namen für bei der Berechnung auftretende Zwischenwerte.
   - Lokale Variablen werden „nach außen“ nicht „weitergegeben“.
   - Die Verwendung von lokalen Variablen als Namen für konstante Zwischenwerte in Berechnungen trägt oft dazu bei, Berechnungen verständlicher zu machen.
   - Lokale Variablen als Namen für konstante Werte helfen, Mehrfachberechnungen desselben Wertes zu vermeiden.

Vorteile von Prozeduren (III)

- Lokale Variablen können auch als Namen für Funktionen und Prozeduren dienen, die als Hilfsmittel verwendet werden.
  Beispiel: In einer komplexen Software ist es sinnvoll, die Funktion `integer_digit_mult` lokal zu der Funktion `integer_mult` zu deklarieren.
- Variablennamen, die in einer Prozedur lokal für konstante Werte oder für Prozeduren benutzt werden, stehen zu (anderen) Verwendungen außerhalb dieser Prozedur zur Verfügung.
- Die Prozedur, in der eine lokale Variable deklariert wird, ist der „Geltungsbereich" dieser Variablen.

Vorteile von Prozeduren (IV)

4. Prozeduren sind austauschbare Programmkomponenten.

   Wie z.B. die Funktion `digit_mult` implementiert ist, ist für die Definition der Funktion `integer_mult`, die diese Funktionen verwendet, unwichtig.

   Jede partiell korrekte bzw. total korrekte Implementierung von `digit_mult`, die der Spezifikation entspricht, kann verwendet werden:
   Sie ergibt eine partiell korrekte bzw. total korrekte Implementierung von `integer_mult`.

   - Während der Programmentwicklung wird zunächst eine „einfache“ Implementierung von Prozeduren verwendet.
   - Diese werden später durch „bessere“ Implementierungen ersetzt, ohne dass solche Änderungen die Korrektheit des Gesamtprogramms in Frage stellen.

4.1.3. Funktion versus Prozedur

Nicht alle Prozeduren sind Funktionen:

Funktionen liefern ein Ergebnis und haben keine Nebeneffekte.
Prozeduren, die keine Funktionen sind, verursachen Nebeneffekte und liefern keine oder uninteressante Werte wie `()` in SML.

- Ist die verwendete Programmiersprache rein funktional, dann sind alle Prozeduren Funktionen.
- Ist die Programmiersprache nicht rein funktional, dann sind manche Prozeduren keine Funktionen.

Leider ist der Sprachgebrauch uneinheitlich und häufig ungenau:

Es wird auch von Funktionen gesprochen, wenn es um Prozeduren (mit Nebeneffekten) geht, die einen Wert liefern (auch „Funktionsprozedur“ genannt).

4.1.4. Definition von Funktionen und Prozeduren in SML

SML bietet zwei alternative Schreibweisen zur Definition von Funktionen oder Prozeduren ohne Pattern Matching:

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

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

SML bietet ebenfalls zwei alternative Schreibweisen zur Definition von Funktionen oder Prozeduren mit Pattern Matching:

```
val rec fak   =   fn 0 => 1
                   | n => n * fak(n-1);

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

[Die zweite Schreibweise ist jeweils nur „syntaktischer Zucker“ für die erste.]

Definition von Funktionen und Prozeduren in SML (II)

- In SML gibt es neben Funktionen aber auch einige Prozeduren, die Nebeneffekte haben, wie die Prozedur `use` zum Laden einer Datei.
- Konvention für SML:
  Solche Prozeduren liefern stets den Wert `()` vom Typ `unit` als Ergebnis.
- Eine Prozedur, die von der Hardware die Uhrzeit erhält und als Wert liefert, verursacht keinen Nebeneffekt.
  - keine Funktion, weil sie nicht referenztransparent ist:

    Der gelieferte Wert ist bei gleichen aktuellen Parameter nicht immer derselbe.

  $\leadsto$ Unterscheidung zwischen (nichtfunktionalen) Prozeduren, wenn sie Nebeneffekte verursachen, und Funktionen ist nicht ganz zutreffend.

  [Die Uhrzeit liefernde Prozedur ist aber ein seltener Grenzfall!]

4.2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen

Prozeduren dienen zur Definition von Zwischenergebnissen und Teilberechnungen.

- Lokale Deklarationen ermöglichen die Verwendung von Variablen für konstante Werte, Funktionen oder Prozeduren innerhalb einer Prozedur, die nur innerhalb dieser Prozedur verwendet werden können.
- Lokale Deklarationen sind in (fast) allen modernen „höheren“ Programmiersprachen möglich.
- Maschinensprachen ermöglichen keine echten lokalen Deklarationen in dem Sinne, dass sie nicht sicherstellen, dass die Geltungsbereiche geschützt sind.

SML bietet zwei syntaktische Möglichkeiten, Namen lokal zu deklarieren:

- Zum einen mit dem Ausdruck „`let`“,
- zum anderen mit dem Ausdruck „`local`“.

4.2.1. Lokale Deklarationen mit `let`

`let`-Ausdrücke werden verwendet, um Deklarationen lokal zu einem Ausdruck zu deklarieren, der selbst keine Deklaration ist.

Betrachten wir die folgende Funktion

$$
\begin{array}{rccl}
f: & \mathbb{N} & \longrightarrow & \mathbb{N} \\
 & x & \longmapsto & (3x+1)^2 + (5x+1)^2 + (3x)^3 + (5x)^3
\end{array}
$$

Eine unmittelbare Implementierung in SML sieht wie folgt aus:

```
fun hoch2(x : int) = x * x;

fun hoch3(x : int) = x * x * x;

fun f(x) = hoch2(3*x + 1) + hoch2(5*x + 1)
         + hoch3(3*x)   + hoch3(5*x);
```

Lokale Deklarationen mit `let` (II)

Kommen die Funktionen `hoch2` und `hoch3` nur in der Definition von `f` vor, so bietet es sich an, diese Funktionen wie folgt lokal zu `f` zu deklarieren:

```
fun f(x) = let fun hoch2(x : int) = x * x
               fun hoch3(x : int) = x * x * x
           in
                 hoch2(3*x + 1) + hoch2(5*x + 1)
               + hoch3(3*x)     + hoch3(5*x)
           end;
```

Die Syntax (d.h. Schreibweise) des definierenden Teils (oder Rumpfs) einer Funktion- oder Prozedurdeklaration mit lokalen Deklarationen ist also:

```
let ... in ... end;
```

Andere Syntax für mehrere lokale Deklarationen

Zwischen `let` und `in` finden sich eine oder mehrere Deklarationen (in der üblichen Syntax), die mit „ ; “ getrennt werden dürfen (aber nicht müssen).

Also auch zulässig:

```
fun f(x) = let fun hoch2(x : int) = x * x;     (* ";" beachten *)
               fun hoch3(x : int) = x * x * x
           in
                 hoch2(3*x + 1) + hoch2(5*x + 1)
               + hoch3(3*x)     + hoch3(5*x)
           end;
```

[Auf die Wirkung von „ ; “ zwischen lokalen Deklarationen gehen wir später ein.]

Lokale Deklaration in Definitionen von anonymen Funktionen

Unter Verwendung einer anonymen Funktion (SML-Konstrukt `fn`, gesprochen lambda ($= \lambda$)) kann die Funktion `f` in SML wie folgt implementiert werden:

```
val f = fn (x) =>
           let fun hoch2(x : int) = x * x
               fun hoch3(x : int) = x * x * x
           in
                 hoch2(3*x + 1) + hoch2(5*x + 1)
               + hoch3(3*x)     + hoch3(5*x)
           end;
```

In anonymen Funktionen kann `let` genau so benutzt werden wie in benannten Funktionen.

Verbesserungen des Programmbeispiels

Die Implementierung der Funktion `f` kann wie folgt verbessert werden:

- An Stelle der lokalen Funktion `hoch2` kann eine Funktion `hoch2plus1` verwendet werden.

  Dies führt zu einem verständlicheren, weil kompakteren, Programm:

```
fun f(x) = let fun plus1hoch2(x : int) =
                    let  fun hoch2(x) = x * x
                    in   hoch2(x + 1)
                    end
               fun hoch3(x : int)   =  x * x * x
           in
                  plus1hoch2(3*x) + plus1hoch2(5*x)
                + hoch3(3*x)      + hoch3(5*x)
           end;
```

Verbesserungen des Programmbeispiels (II)

- Die mehrfache Berechnung von `3*x` und `5*x` kann mit Hilfe lokaler Variablen vermieden werden:

  (Mehrfachberechnungen können sehr kostspielig sein.)

```
(*)    fun f(x) = let fun plus1hoch2(x : int) = (x + 1) * (x + 1)
                      fun hoch3(x : int)      = x * x * x
                      val x3 = 3*x
                      val x5 = 5*x
                  in
                        plus1hoch2(x3) + plus1hoch2(x5)
                      + hoch3(x3)      + hoch3(x5)
                  end;
```

Anmerkung: Manche Übersetzer erkennen in einigen Fällen mehrfache Berechnungen und führen eine ähnliche Änderung durch wie in diesem Beispiel.

Verschachtelte lokale Deklarationen

Verschachtelte lokale Deklarationen sind selbstverständlich möglich wie etwa in der vorangehenden Deklaration der Funktion `f` oder in der folgenden Deklaration:

```
fun h(x) = let
               fun g(y) = let
                              fun hoch2(z : int) = z * z
                          in
                              hoch2(y) + hoch2(y + 1)
                          end
           in
               g(x) * g(x + 1)
           end;
```

4.2.2. Lokale Deklarationen mit `local`

`local`-Ausdrücke erlauben, Deklarationen lokal zu Deklarationen zu deklarieren:

$$
\begin{array}{rccc}
f: & \mathbb{N} & \rightarrow & \mathbb{N} \\
 & x & \mapsto & (3x+1)^2 + (5x+1)^2 + (3x)^3 + (5x)^3
\end{array}
$$

kann ähnlich wie in `(*)` wie folgt deklariert werden:

```
local fun plus1hoch2(x : int) = x * x + 1
      fun hoch3(x : int)      = x * x * x
in
  fun f(x) =
      let val x3 = 3 * x
          val x5 = 5 * x
      in
          plus1hoch2(x3) + plus1hoch2(x5) + hoch3(x3) + hoch3(x5)
      end
end
```

Lokale Deklarationen mit `local` (II)

Die Syntax eines local-Ausdrucks ist: `local ... in ... end`

- Die lokalen Deklarationen von `x3` und `x5` können *nicht* zwischen `local` und dem ersten `in` stehen:

  Dieser Bereich liegt außerhalb des Geltungsbereiches des Parameters `x`; zwischen `local` und dem ersten `in` ist `x` daher unbekannt.

- Zwischen `in` und `end` kann noch eine weitere Funktion `f'` deklariert werden, die ebenfalls die lokalen Definitionen `plus1hoch2` und `hoch3` verwendet.

  Bei `let`–Ausdrücken (wie in `(*)`) ist dies nicht möglich.

4.2.3. Unterschied zwischen `let` und `local`

`let` ermöglicht Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind.

Beispiel:

```
let val zwei = 2
    val drei = 3
in
    zwei + drei
end;
```

Unterschied zwischen `let` und `local` (II)

Nicht erlaubt: `let`–Deklarationen lokal zu Ausdrücken, die selbst Deklarationen sind:

```
let val zwei = 2
    val drei = 3
in
    val fuenf = zwei + drei                        (* inkorrekt *)
end;

let val zwei = 2
    val drei = 3
in
    fun f(x) = x + zwei + drei                     (* inkorrekt *)
end;
```

Unterschied zwischen `let` und `local` (III)

In solchen Fällen kann `local` verwendet werden:

```
local val zwei = 2
      val drei = 3
in
      val fuenf = zwei + drei
end;

local val zwei = 2
      val drei = 3
in
      fun f(x) = x + zwei + drei
end;
```

- Nach der ersten Deklaration ist die Variable `fuenf` mit Wert `5` deklariert.
- Die Variablen `zwei` und `drei` sind dagegen nicht deklariert.
- Nach der zweiten Deklaratrion ist ganz entsprechend nur die Funktion `f` deklariert.

Unterschied zwischen `let` und `local` (IV)

`local` ermöglicht wiederum keine Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind.

Der folgende Ausdruck ist daher inkorrekt:

```
local val zwei = 2
      val drei = 3
in
      zwei + drei                                  (* inkorrekt *)
end;
```

4.2.4. Blockstruktur und Überschatten

Programmiersprachen, in denen Namen für konstante Werte, Funktionen oder Prozeduren lokal deklariert werden können, besitzen eine sogenannte „Blockstruktur“.

$\leadsto$ *Blocksprachen*

Die Blockstruktur ist ein wichtiges Mittel zur Strukturierung von Programmen:

- Sie ermöglicht eine Begrenzung der Wirkung von Deklarationen.
- Ist ein Name `N` lokal zu einem Programmteil deklariert, kann derselbe Name `N` in einem anderen Programmteil mit ganz anderer Bedeutung verwendet werden.

Ein `let`- oder `local`-Ausdruck stellt einen sogenannten *Block* dar.
(siehe folgendes Beispiel)

Blockstruktur und Überschatten (II)

```
- fun f(x) = let val a = 2
             in
                 a * x
             end;
val f = fn : int -> int

- fun g(x) =  let val a = 2000
              in
                 a * x
              end;
val g = fn : int -> int

- f(1);
val it = 2 : int

- g(1);
val it = 2000 : int
```

4.2.5. Festlegung der Geltungsbereiche von Namen — Einführung

- In SML ist ein *Block* einfach ein Ausdruck.
- Die Verwendung der Bezeichung *Block* wurde hier von imperativen Programmiersprachen übertragen, für die der Blockbegriff ursprünglich eingeführt wurde.
- Ein Block ist also eine rein syntaktische Einheit, die weder von den aktuellen Parametern einer Funktionsanwendung noch vom aktuellen Zustand der Umgebung abhängt.
- Der Geltungsbereich eines Namens (oder einer Variablen) in Blocksprachen erfolgt nach den folgenden einfachen Prinzipien.

Festlegung der Geltungsbereiche von Namen . . . (II)

1. Der Geltungsbereich eines Namens `N` ist der Block, der die Deklaration von `N` enthält — mit Ausnahme aller in dem Block vorkommenden Blöcke, die Deklarationen desselben Namens `N` enthalten.

2. Kommt ein Name `N` in einem Block `B` vor und ist `N` in `B` nicht deklariert, so gilt:

   (a) `N` muss in einem Block `B'` deklariert sein, der `B` umfasst.

   (b) Wenn mehrere Blöcke, die `B` umfassen, Deklarationen für `N` enthalten, so gilt die Deklaration desjenigen Blocks, der `B` am engsten umfasst.

Dass ein Block eine rein syntaktische Einheit ist, entspricht dem Prinzip der statischen Bindung — Blocksprachen beruhen also immer auf der statischen Bindung.

Die Bezeichnung „lexikalische Bindung“ anstelle von „statische Bindung“ hebt diese Tatsache hervor.

4.2.6. Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung

Die eben aufgeführten Regeln lassen sich einfach implementieren:

$\leadsto$ Umgebung mit einer geordneten Liste von Gleichungen der Gestalt `Name = Wert`

- Bei der Auswertung eines neuen Ausdrucks ($\rightarrow$ neuer Block) führt jede Deklaration eines Namens `N` für einen Wert `W` zu einem Eintrag `N = W` am Anfang der Umgebung.
- Beim Austritt aus einem Block werden alle Einträge der Gestalt `N = W` gelöscht, die während der Auswertung des Ausdrucks in die Umgebung eingefügt wurden.
- Um den Wert eines Namens zu ermitteln, wird die Umgebung vom Anfang her durchlaufen:
  Die zuerst gefundene Gleichung `N = W` gilt als Definition von `N`.

  Das heißt, die zuletzt hinzugefügte Gleichung für `N` gilt,
  „ältere“ Deklarationen werden so „überschattet“.

4.2.7. Überschatten durch verschachtelte lokale Deklarationen

Lokale Deklarationen ermöglichen das Überschatten von Deklarationen:

```
- fun f(x) = let val a = 2
                 fun g(x) = let val a = 3
                            in
                                a * x
                            end
             in
                 a * g(x)
             end;
val f = fn : int -> int

- f(1);
val it = 6 : int
```

Überschatten durch verschachtelte lokale Deklarationen (II)

Die Anwendung des Substitutionsmodell auf den Ausdruck `f(1)` erklärt den Wert dieses Ausdrucks:

```
f(1)
a * g(1)
2 * g(1)
2 * (a * 1)
2 * (3 * 1)
2 * 3
6
```

Überschatten durch verschachtelte lokale Deklarationen (III)

- Die Anwendung des Substitutionsmodells wird hier von den Regeln zur Festlegung der Geltungsbereiche von Namen als auch der Algorithmus zur Verwaltung der Umgebung verwendet werden.

- Mit der Einführung von lokalen Variablen verliert also das Substitutionsmodell einen Teil seiner Einfachheit.

- Das Überschatten der ersten Deklaration von `a` wäre hier nicht notwendig gewesen.

  Durch Umbenennung des inneren `a` in `b` würde keine Namensgleichheit auftreten.

- Übermäßiger Einsatz des Überschattens macht Programme unleserlich und mißverständlich.

4.2.8. Festlegung der Geltungsbereiche von Namen unter Verwendung der Umgebung – Fortsetzung

Auswertung von val-Deklarationen

- Die Auswertung einer Deklaration `val N = A` geschieht auch in einer Umgebung U.
- Zunächst wird der Ausdruck `A` in dieser Umgebung U ausgewertet.
- Erst danach wird die Umgebung U (an ihrem Anfang) um die Gleichung `N = A` erweitert.

Zur Auswertung des Rumpfes (des definierenden Teils) einer Deklaration gilt also noch die „alte Umgebung" vor Berücksichtigung dieser Deklaration.

Eintrag einer Funktionsdeklaration in die Umgebung

Zur Auswertung einer Funktionsdeklaration wie

```
(*)     val N = fn P => W
```

wird die folgende Gleichung zu der Umgebung (am Anfang) hinzugefügt.

```
(**)    N = fn P => W
```

- Zum Zeitpunkt der Auswertung der Deklaration (*) wird der Ausdruck `W` nicht ausgewertet.

  Dies ergäbe auch keinen Sinn:

  Zum Zeitpunkt der Auswertung der Deklaration (*) sind keine aktuellen Parameter bekannt, daher ist auch keine Anwendung der Funktion namens `N` durchzuführen.

Eintrag einer Funktionsdeklaration in die Umgebung (II)

Bei der (späteren) Auswertung einer Anwendung der Funktion namens `N` wird aber `W` ausgewertet:

- Die Umgebung ist eine geordnete Liste.
- Aufgrund zwischenzeitlich ausgewerteter Deklarationen können in dieser Liste andere Gleichung vor der für die Funktion namens `N` stehen.
- Die Blockstruktur (bzw. das Prinzip der statischen Bindung) verlangt, dass nur die Gleichungen, die „älter" als die Deklaration von `N` sind, zur Auswertung von `W` berücksichtigt werden.

Umgebung [wird vom Anfang her gelesen oder erweitert]:

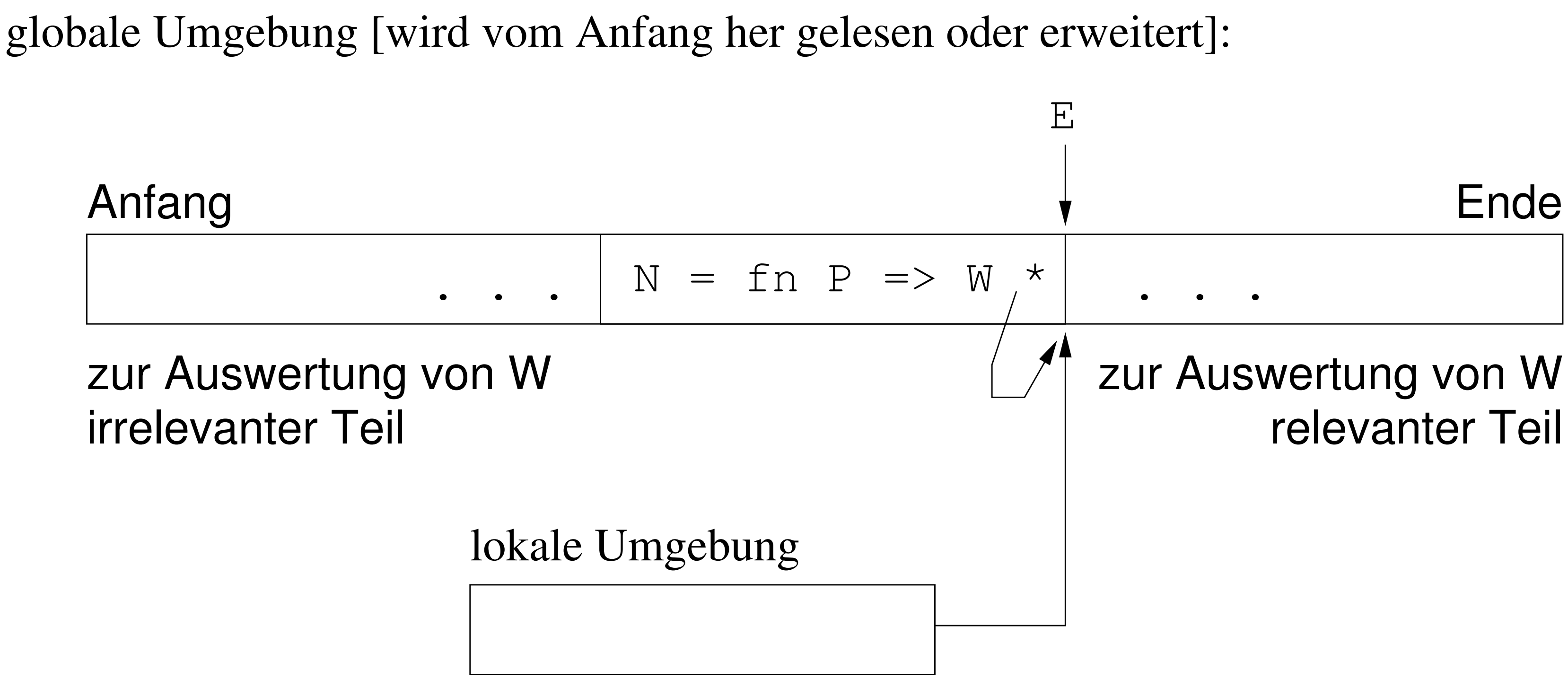
Anfang		Ende
. . .	`N = fn P => W`	. . .

zur Auswertung von W irrelevanter Teil | | zur Auswertung von W relevanter Teil

Auswertung einer Funktionsanwendung — Lokale Umgebung

Zur Auswertung einer Anwendung der Funktion namens `N` müssen aber auch die aktuellen Parameter der Funktionsanwendung berücksichtigt werden.

Der Teil der Umgebung, der zur Auswertung von `W` relevant ist, wird also um eine sogenannte „lokale Umgebung“ erweitert:



Auswertung einer Funktionsanwendung—Lokale Umgebung (II)

- `*` stellt einen Verweis dar (auf dessen Funktion wir etwas später eingehen werden).
- Für die Auswertung einer Anwendung der Funktion mit Namen `N` besthet die zu berücksichtigende Umgebung aus:
  - der lokalen Umgebung
  - gefolgt von dem Teil der (nichtlokalen, auch global genannten) Umgebung, die zur Auswertung von `W` relevant ist.

  Der Verweis `E` gibt die Stelle an, ab der die Umgebung während der Auswertung einer Anwendung der Funktion mit dem Namen `N` gelesen wird.
- Da jede Prozedurdeklaration lokale Deklarationen beinhalten kann, kann während einer Auswertung eine Verkettung von mehreren lokalen Umgebungen entstehen.

  Diese Verkettung spiegelt die Schachtelung von Ausdrucken wider, in denen Parametern von Funktionsanwendungen oder lokale Variablen deklariert werden.

Sequenzielle Auswertung von lokalen `let`-Deklarationen

Aufeinanderfolgende Deklarationen werden in der gegebenen Reihenfolge ausgewertet:

```
- val x = 2;
val x = 2 : int

- let val x = 3 * x;
      val x = 5 * x
  in
      1 + x
  end;
val it = 31 : int
```

Sequenzielle Auswertung von lokalen `let`-Deklarationen (II)

Der `let`-Ausdruck wirkt also genau wie der folgende, in dem alle Namenskonflikte durch Umbenennung beseitigt sind:

```
- val x = 2;
val x = 2 : int

- let val x3 = 3 * x;
      val x5 = 5 * x3
  in
      1 + x5
  end;
val it = 31 : int
```

Nichtsequenzielle Auswertung von lokalen Deklarationen

Es gibt eine Variante von `let`, bei der aufeinanderfolgende Deklarationen gemeinsam behandelt werden:

```
- val x = 2;
val x = 2 : int

- let val x  = 3 * x
      and x5 = 5 * x
  in
      x * x5
  end;
val it = 60 : int
```

Nichtsequenzielle Auswertung von lokalen Deklarationen (II)

Wirkung von `and`:

- Zuerst werden alle Ausdrücke in den Rümpfen der Deklarationen in derselben äußeren Umgebung ausgewertet.

  Danach werden alle berechneten Werte gleichzeitig an ihre Namen gebunden.
- An die lokale Variable `x5` wird also der Wert `10` gebunden.

  Ersetzen wir `and` durch `val` wird `x5` an den Wert `30` gebunden.
- Das Konstrukt „`and`“ kann selbstverständlich auch in `local`-Ausdrücken verwendet werden.

In SML wie in vielen Programmiersprachen drückt „`;`“ die Sequenzierung von Ausdrücken aus:

Daher kann zur Verdeutlichung bei sequenzieller Auswertung lokaler Deklarationen „`;`“ zwischen die Deklarationen geschrieben werden.

Wechselseitige Rekursion

- Zwei oder mehrere Funktionen heißen „wechselseitig rekursiv“, wenn sie sich wechselseitig aufrufen.
- Das `and`-Konstrukt ist für die Deklaration von wechselseitig rekursiven Funktionen unabdingbar, weil die Rümpfe aller Deklarationen in derselben Umgebung ausgewertet werden müssen.

Ein einfaches Beispiel für zwei wechselseitig rekursive Funktionen ist:

```
val rec ungerade = fn 0 => false
                    | 1 => true
                    | n => gerade(n-1)
and
        gerade   = fn 0 => true
                    | 1 => false
                    | n => ungerade(n-1);
```

Wechselseitige Rekursion (II)

Unter Verwendung des `fun`-Konstrukts sieht dies so aus:

```
fun ungerade(0) = false
  | ungerade(1) = true
  | ungerade(n) = gerade(n-1)
and
    gerade(0) = true
  | gerade(1) = false
  | gerade(n) = ungerade(n-1);
```

Auswertung von `val-rec`–Deklarationen

Eine Funktionsdeklaration der Form

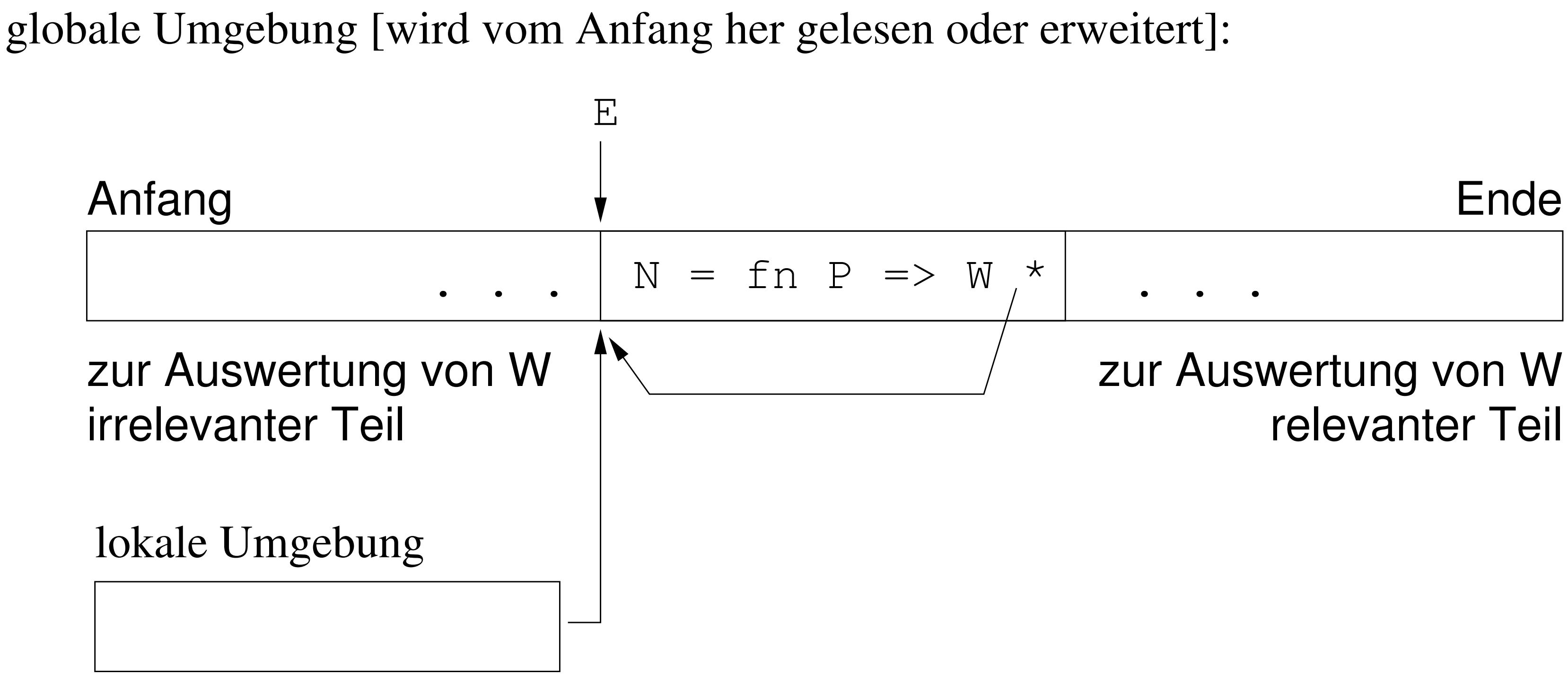
```
val rec N = fn P => W
```

wird ähnlich wie eine Funktionsdeklaration der Form

```
val N = fn P => W
```

behandelt mit dem Unterschied, dass der Verweis am Ende der lokalen Umgebung *vor* statt hinter die Gleichung `N = fn P => W` in der globalen Umgebung zeigt.

Auswertung von `val-rec`–Deklarationen (II)



Der Verweis `E` gibt die Stelle an, ab dem die Umgebung während der Auswertung einer Anwendung der rekursiven Funktion namens `N` gelesen wird.

Auswertung von `fun`-Deklarationen

Ein Funktionsdeklaration der Form

```
fun N P = W
```

wird genauso wie die Funktionsdeklaration

```
val rec N P = W
```

ausgewertet.

Der Konstrukt „`fun`“ ist lediglich „syntaktischer Zucker“.

4.3. Prozeduren versus Prozesse

- Notwendigkeit der Rekursion
- Rekursion vs. Iteration — Vergleich
- Rekursion vs. Iteration — Komplexitätsaspekte
- Endrekursion
- Lineare und quadratisch Rekursion, Rekursion der Potenz n
- Iterative Berechnung der Fibonacci-Zahlen
- Memoisierung
- Prozedur vs. Prozess

4.3.1. Notwendigkeit der Rekursion

In rein funktionalen Programmiersprachen ist Rekursion die einzige Möglichkeit, eine Anzahl von Wiederholungen zu ermöglichen, die von den aktuellen Parametern abhängt.

Betrachten wir z.B. die folgende Funktion:

$$
\begin{array}{lcll}
\text{summe}(0) & = & 0 & \\
\text{summe}(n) & = & n + \ldots + 0 & \text{falls } n \in \mathbb{N} \text{ und } n \geq 1
\end{array}
$$

Sie lässt sich in SML (mit Pattern Matching in der „syntaktisch verzuckerten“ Form) wie folgt rekursiv implementieren:

```
fun summe(0) = 0
  | summe(n) = n + summe(n-1);
```

Notwendigkeit der Rekursion (II)

Ohne Rekursion müsste man die Anzahl der Summanden unabhängig von dem formalen Parameter `n` festlegen; z.B.

```
fun summe'(0) = 0
  | summe'(1) = 1 + 0
  | summe'(2) = 2 + 1 + 0
  | summe'(3) = 3 + 2 + 1 + 0
  | summe'(4) = 4 + 3 + 2 + 1 + 0
  | ???
```

Kann man diese Deklaration ohne Rekursion vollenden, so dass `summe'` auf der unendlichen Menge der natürlichen Zahlen total ist?

Mit den bisher eingeführten Mitteln ist das unmöglich.

4.3.2. Rekursion versus Iteration: Grundkonstrukte beider Berechnungsmodelle im Vergleich

Die Rekursion ist nicht die einzige Möglichkeit, Wiederholungen zu spezifizieren.

Betrachten wird den früher vorgestellten (rekursiven) Algorithmus zum Zeichnen eines Viereck auf den Boden:

Viereckalgorithmus 1:

Zeichne eine Seite wie folgt:

> Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts.
>
> Wenn Du nicht am Startpunkt stehst, dann zeichne eine Seite unter Anwendung des oben geschilderte Verfahrens.

Rekursion versus Iteration: . . . (II)

Dies lässt sich aber auch so formulieren:

Viereckalgorithmus 2:

Wiederhole 4-mal: Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts.

oder auch wie folgt:

Viereckalgorithmus 3:

Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts. Wiederhole dies, solange Du nicht am Startpunkt stehst.

Vergleich der Viereckalgorithmen

Die drei obigen Viereckalgorithmen unterscheiden sich hinsichtlich der Form der Wiederholung:

- Der Viereckalgorithmus 1 ist rekursiv, die Algorithmen 2 und 3 sind es nicht.
- Auch die Viereckalgorithmen 2 und 3 unterscheiden sich wesentlich voneinander:
  - Im Viereckalgorithmus 2 ist die Anzahl der Wiederholungen durch einen expliziten Wert (4) festgelegt;
  - im Viereckalgorithmus 3 hängt die Anzahl der Wiederholungen von einer Bedingung ab, die nach jeder Wiederholung ausgewertet werden muss.

$\rightarrow$ sogenannte *Iterationsschleifen* (2 und 3 sind iterative Algorithmen)

Algorithmus 2: eine sogenannte „For-Schleife“
Algorithmus 3: eine sogenannte „While-Schleife“

4.3.3. Rekursion versus Iteration: Komplexitätsaspekte

Betrachten wir die Fakultätsfunktion, die über den natürlichen Zahlen total ist.

Eine etwas unpräzise Definition der Fakultätsfunktion sieht wie folgt aus:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1) * \ldots * 1 \qquad \text{falls } n \in \mathbb{N} \text{ und } n \geq 1 \end{aligned}$$

Die folgende rekursive Definition derselben Funktion ist präziser, weil sie nicht von der Schreibweise „. . . “ (d.h. usw.) Gebrauch macht:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \qquad \text{falls } n \in \mathbb{N} \text{ und } n \geq 1 \end{aligned}$$

Rekursion versus Iteration: Komplexitätsaspekte (II)

Diese rekursive Definition lässt sich in SML direkt übertragen:

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

Auswertung gemäß dem Substitutionsmodell von `fak(4)`:

```
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)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Rekursion versus Iteration: Komplexitätsaspekte (III)

- Diese Auswertung bedarf einigen Speichers für unvollendete Zwischenberechnungen wie etwa `4 * (3 * (2 * fak(1)))`.
- Damit die unterbrochenen Zwischenberechnungen hinterher korrekt weitergeführt werden, ist zudem eine Buchführung notwendig, die Zeit kostet.

Es geht aber auch mit weniger Speicher und weniger Buchführung, z.B. für `4!` :

```
(*)     4        1
        4 - 1    1 * 4
        3        4
        3 - 1    4 * 3
        2        12
        2 - 1    12 * 2
        1        24
        1 - 1    24 * 1
        0        24
```

Rekursion versus Iteration: Komplexitätsaspekte (IV)

Hier wurde nicht nur ein einziger, sondern zwei Ausdrücke schrittweise umgewandelt:

- Links wurde der jeweilige Wert von `n` dargestellt,
- rechts die jeweiligen Teilprodukte.

Rechts werden sozusagen die Zwischenergebnisse aufgesammelt („akkumuliert“).

Eine Variable zum Aufsammeln von Zwischenergebnissen nennt man üblicherweise einen *Akkumulator*.

Rekursion versus Iteration: Komplexitätsaspekte (V)

Durch Verwendung von Zustandsvariablen für `n` sowie für den Akkumulator kann man den Algorithmus wie folgt mit einer Iterationsschleife spezifizieren:

(**) Iterativer Algorithmus zur Berechnung von `fak(!n)` für `!n`$\in \mathbb{N}$:
(in einer imperativen Phantasiesprache, die Zustandsvariablen mit expliziter Dereferenzierung und einen Dereferenzierungsoperator „`!`“ hat)

```
akk := 1
while !n > 0 do
   akk := !akk * !n;   n := !n - 1
end-while
return !akk
```

Dieser iterative Algorithmus führt genau zu den Schritten aus `(*)`, wobei `n` die Werte der linken Spalte von `(*)` und `akk` die der rechten Spalte annimmt.

Rekursion versus Iteration: Komplexitätsaspekte (VI)

Lassen sich die Berechnungsschritte in (*) auch rein funktional erzielen?

Dies leistet die folgende Funktion:

```
fun fak_iter(n) =
    let fun hilf_fak_iter(n, akk) =
                        if n = 0 then akk
                                 else hilf_fak_iter(n - 1, akk * n)
    in
        hilf_fak_iter(n, 1)
    end
```

Rekursion versus Iteration: Komplexitätsaspekte (VII)

Unter Verwendung des Substitutionsmodells können wir uns vergewissern, dass die Auswertung von `fak_iter(4)` genau die Schritte (*) durchläuft:

```
fak_iter( 4 )
hilf_fak_iter(4,      1      )
hilf_fak_iter(4 - 1,  1 * 4  )
hilf_fak_iter(3,      4      )
hilf_fak_iter(3 - 1,  4 * 3  )
hilf_fak_iter(2,      12     )
hilf_fak_iter(2 - 1,  12 * 2 )
hilf_fak_iter(1,      24     )
hilf_fak_iter(1 - 1,  24 * 1 )
hilf_fak_iter(0,      24     )
24
```

Rekursion versus Iteration: Komplexitätsaspekte (VIII)

Die rekursive Funktion `fak_iter` implementiert also den iterativen Algorithmus `(**)`.

Vergleich von *Iteration* und *Rekursion* am Beispiel der Fakultätsfunktion:

- Der Unterschied zwischen der Funktion `fak_iter` und dem iterativen Algorithmus ist nicht sehr wesentlich.

  Beide Formalismen scheinen ähnlich anschaulich oder unanschaulich zu sein.
- Welche Form ist „natürlicher“? [kann man nicht wirklich objektiv beantworten]
- `fak` ist zwar weniger effizient als `(**)` bzw. `fak_iter`, aber dafür viel einfacher und leichter zu entwickeln.
- Ein Korrektheitsbeweis für `fak` ist viel einfacher als für `(**)`.

Die Gegenüberstellung von Iteration und Rekursion ist oft von sehr subjektiven Argumenten (und Vorurteilen) geprägt.

Beide Techniken haben ihre Berechtigung.

4.3.4. Endrekursion

Die Funktion `fak_iter` (mit Hilfsfunktion `hilf_fak_iter`) zeichnet sich dadurch aus, dass keine Zwischenberechnungen unterbrochen werden.

Ob die Auswertung einer rekursiven Funktion Unterbrechungen von Berechnungen verlangt, ist syntaktisch erkennbar:

- Kommt der rekursive Aufruf `R` in einem zusammengesetzten Ausdruck `A` vor, der keine Fallunterscheidung (`if-then-else` oder `case`) ist, werden Berechnungen unterbrochen:

  Die Auswertung von `A` ist erst dann möglich, wenn der zugehörige Teilausdruck `R` ausgewertet wurde.

  Beispiel: `n * fak(n - 1)` im Falle der Funktion `fak`.

- Kommt der rekursive Aufruf nicht in einem zusammengesetzten Ausdruck vor, der keine Fallunterscheidung (`if-then-else` oder `case`) ist, dann sind keine Unterbrechungen von Berechnungen nötig.

  Beispiel: die Funktion `hilf_fak_iter`.

Endrekursion (II)

Eine rekursive Funktion, deren Rumpf keinen rekursiven Aufruf enthält, der ein echter Teilausdruck eines zusammengesetzten Ausdrucks `A` ist, so dass `A` weder ein `if-then-else`– noch ein `case`–Ausdruck ist, nennt man „endrekursiv“ (*tail recursive*).

- Der Ausschluss von `if-then-else`– und `case`–Ausdrücken spiegelt wider, dass diese Ausdrücke Sonderausdrücke sind, also anders ausgewertet werden.

  Würden Fallunterscheidungen wie herkömmliche Ausdrücke behandelt, dann würde die Definition nichttriviale rekursive Funktionen, die Basisfälle von Rekursionsfälle unterscheiden, ausschließen.

- Endrekursive Funktionen sind rekursiv. Ihre Auswertung führt aber zu iterativen (Berechnungs-) Prozessen.

- Nennt man eine Prozedur (bzw. Programm) *rekursiv* (bzw. *endrekursiv*), so meint man die Syntax der Prozedur, nicht den zugehörigen (Berechnungs-) Prozess.

Endrekursion (III)

Die Funktion `hilf_fak_iter` ist also rein syntaktisch rekursiv, aber auch endrekursiv. Damit ist sichergestellt, dass der Berechnungsprozess, der von dieser Funktion ausgelöst wird, ein iterativer Prozess ist.

- Ältere Implementierungen funktionaler Sprachen haben auch bei endrekursiven Funktionen die Berechnung unterbrochen.

  Die so entstehenden Berechnungsprozesse waren dadurch natürlich zeitaufwändiger als die durch Iterationsschleifen in imperativen Sprachen verursachten.

- Jede moderne Implementierung einer Programmiersprache erzeugt aus einem endrekursiven Programm einen iterativen (Berechnungs-) Prozess.

  Das so entstehende (Maschinen-) Programm ist genauso gut (oder schlecht) wie eines, das aus einer imperativen Sprache generiert wird.

4.3.5. Lineare und quadratische Rekursion, Rekursion der Potenz n

Beispiel: „Fibonacci-Zahlen“, die wie folgt definiert sind:

Definition Fibonacci-Zahlen:

$$\begin{array}{rcl} fib(0) & = & 0 \\ fib(1) & = & 1 \\ fib(n) & = & fib(n-1) + fib(n-2) \text{ für } n \in \mathbb{N} \text{ und } n \geq 2 \end{array}$$

Die Funktion `fib` lässt sich wie folgt in SML rekursiv implementieren:

```
fun fib(0) = 0
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2);
```

Lineare und quadratische Rekursion, Rekursion der Potenz n (II)

zeitliches Verhalten der Funktion `fib`:

- Bis etwa `n = 20` erfolgt die Auswertung von `fib(n)` schnell.
- Für größere Werte wie z.B. `n = 35` oder `n = 40` nimmt die Rechenzeit für die Auswertung deutlich zu.

Ursache: die besondere Form der Rekursion in de Definition:

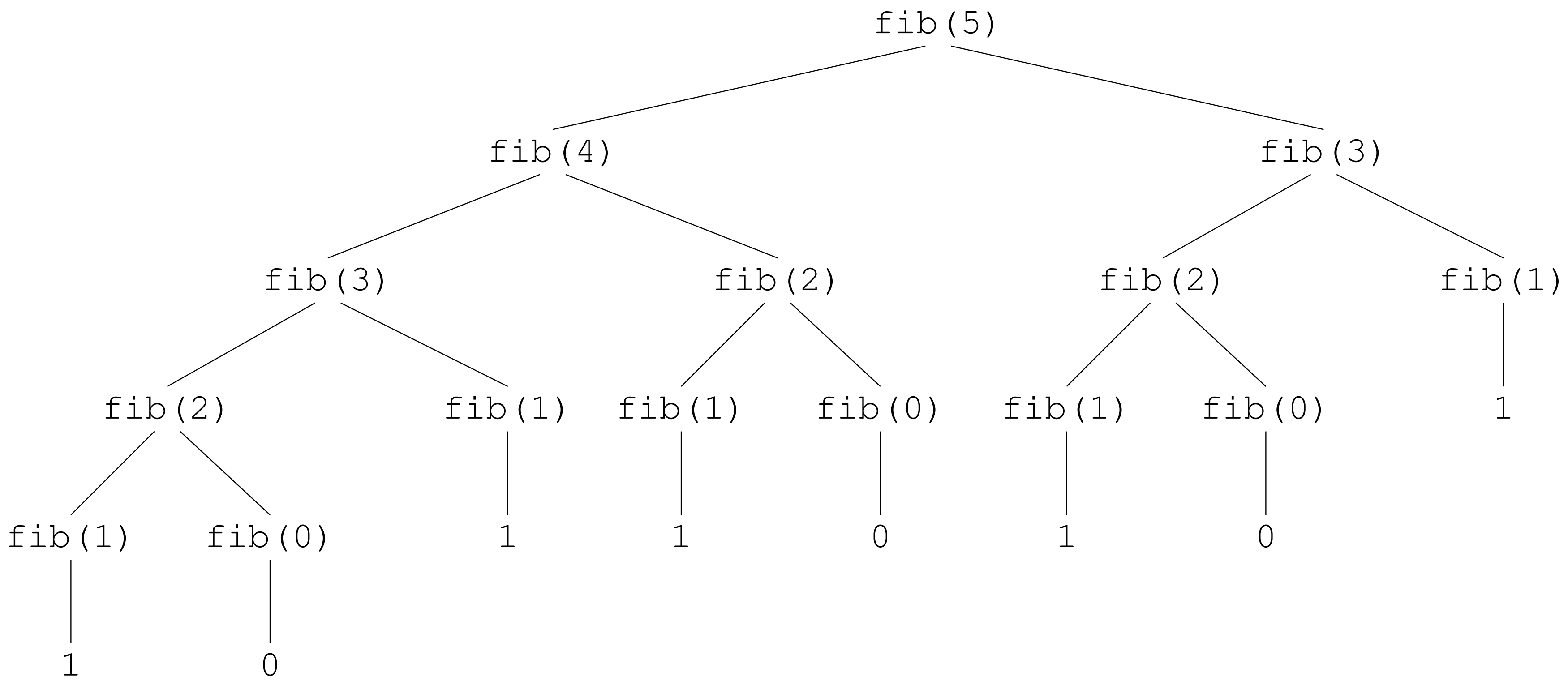
Im definierenden Teil der Funktion `fib` kommen *zwei* rekursive Aufrufe vor.

- Kommt nur ein rekursiver Aufruf im Rumpf einer Funktion vor, so heißt die Funktion *linear rekursiv*.
- Kommen zwei rekursive Aufrufe im Rumpf einer Funktion vor, so heißt die Funktion *quadratisch rekursiv*.
- Kommen n rekursive Aufrufe im Rumpf einer Funktion vor, so heißt die Funktion *rekursiv in der Potenz* n.

Lineare & quadratische Rekursion, Rekursion der Potenz n (III)

Rekursive Funktionen, die quadratisch rekursiv oder rekursiv in der Potenz n sind, heißen auch *baumrekursiv*.

Die Auswertung eines Aufrufs `fib(5)` kann wie folgt dargestellt werden:



Lineare & quadratische Rekursion, Rekursion der Potenz n (IV)

Es ist interessant, dass die Fibonacci-Funktion sich asymptotisch wie die Funktion

$$\begin{array}{ccc} \mathbb{N} & \rightarrow & \mathbb{N} \\ \mathtt{n} & \mapsto & \phi^{\mathtt{n}} \end{array}$$

verhält.

Asymptotisch bedeutet: Für hinreichend große `n` wird der Unterschied zwischen `fib(n)` und $\phi^{\mathtt{n}}$ beliebig klein wird.

ϕ ist hier der „goldene Schnitt“, d.h. die Zahl, die durch die Gleichung $\phi^2 = \phi + 1$ definiert ist, also $\phi = \frac{1+\sqrt{5}}{2} \approx 1,6180$.

Mit Hilfe dieses Zusammenhangs kann man zeigen, dass die Anzahl der Blätter im obigen Baum und damit die benötigte Berechnungszeit exponenziell mit `n` wächst.

Hinweis: Dass eine Funktion *quadratisch rekursiv* ist bedeutet nicht, dass auch ihr Zeitbedarf proportional zu $\mathtt{n}^2$ ist.

4.3.6. Iterative Auswertung der baumrekursiven Funktion `fib`

Auch für baumrekursive Funktionen lassen sich oft iterative Auswertungen finden. Im Fall von `fib(n)` kann dies mit zwei Akkumulatoren `akk1` und `akk2` und einer Zwischenvariablen `z` geschehen:

(***) Iterativer Algorithmus zur Berechnung von `fib(!n)` für `!n`$\in \mathbb{N}$:
(wieder in einer imperativen Phantasiesprache mit Dereferenzierungsoperator „ ! “):

```
i := 0
akk1 := 0                  (* !akk1: die i-te Fibonaccizahl *)
akk2 := 1                  (* !akk2: die (i+1)-te Fibonaccizahl *)
while !i < !n do
     i := !i + 1
     z := !akk2
     akk2 := !akk1 + !akk2
     akk1 := !z
end-while
return !akk1
```

Iterative Auswertung der baumrekursiven Funktion `fib` (II)

`z` ist eine Hilfsvariable, um den Inhalt von `akk2` zu retten, bevor der Inhalt von `akk2` verändert wird, damit der alte Inhalt anschließend zum Inhalt von `akk1` gemacht werden kann.

Die Berechnung durchläuft z.B. für `!n = 8` die folgenden Zustände:

`!i`	`!akk1`	`!akk2`
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

4.3.7. Memoisierung

Es fällt auf, dass eine Funktion wie `fib` wiederholte Auswertungen der gleichen Ausdrücke verlangt.

Zur Auswertung von `fib(5)` werden

`fib(4)`	1 mal
`fib(3)`	2 mal
`fib(2)`	3 mal
`fib(1)`	5 mal
`fib(0)`	3 mal

ausgewertet.

Im Folgenden:
Ein Ansatz zur effizienten Auswertung der baumrekursiven Funktion `fib` ohne Veränderung ihrer Definition

Memoisierung (II)

Der Ansatz besteht in einer Veränderung des Auswertungsalgorithmus.

Dabei wird Buch über bereits ausgewertete Aufrufe der Funktion (hier: `fib`) geführt:

- Wurde ein rekursiver Aufruf `A` mit Ergebniswert `W` ausgewertet, so wird die Gleichung `A = W` in einer Tabelle T gespeichert (”memoisiert“).
- Bei Auswertung eines rekursiven Aufrufs `A` wird zunächst in der Tabelle T nach einem Eintrag `A = W` gesucht.

  Gibt es einen solchen Eintrag, so wird der Aufruf mttA nicht wieder ausgewertet, sondern der Wert `W` aus der Tabelle geliefert.

Die „Memoisierung“ führt hier nicht zu einem iterativen (Berechnungs-) Prozess, verbessert jedoch die Auswertungszeiten erheblich.

Dafür ist eine Buchführung notwendig, die zeitaufwändig ist.
Es hängt vom Programm ab, ob diese Buchführung sich lohnt.

4.3.8. Prozedur versus Prozess

Die Beispiele der Endrekursion und der Memoisierung zeigen:

- Die Syntax einer Prozedur (oder eines Programms) bestimmt nicht allein, welcher Art der zugehörige (Berechnungs-) Prozess ist.

  Der (Berechnungs-) Prozess hängt sowohl von der Prozedur (oder dem Programm) als auch vom Auswertungsalgorithmus ab.

Die Auswertungsalgorithmen moderner funktionaler Programmiersprachen erkennen die Endrekursion und erzeugen aus endrekursiven Programmen iterative Prozesse.

4.4. Ressourcenbedarf — Größenordnungen

Prozesse verbrauchen vor allem zwei Ressourcen: *Rechenzeit* und *Speicherplatz*.

Es ist üblich, diesen Verbrauch durch Vergleichsmaße („Größenordnungen") anzugeben.

- Zunächst wird für den betrachteten Prozess ein Parameter n festgelegt, der eine natürliche Zahl ist und der die „Größe" des Problems abschätzt.
  - Multiplikationsalgorithmus: $n =$ Länge der Operanden
  - Fibonacci-Zahlen: $n =$ Aufrufparameter von `fib`
- Für die betrachtete Ressource (Rechenzeit bzw. Speicherplatz) wird eine Rechenzeit- bzw. Speicherplatz-Einheit festgelegt.
- Der Ressourcenverbrauch eines Prozesses wird als Funktion $r : \mathbb{N} \rightarrow \mathbb{N}$ definiert, die jede Größe n eines Problems auf die Anzahl benötigter Rechenzeit- bzw. Speicherplatz-Einheiten abbildet.

Ressourcenbedarf — Größenordnungen (II)

Definition (Größenordnung)

Seien $f : \mathbb{N} \rightarrow \mathbb{N}$ und $s : \mathbb{N} \rightarrow \mathbb{N}$ zwei Funktionen (s wie Schranke).
Die Funktion f ist von der Größenordnung $O(s)$, geschrieben $f \in O(s)$, wenn es $k \in \mathbb{N}$ und $m \in \mathbb{N}$ gibt, so dass gilt:

Für alle $n \in \mathbb{N}$ mit $n \geq m$ ist $f(n) \leq k * s(n)$.

- Die Konstante k ist in dieser Definition unabhängig von n.

  Es muss also ein und dieselbe Konstante k sein, die für alle $n \in \mathbb{N}$ garantiert, dass $f(n) \leq k * s(n)$ gilt.
- Wenn eine solche Konstante k nicht existiert, ist f nicht von der Größenordnung $O(s)$.
- $O(s)$ repräsentiert die Menge aller Funktionen, die bzgl. s die in der Definition angegebene Eigenschaft erfüllen.

Ressourcenbedarf — Größenordnungen (III)

- Ist f von der Größenordnung $O(s)$, wobei s eine Funktion z.B. $n \mapsto 2n$ oder $n \mapsto n^4$ ist, so sagt man vereinfachend, dass f von der Größenordnung $O(2n)$ bzw. $O(n^4)$ ist.

  Dabei wird $2n$ bzw. n^4 als „Kurzschreibweise“ für die eigentliche Funktion s verwendet.

- Ferner schreibt man oft auch $f = O(s)$ statt $f \in O(s)$.

  Dies ist formal eigentlich inkorrekt, da f nicht gleichzeitig Element von $O(s)$ und gleich zu $O(s)$ sein kann.

  Außerdem könnte man aus z.B. $f2 = O(n)$ und $f3 = O(n)$ fälschlich schließen, dass $f2$ und $f3$ identisch sind.

  Dies ist aber nicht notwendigerweise so, etwa für $f2$ und $f3$ wie folgt:

$$f2: \begin{array}{ccc} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & 2n \end{array} \qquad f3: \begin{array}{ccc} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & 3n \end{array}$$

4.5. Beispiel: Der größte gemeinsame Teiler

- Der größte gemeinsame Teiler (ggT) zweier natürlicher Zahlen a und b ist die größte natürliche Zahl, durch die sowohl a als b teilbar (d.h. ohne Rest dividierbar) ist.

  z.B. $30 = 2 * 3 * 5$ ist der ggT von $60 = (2^2) * 3 * 5$ und $150 = 2 * 3 * (5^2)$.

- Die Notation $t|a$ besagt, dass t ein Teiler von a ist (das heißt, es gibt ein $k \in \mathbb{N}$ mit $a = t * k$).

- Ist t ein gemeinsamer Teiler von a und von b, dann gilt $t|a$ und $t|b$.

- Der ggT zweier natürlicher Zahlen a und b kann leicht aus den Zerlegungen in Primzahlen von a und von b ermittelt werden. Er ist der Produkt aller p^n mit:
  - die Primzahl p kommt in jeder der beiden Zerlegungen (einmal) vor, einmal mit Exponent n_1, einmal mit Exponent n_2.
  - n ist das Minimum von n_1 und n_2.

Beispiel: Der größte gemeinsame Teiler (II)

Die Zerlegung einer natürlicher Zahl in Primzahlen ist eine zeitaufwendige Aufgabe, so dass dieser Berechnungsansatz des ggT zweier natürlicher Zahlen ziemlich ineffizient ist.

Ein effizienterer Ansatz zur Berechnung des ggT zweier natürlicher Zahlen beruht auf der folgenden Eigenschaft:

Satz:

Seien $a \in \mathbb{N}$ und $b \in \mathbb{N}$ mit $a \geq b$. Sei r der Rest der Ganzzahldivision von a durch b (d.h. $a = (b * c) + r$ für ein $c \in \mathbb{N}$). Sei $t \in \mathbb{N}$.

$$t|a \text{ und } t|b \text{ genau dann, wenn } t|b \text{ und } t|r.$$

Beispiel: Der größte gemeinsame Teiler (III)

Beweis:

Seien a, b, c und r wie im Satz definiert.

Notwendige Bedingung (Richtung von links nach rechts; „$\Longrightarrow$“):

Sei angenommen, dass $t|a$ und $t|b$.

Zu zeigen ist, dass $t|b$ und $t|r$ gelten.
Da nach Annahme $t|b$ gilt, reicht es aus, $t|r$ zu zeigen.
Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.
Da $t|a$ gilt, gibt es t_a mit $a = t * t_a$.
Nach Annahme gilt $a = b * c + r$
also $t * t_a = a = b * c + r = t * t_b * c + r$,
also $r = t * t_a - t * t_b * c = t * (t_a - t_b * c)$, d.h. $t|r$.

Beispiel: Der größte gemeinsame Teiler (IV)

Hinreichende Bedingung (Richtung von rechts nach links; „$\Longleftarrow$“):

Sei angenommen, dass $t|b$ und $t|r$.

Zu zeigen ist, dass $t|a$ und $t|b$ gelten.
Da nach Annahme $t|b$ gilt, reicht es aus, $t|a$ zu zeigen.
Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.
Da $t|r$ gilt, gibt es t_r mit $r = t * t_r$.
Nach Annahme gilt $a = b * c + r$,
also $a = t * t_b * c + t * t_r = t * (t_b * c + t_r)$, d.h. $t|a$.

qed.

Aus dem Satz folgt:
Der größte gemeinsame Teiler $ggT(a, b)$ zweier natürlicher Zahlen a und b mit $a \geq b$ und $a = b * c + r$ ist gleich dem größten gemeinsamen Teiler $ggT(b, r)$ von b und r.

Beispiel: Der größte gemeinsame Teiler (V)

Diese Beobachtung liefert einen rekursiven Ansatz zur Berechnung des größten gemeinsamen Teilers zweier natürlichen Zahlen a und b:

1. Wenn $a < b$, dann vertausche a und b.
2. Andernfalls (d.h. $a \geq b$) ist der größte gemeinsame Teiler von a und b der größte gemeinsame Teiler von b und r, wobei r der Rest der Ganzzahldivision von a durch b (d.h. in SML **r = a mod b**) ist.

Die aktuellen Parametern (a, b) werden paarweise bei jedem rekursiven Aufruf (außer beim ersten, falls $a < b$) echt kleiner, weil der Rest einer Ganzzahldivision durch b echt kleiner als b ist.

Beispiel: Der größte gemeinsame Teiler (VI)

Was ist aber der Basisfall der Rekursion?

- Man beachte zuerst, dass $ggT(a, 0) = a$ ist, weil $0 = 0 * a = 0 * a + 0$ gilt.

  Der Rest der Ganzzahldivision von 0 durch a ist 0 (d.h. für jedes $a \in \mathbb{N} : a|0$).

Führt aber der oben geschilderte rekursive Ansatz zwangsläufig zu einem rekursiven Aufruf mit aktuellen Parametern der Form $(a, 0)$?

Beweis:

Bei jedem rekursiven Aufruf ist der zweite aktuelle Parameter (d.h. der Wert des formalen Parameters b) eine natürliche Zahl, die echt kleiner als der zweite aktuelle Parameter des vorherigen rekursiven Aufrufs ist.
Zudem ist diese natürliche Zahl immer größer oder gleich 0.
Zwischen dem Wert von b (beim ersten Aufruf) und 0 gibt es nur endlich viele natürlichen Zahlen.
$\leadsto$ nach endlich vielen rekursiven Aufrufen hat b den Wert 0

qed.

Beispiel: Der größte gemeinsame Teiler (VII)

Damit können wir die folgende endrekursive Funktion zur Berechnung des ggT zweier natürlichen Zahlen definieren:

```
fun ggT(a, b) =
     if a < b then ggT(b, a)
              else if b = 0 then a
                             else ggT(b, a mod b);
```

Der Algorithmus, den die Funktion `ggT` implementiert, konvergiert sehr schnell, wie die folgende Anwendung des Substitutionsmodells zeigt:

```
ggT(150, 60)
ggT(60, 30)
ggT(30, 0)
30
```

Beispiel: Der größte gemeinsame Teiler (VIII)

- Dieser Algorithmus zur Berechnung des ggT zweier natürlicher Zahlen wird Euklid (ca. 3. Jhdt. vor Christus) zugeschrieben, weil er in Euklids „Elemente der Mathematik“ erwähnt ist.

- Er gilt als der älteste bekannte Algorithmus weil er im Gegensatz zu anderen überlieferten Algorithmen aus älteren oder sogar jüngeren Zeiten in Euklids „Elemente der Mathematik“ nicht mittels Beispielen, sondern abstrakt (mit Redewendungen anstelle von Variablen) spezifiziert ist.

  weil er im Gegensatz zu anderen überlieferten Algorithmen aus älteren oder sogar jüngeren Zeiten in Euklids „Elemente der Mathematik“ nicht mittels Beispielen, sondern abstrakt (mit Redewendungen anstelle von Variablen) spezifiziert ist.

- Die vorangehende Spezifikation des Euklid'schen Algorithmus ist eine endrekursive Funktion. Die Auswertung dieser Funktion löst also einen iterativen (Berechnungs-) Prozess aus.

Beispiel: Der größte gemeinsame Teiler (IX)

Ein Satz des französischen Mathematikers Gabriel Lamé (19. Jhdt.) ermöglicht es, die Rechenzeit des Euklid'schen Algorithmus abzuschätzen:

Satz (Lamé)

Seien $a \in \mathbb{N}$ und $b \in \mathbb{N}$, so dass $a \geq b$.

Benötigt der Euklid'sche Algorithmus zur Berechnung von $ggT(a, b)$ insgesamt n Rekursionsschritte, so gilt $b \geq fib(n)$, wobei $fib(n)$ die n-te Fibonacci-Zahl ist.

Anmerkung:
$n = 0$, also kein rekursiver Aufruf erfolgt, wenn $\mathtt{a} < \mathtt{b}$ oder $\mathtt{b} = 0$; da $\mathtt{a} \geq \mathtt{b}$ vorausgesetzt wird, bleibt nur der Fall $\mathtt{b} = 0$

Beispiel: Der größte gemeinsame Teiler (X)

Beweis:

Betrachten wir drei aufeinander folgende rekursive Aufrufe des Algorithmus:

$$\mathtt{ggT}(\mathtt{a}_0, \mathtt{b}_0) \qquad \mathtt{ggT}(\mathtt{a}_1, \mathtt{b}_1) \qquad \mathtt{ggT}(\mathtt{a}_2, \mathtt{b}_2)$$

Nach Definition gilt:

$\mathtt{a}_1 = \mathtt{b}_0$
$\mathtt{b}_1 = \mathtt{a}_0 \ \mathtt{mod}\ \mathtt{b}_0$, d.h. $\mathtt{a}_0 = \mathtt{b}_0 * \mathtt{c}_0 + \mathtt{b}_1$ (für ein $\mathtt{c}_0 \in \mathbb{N}$)
$\mathtt{a}_2 = \mathtt{b}_1$
$\mathtt{b}_2 = \mathtt{a}_1 \ \mathtt{mod}\ \mathtt{b}_1$, d.h. $\mathtt{a}_1 = \mathtt{b}_1 * \mathtt{c}_1 + \mathtt{b}_2$ (für ein $\mathtt{c}_1 \in \mathbb{N}$)

Da $\mathtt{c}_0 \geq 1$ und $\mathtt{c}_1 \geq 1$ ist, folgt:
(*) $\mathtt{a}_0 \geq \mathtt{b}_0 + \mathtt{b}_1$ und $\mathtt{a}_1 = \mathtt{b}_0 \geq \mathtt{b}_1 + \mathtt{b}_2$

Dies entspricht gerade dem Bildungsgesetz für die Fibonacci-Zahlen.

$\leadsto$ *Hypothese:* $\mathtt{b}_0 \geq fib(n)$
[$n =$ Anzahl rekursiver Aufrufe während der Berechnung von $(\mathtt{a}_0, \mathtt{b}_0)$]

Beispiel: Der größte gemeinsame Teiler (XI)

Diese Hypothese wird zunächst noch verschärft zu:

Für alle $n \in \mathbb{N}$ gilt: für alle $k \in \mathbb{N}$, $k \leq n$ und alle $\mathtt{a}, \mathtt{b} \in \mathbb{N}$ mit $\mathtt{a} \geq \mathtt{b}$, für die die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau k rekursive Aufrufe benötigt, gilt $\mathtt{b} \geq fib(k)$.

[mit $k = n$ ergibt sich daraus die ursprüngliche Hypothese]

Seien im Folgenden $k, n \in \mathbb{N}$, $k \leq n$, $\mathtt{a}, \mathtt{b} \in \mathbb{N}$ mit $\mathtt{a} \geq \mathtt{b}$, für die die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau k rekursive Aufrufe benötigt:

Basisfall: $n = 0$
Da $0 \leq k \leq n$ folgt $k = 0$: dann gilt $\mathtt{b} \geq 0 = fib(0) = fib(k)$.

Basisfall: $n = 1$
Da $o \leq k \leq n$ folgt $k = 0$ oder $k = 1$: für $k = 0$ ist die Hypothese erfüllt, wie im vorigen Fall gezeigt. Bleibt $k = 1$, d.h. genau ein rekursiver Aufruf:
dann gilt $\mathtt{b}_1 = 0$, damit Rekursion nach einem Schritt abbricht;
da $\mathtt{b}_1 = \mathtt{a}_0 \ \mathtt{mod} \ \mathtt{b}_0$, ist $\mathtt{b} = \mathtt{b}_0 > \mathtt{b}_1$, also $\mathtt{b} \geq 1 = fib(1) = fib(k)$.

Beispiel: Der größte gemeinsame Teiler (XII)

Induktionsfall:
Sei $n \geq 2$; *Induktionsannahme:*
für alle $0 \leq k \leq n$ gelte die zu beweisende Hypothese.

Sei nun $k \leq n+1$: Für $k \leq n$ gilt die Induktionsannahme, bleibt $k = n+1$:

Die Auswertung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ benötige also $n+1$ rekursive Aufrufe.
Sei $\mathtt{a}_0 = \mathtt{a}$ und $\mathtt{b}_0 = \mathtt{b}$ und seien $\mathtt{ggT}(\mathtt{a}_1, \mathtt{b}_1)$ und $\mathtt{ggT}(\mathtt{a}_2, \mathtt{b}_2)$ die zwei ersten rekursiven Aufrufe.
Nach Konstruktion benötigt die Auswertung von $\mathtt{ggT}(\mathtt{a}_1, \mathtt{b}_1)$ genau n rekursive Aufrufe und die Auswertung von $\mathtt{ggT}(\mathtt{a}_2, \mathtt{b}_2)$ genau $n-1$ rekursive Aufrufe.
Nach Induktionsannahme gilt damit $\mathtt{b}_1 \geq fib(n)$ und $\mathtt{b}_2 \geq fib(n-1)$.
Gemäß (*) gilt ferner: $\mathtt{b}_0 \geq \mathtt{b}_1 + \mathtt{b}_2$.

Also $\mathtt{b} = \mathtt{b}_0 \geq fib(n) + fib(n-1) = fib(n+1) = fib(k)$.

qed.

Beispiel: Der größte gemeinsame Teiler (XIII)

Benötigt der Euklid'sche Algorithmus zur Berechnung von $\mathtt{ggT}(\mathtt{a}, \mathtt{b})$ genau n rekursive Aufrufe, so gilt nach dem Satz von Lamé (wobei ϕ der „goldene Schnitt“ ist):

$$\mathtt{b} \geq fib(n) \approx \phi^n$$

Daraus folgt, dass asymptotisch, also für große n, gilt (mit der Schreibweise $\log_\phi$ für den Logarithmus zur Basis ϕ):

$$\log_\phi \mathtt{b} \geq n$$

Da es aber eine Konstante k gibt mit $\log_\phi \mathtt{b} \leq k * \ln \mathtt{b}$, wobei $\ln$ den Logarithmus zur Basis e bezeichnet, gilt asymptotisch auch $n \leq k * \ln \mathtt{b}$, also

$$\mathtt{ggT}(\mathtt{a}, \mathtt{b}) \in O(\ln(\mathtt{b})).$$

Die Anzahl der Rekursionsschritte, die der Euklid'sche Algorithmus zur Berechnung des ggT zweier natürlicher Zahlen benötigt, ist höchstens logarithmisch in diesen Zahlen (oder präziser in der kleineren der beiden Zahlen).

5. Die vordefinierten Typen von SML

- 5.1 Was sind Typen?
- 5.2 Die Basistypen von SML
- 5.3 Zusammengesetzte Typen in SML
- 5.4 Listen
- 5.5 Beispiele: Grundlegende Listenfunktionen
- 5.6 Hinweis auf die Standardbibliothek von SML

5.1. Was sind Typen?

Ein Typ (oder Datentyp) ist eine Menge von Werten:

- Mit einem Typ werden Operationen (bzw. Prozeduren) zur Bearbeitung der Daten des Typs angeboten.
- Eine n-stellige Operation über einem Typ T ist eine Funktion $T^n \to T$, wobei T^n für $T \times \ldots \times T$ (n Mal) steht.

  Bsp.: die ganzzahlige Addition ist eine binäre Operation über dem Typ „ganze Zahl“.
- Ein Typ kann *vordefiniert* sein, d.h. von der Programmiersprache als Wertemenge angeboten werden.

  Mit einem vordefinierten Typ bieten Programmiersprachen die Operationen, Funktionen oder Prozeduren an, die zur Bearbeitung von Daten des Typs üblich sind.
- Moderne Programmiersprachen ermöglichen es, dass man selbst Typen definiert kann.

Selbstdefinierte Datentypen: Beispiele

- ein Typ „Wochentag“ mit Wertemenge

$$\{\text{Montag}, \text{Dienstag}, \ldots, \text{Sonntag}\}$$

- ein Typ „Uhrzeit“ mit einer Wertemenge

$$\{h : m : s \mid h \in \mathbb{N}, 0 \leq h < 24, m \in \mathbb{N}, 0 \leq m < 60, s \in \mathbb{N}, 0 \leq s < 60\}$$

- ein Typ „komplexe Zahl“ mit Wertemenge

$$\{a + ib \mid a \text{ und } b \text{ vom Typ „reelle Zahlen“ }\}$$

Selbstdefinierte Datentypen: Beispiele (II)

- ein Typ „Übungsgruppe“ zur Bündelung folgender Merkmale einer Übungsgruppe:
  - Der Name des Übungsleiters (als Zeichenfolge dargestellt)
  - Die Namen der in dieser Übungsgruppe angemeldeten Studenten (als Zeichenfolgen dargestellt)
  - Der wöchentliche Termin der Übung (als Paar aus Wochentag und Uhrzeit dargestellt)
  - Der Ort, wo die Übungsgruppe sich trifft (als Zeichenfolge dargestellt)

$\rightsquigarrow$ Spezifikation eigener Typen verlangt Mittel zur Zusammensetzung von Typen

In diesem Kapitel:

- die vordefinierten Typen von SML (teilw. Wiederholung!)
- „einfache“ Mechanismen zum Zusammensetzen von Typen
  [weitere Mechanismen dann in Kapitel 8 und 11]

5.2. Die Basistypen von SML

- ganze Zahlen (`int`)
- reelle Zahlen (`real`)
- Boole'sche Werte (`bool`)
- Zeichenfolgen (`string`)
- Zeichen (`char`)
- der Typ `unit`

5.2.1. Ganze Zahlen

Der SML-Typ `int` (integer) steht für die „ganzen Zahlen" ($\mathbb{Z}$).

- Das Vorzeichen negativer ganzen Zahlen wird in SML `~` geschrieben: z.B. `~89`.
- Führende Nullen sind in SML erlaubt:
  z.B. `007`, `089`, `~002`.
- Über dem Typ `int` bietet SML die folgenden binären Operationen an:

`+`	(infix)	Addition
`-`	(infix)	Subtraktion
`*`	(infix)	Multiplikation
`div`	(infix)	ganzzahlige Division
`mod`	(infix)	Rest der ganzzahligen Division

Ganze Zahlen (II)

- SML bietet über dem Typ `int` die folgenden Vergleichsoperatoren an:

=	(infix)	gleich
<>	(infix)	nicht gleich
<	(infix)	echt kleiner
<=	(infix)	kleiner-gleich
>	(infix)	echt größer
>=	(infix)	größer-gleich

  Diese Vergleichsoperatoren sind Funktionen vom Typ $\texttt{int} \times \texttt{int} \to \texttt{bool}$.

- Die Funktion `real` vom Typ $\texttt{int} \to \texttt{real}$ konvertiert eine ganze Zahl in eine Gleitkommazahl mit demselben mathematischen Wert.

5.2.2. Reelle Zahlen

Der SML-Typ `real` bezeichnet die *Gleitkommazahlen*:

- Zumeist wird auch (inkorrekterweise!) von „reellen Zahlen“ ($\mathbb{R}$) gesprochen.

  Tatsächlich entspricht `real` einer *endlichen* Teilmenge der rationalen Zahlen ($\mathbb{Q}$), wobei die Arithmetik von den erwarteten mathematischen Gesetzmäßigkeiten abweichen kann.

- Zur Darstellung von Gleitkommazahlen in SML können zwei Konstrukte (zusammen oder nicht zusammen) verwendet werden:
  - Der Punkt zur Darstellung von Dezimalbruchzahlen:

    z.B. `31.234`, `123.0`, `012.0`, `~2.459`
  - Die Mantisse-Exponent-Notation (E-Notation) zur Darstellung von Zehnerpotenzen:

    z.B. `123E5`, `123.0E~3`, `123.0e~3`, `~0.899e4`

  In der E-Notation ist sowohl ein großes „`E`“ als auch ein kleines „`e`“ erlaubt.

Reelle Zahlen (II)

- Führende Nullen in Dezimalbruchzahlen sowie in Mantissen und Zehnerexponenten sind erlaubt.

  Vor dem Punkt einer Dezimalbruchzahl verlangt SML aber eine Ziffer:

  die Schreibweise `.89` ist in SML nicht zulässig!

- Über dem Typ `real` bietet SML die folgenden binären Operationen an:

`+`	(infix)	Addition
`-`	(infix)	Subtraktion
`*`	(infix)	Multiplikation
`/`	(infix)	Division

Achtung: die Arithmetik mit Gleitkommazahlen hat ihre eigenen Gesetzmäßigkeiten!

Arithmetische Berechnungen mit Gleitkommazahlen ergeben oft nur Approximationen der tatsächlichen Werte.

Reelle Zahlen (III)

- Arithmetik mit Gleitkommazahlen; Beispiel:

```
- 1234567890.0 + 0.005;
val it = 1234567890.01 : real

- 1234567890.0 + 0.0005;
val it = 1234567890.0 : real
```

$\rightarrow$ Fehlerabschätzungsmethoden aus der numerischen Mathematik notwendig!

- SML bietet über dem Typ `real` die folgenden Vergleichsoperatoren an:

`<`	(infix)	echt kleiner	`>`	(infix)	echt größer
`<=`	(infix)	kleiner-gleich	`>=`	(infix)	größer-gleich

- Konvertierungsfunktionen vom Typ `real` $\rightarrow$ `int`:

`floor` — rundet „nach unten“ ab
`ceil` — rundet „nach oben“
`trunc` — rundet durch Weglassen der Nachkommastellen

Reelle Zahlen (IV)

Der SML-Typ `real` umfasst noch zwei spezielle Werte, die als Ergebnis bestimmter Operationen auftreten können:

```
- 1.0 / 0.0;
val it = inf : real

- 0.0 / 0.0;
val it = nan : real

- Math.sqrt(~1.0);
val it = nan : real

- 1.0 + (1.0 / 0.0);
val it = inf : real

- 1.0 + (0.0 / 0.0);
val it = nan : real
```

Reelle Zahlen (V)

`inf`	steht für	*infinite* (unendlich)
`nan`	steht für	*not-a-number* (keine Zahl)

- Diese Werte werden von allen Operationen des Typs `real` als Argument akzeptiert.

- Die Einzelheiten dieser Definitionen folgen einem internationalen Standard für Gleitkommazahlen in Programmiersprachen:

  IEEE standard 754-1985 und ANSI/IEEE standard 854-1987.

5.2.3. Boole'sche Werte

Der SML-Typ `bool` (Boole'sche Werte) besteht aus der Wertemenge {`true`, `false`}.

Über dem Typ `bool` bietet SML die folgenden Operatoren an:

`not`	(präfix, unär)	Negation
`andalso`	(infix, binär)	Konjunktion
`orelse`	(infix, binär)	Disjunktion

Die Operatoren `andalso` und `orelse` sind in SML keine Funktionen:

- Während der Auswertung von `A1 andalso A2` wird zunächst nur `A1` ausgewertet.
  Hat `A1` den Wert `false`, so wird `A2` nicht ausgewertet (und `false` als Wert von `A1 andalso A2` geliefert).
- Während der Auswertung von `A1 orelse A2` wird zunächst nur `A1` ausgewertet.
  Hat `A1` den Wert `true`, so wird `A2` nicht ausgewertet (und `true` als Wert von `A1 andalso A2` geliefert).

5.2.4. Zeichenfolgen

Der SML-Typ `string` ist die Menge der endlichen Zeichenfolgen.

In SML werden Zeichenfolgen eingeklammert zwischen zwei `"` geschrieben.

`""` bezeichnet in SML die leere Zeichenfolge.

Das Zeichen `"` wird in SML innerhalb einer Zeichenfolge `\"` geschrieben:

z.B `"ab\"cd"` bezeichnet in SML die Zeichenfolge `ab"cd`.

Weitere „escape sequences", die mit dem Zeichen \ anfangen, dienen zur Darstellung von Sonderzeichen in SML:

`\n` : newline
`\t` : tab
`\\` : \

Zeichenfolgen (II)

Die Folge:

\ gefolgt von *white-space*–Zeichen gefolgt von \

ermöglicht es, sogenannte *white-space*–Zeichen wie newline, tab oder Leerzeichen, die zur lesbareren Darstellung eines Programms nützlich sind, innerhalb einer SML-Zeichenfolge zu ignorieren: z.B.

```
- "aaaa\   \b";
val it = "aaaab" : string

- "ccc\
=
=
= \d";
val it = "cccd" : string
```

Zeichenfolgen (III)

Über dem Typ string bietet SML die folgenden Operationen an:

`size`	(präfix, unär)	Länge einer Zeichenfolge
`^`	(infix, binär)	Konkatenation (Aneinanderfügen) zweier Zeichenfolgen

SML bietet über dem Typ `string` die folgenden Vergleichsoperatoren an:

=	(infix)	gleich
<>	(infix)	nicht gleich
<	(infix)	echt kleiner
<=	(infix)	kleiner-gleich
>	(infix)	echt größer
>=	(infix)	größer-gleich

Basis ist die sogenannte *lexikographische Ordnung*: z.B. `"a" < "aa" < "b"`

5.2.5. Zeichen

Der SML-Typ `char` besteht aus der Menge der Zeichen.

- Zeichen und Zeichenfolgen der Länge 1 sind verschiedene Dinge.
  [genauso wie die Menge $\{2\}$ und die ganze Zahl 2]
- Im SML wird ein Zeichen `z` als `#"z"` geschrieben (`#` gefolgt von der Zeichenfolge `"z"`).
- Über dem Typ `char` bietet SML die folgenden Funktionen an:

$\texttt{chr} : \texttt{int} \to \texttt{char}$	für $0 \leq \texttt{n} \leq 255$ liefert `chr(n)` das Zeichen mit Code `n`;
$\texttt{ord} : \texttt{char} \to \texttt{int}$	liefert den Code eines Zeichens `z`

Zeichen (II)

Zum Beispiel:

```
- chr(100);
val it = #"d" : char

- ord(#"d");
val it = 100 : int

-  chr(ord(#"d"));
val it = #"d" : char

- ord(chr(89));
val it = 89 : int
```

Die Kodierung von Zeichen, d.h. der Zuordnung numerischer Codes zu Zeichen des Datentyps `char`, basiert auf dem ASCII-Code (American Standard Code for Information Interchange).

ASCII-Code / ASCII-Zeichensatz

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

ASCII-Code / ASCII-Zeichensatz (II)

Erläuterung der *Steuerzeichen*:

NUL	= Null,	SOH	= Start of heading,	STX	= Start of text,
ETX	= end of text,	EOT	= end of transmission,	ENQ	= enquiry,
ACK	= acknowledge,	BEL	= bell,	BS	= backspace,
HT	= horizontal tab,	LF	= line feed,	VT	= vertical tab,
FF	= form feed,	CR	= carriage return,	SO	= shift out,
SI	= shift in,	DLE	= data link escape,	DC1	= device control 1,
DC2	= device control 2,	DC3	= device control 3,	DC4	= device control 4,
NAK	= negative acknowledge,	SYN	= synchronous idle,	ETB	= end of transmission block,
CAN	= cancel,	EM	= end of medium,	SUB	= substitute,
ESC	= escape,	FS	= file separator,	GS	= group separator,
RS	= record separator,	US	= unit separator,	DEL	= delete.

Zeichen (III)

Konvertierungsfunktionen zwischen Zeichen und Zeichenfolgen:

`str` (unär) und `String.sub` (binär; aus der SML-Standardbibliothek)

```
- str(#"a");
val it = "a" : string

-  String.sub("abcd", 0);
val it = #"a" : char

 - String.sub("abcd", 2);
 val it = #"c" : char
```

5.2.6. `unit`

- Der SML-Typ `unit` besteht aus der Wertemenge $\{()\}$.
- `()` wird oft *unity* ausgesprochen.
- Dieser einzige Wert des Typs `unit` wird als Wert von Prozeduraufrufen verwendet.

5.3. Zusammengesetzte Typen in SML

- Vektoren (Tupel)
- Verbunde (Records)
- Vektoren als Verbunde

5.3.1. Vektoren (Tupel)

$(\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_\mathtt{n})$ ist ein $\mathtt{n}$-stelliger Vektor ($\mathtt{n}$-Tupel) vom Typ $\mathtt{t}_1 * \mathtt{t}_2 * \ldots * \mathtt{t}_\mathtt{n}$:

- wenn $\mathtt{n} \geq 0$, $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_\mathtt{n}$ SML-Typen und $\mathtt{A}_1, \mathtt{A}_2, \ldots \mathtt{A}_\mathtt{n}$ Ausdrücke der Typen $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_\mathtt{n}$
  (dabei bezeichnet „*“ in SML das kartesische Produkt von Typen).

Zum Beispiel:

```
- ("abc", 44, 89e~2);
val it = ("abc",44,0.89) : string * int * real

- (("abc", 44), (44, 89e~2));
val it = (("abc",44),(44,0.89)) : (string * int) * (int * real)
```

Man beachte, dass Komponenten von Vektoren selbst Vektoren sein dürfen.

Vektoren (Tupel) (II)

Die Gleichheit über Vektoren (derselben Länge!) ist komponentenweise definiert:

```
- val eins = 1;
val eins = 1 : int

- val drei = 3;
val drei = 3 : int

- (eins, drei) = (1, 3);
val it = true : bool
```

Vektoren der Längen 1 und 0 stellen Ausnahmen dar:

- Ein einstelliger Vektor ist in SML identisch mit seiner (einzigen) Komponente:

  ```
  - (3) = 3;
  val it = true : bool
  ```

- Der 0-stellige Vektor `()` ist der (einzige) Wert des SML-Typs `unit`.

Vektoren (Tupel) (III)

In SML hängen Vektoren und Argumente von „mehrstelligen“ Funktionen wie folgt zusammen:

In einer Funktionsanwendung $\texttt{f}(\texttt{a}_1, \texttt{a}_2, \texttt{a}_3)$ stellt $(\texttt{a}_1, \texttt{a}_2, \texttt{a}_3)$ einen Vektor dar, so dass die Funktion `f` einstellig ist (und auf dreistellige Vektoren angewandt wird); z.B.:

```
- fun f(n:int, m:int) = n + m;
val f = fn : int * int -> int

- val paar = (1, 2);
val paar = (1,2) : int * int

- f paar;
val it = 3 : int
```

In SML sind also jede Funktion und der Wert eines jeden Ausdrucks einstellig.

Vektoren (Tupel) (IV)

Pattern Matching zur Selektion der Komponenten eines Vektors in SML:

```
- val tripel = (1, #"z", "abc");
val tripel = (1,#"z","abc") : int * char * string

 - val (komponente1, komponente2, komponente3) = tripel;
 val komponente1 = 1 : int
 val komponente2 = #"z" : char
 val komponente3 = "abc" : string
```

Selektion der Komponenten eines Vektors mit den Funktionen #1, #2, usw.:

```
- #1("a", "b", "c");
val it = "a" : string

- #3("a", "b", "c");
val it = "c" : string
```

5.3.2. Deklaration eines Vektortyps

Einem Vektortyp $\mathtt{t}_1 * \mathtt{t}_2 * \ldots * \mathtt{t}_n$ kann wie folgt ein Name gegeben werden:

```
- type punkt = real * real;

- fun abstand(p1: punkt, p2: punkt) =
        let fun quadrat(z) = z * z
            val delta_x = #1(p2) - #1(p1)
            val delta_y = #2(p2) - #2(p1)
        in
            Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
        end;
val abstand = fn : punkt * punkt -> real

- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Deklaration eines Vektortyps (II)

Man beachte, dass `punkt` lediglich ein Synonym für `real * real` ist.

In der Tat ist `(real * real) * (real * real)` der Typ des aktuellen Parameters der vorangehenden Funktionsanwendung (wie gesagt ist die Funktion einstellig, und ihr Argument ist somit ein Paar von Paaren von Gleitkommazahlen).

Wegen der Typ-Constraints `p1: punkt` und `p2: punkt` verlangt die Definition der lokalen Funktion `quadrat` keine Typ-Constraints, um die Überladung der Multiplikation aufzulösen.

5.3.3. Verbunde (Records)

Ein n-stelliger Vektor ist eine geordnete Zusammensetzung von n Komponenten, so dass die Komponenten durch ihre Position bestimmt werden.

Ein dreistelliger Vektor wie `("Bry", #"F", 2210)` vom Typ `string * char * int` kann als eine Zusammensetzung dreier Komponenten auch so beschrieben werden:

- die ganze Zahl hat die Position 3,
- die Zeichenfolge hat die Position 1,
- das Zeichen hat die Position 2.

Es bietet sich an, anstelle von Positionen (Zahlen) Bezeichner zu verwenden, z.B.:

- Nachname
- Vornamenbuchstabe
- Durchwahl

Verbunde (Records) (II)

Diese Idee liegt den *Verbunden* (oder *records*) zu Grunde; in SML:

```
- val adressbucheintrag = {Nachname            = "Bry",
                           Vornamenbuchstabe = #"F",
                           Durchwahl           = "2210"};
val adressbucheintrag =
  {Durchwahl="2210",Nachname="Bry",Vornamenbuchstabe=#"F"}
  : {Durchwahl:string, Nachname:string, Vornamenbuchstabe:char}
```

- Die Komponenten eines Verbundes werden mit Bezeichnern statt mit Positionen identifiziert:

  $\rightsquigarrow$ Die Reihenfolge der Komponenten eines Verbunds spielt keine Rolle.

Verbunde (Records) (III)

- Die Bezeichner der Komponenten eines Verbundes sind Teil des Typs des Verbundes:

  $\leadsto$ Die Verbunde {`a = 1, b = 2`} und {`aaa = 1, bbb = 2`} haben nicht denselben Typ:

```
- {a = 1, b = 2};
val it = {a=1,b=2} : {a:int, b:int}

- {aaa = 1, bbb = 2};
val it = {aaa=1,bbb=2} : {aaa:int, bbb:int}
```

- Verbunde werden komponentenweise verglichen:

```
-  {a=1, b=2} = {b=2, a=1};
val it = true : bool
```

Verbunde (Records) (IV)

- Zur Selektion der Komponentenwerte eines Verbundes bietet SML die komponentenspezifischen Funktionen #*Bezeichner* an:

```
- #bbb({aaa=1,bbb=2});
val it = 2 : int

- #a({a = 1, b = 2});
val it = 1 : int
```

- SML bietet auch die folgende Kurzschreibweise für Deklarationen an; so werden Variablen mit denselben Namen wie die Komponenten deklariert:

```
- val info1dozent = {Nachname = "Bry", Vorname = "Francois"};
val info1dozent = {Nachname="Bry",Vorname="Francois"}
   : {Nachname:string, Vorname:string}

- val {Nachname, Vorname} = info1dozent;
val Nachname = "Bry" : string
val Vorname = "Francois" : string
```

Verbunde (Records) (V)

Das Folgende ist aber nicht möglich:

```
- val {nn, vn} = info1dozent;
 stdIn:1.1-39.11 Error: pattern and expression in val dec don't
 agree [tycon mismatch]
   pattern:    {nn:'Z, vn:'Y}
   expression:    {Nachname:string, Vorname:string}
   in declaration:
     {nn=nn,vn=vn} =
       (case info1dozent
         of {nn=nn,vn=vn} => (nn,vn))
```

Verbunde sind den „structures“ der Programmiersprache C und den „records“ der Programmiersprachen Pascal und Modula ähnlich.

5.3.4. Deklaration eines Vektor- oder Verbundstyps

Wie für Vektoren bietet SML die Möglichkeit an, einem Verbundtyp einen Namen zu geben, wie z.B.:

```
- type complex = real * real;

- type dateieintrag = {Vorname:string, Nachname:string};
```

Der so vergebene Name ist lediglich ein Synonym für den Verbundtyp.

5.3.5. Vektoren als Verbunde

In SML sind Vektoren Verbunde mit besonderen Komponentenbezeichnern, wie die folgende Sitzung zeigt:

```
- {1="abc", 2="def"};
val it = ("abc","def") : string * string

- fun vertauschen {1 = x:string, 2 = y:string} =  {1=y, 2=x};
val vertauschen = fn : string * string -> string * string

- val paar = ("abc", "def");
val paar = ("abc","def") : string * string

- vertauschen paar;
val it = ("def","abc") : string * string
```

5.4. Listen

- Der Begriff „Liste“ kommt in den meisten Programmiersprachen und in vielen Algorithmen vor:

  — mit einigen unwesentlichen Unterschieden vor allem in der Syntax.

- Wir wollen zunächst den Begriff „Liste“ unabhängig von jeglicher Programmiersprache erläutern.

- Danach werden wir den SML-Typ „Liste“ einführen.

5.4.1. Der Begriff „Liste“ in Algorithmenspezifikations- und Programmiersprachen

- Eine Liste ist eine endliche geordnete Folge von Elementen.
- Listen werden oft dargestellt als: `[1, 2, 3]` oder `["a", "bcd", "e", "fg"]`.
- Die leere Liste ist möglich: sie wird dargestellt als `[]`.
- Der Typ Liste verfügt über eine Funktion `cons` (für *list constructor*) um Listen wie folgt aufzubauen:
  - Angewandt auf einen Wert `W` und eine Liste `L` bildet `cons` die Liste, deren erstes (d.h. am weitesten links stehendes) Element `W` ist und deren weitere Elemente die Elemente der Liste `L` (in derselben Reihenfolge) sind.

  `cons(5, [9, 8])` (d.h. `cons` angewandt auf `5` und die Liste `[9, 8]`) bildet die Liste `[5, 9, 8]` als Wert.
- Die Funktion `cons` wird oft infix geschrieben.

  Man schreibt also `5 cons [9, 8]` statt `cons(5, [9, 8])`.

Der Begriff „Liste" . . . (II)

- Aus der Definition von `cons` folgt, dass eine Liste `[5, 9, 8]` auch so notiert werden kann:

  ```
  5 cons (9 cons (8 cons []))
  ```

  Ist zudem `cons` rechtsassoziativ (was in vielen Programmiersprachen der Fall ist), so kann eine Liste `[5, 9, 8]` auch wie folgt notiert werden:

  ```
  5 cons 9 cons 8 cons []
  ```

  `[5, 9, 8]` stellt oft „nur" syntaktischen Zucker für diese Form dar.

Der Begriff „Liste“ . . . (III)

- Der Typ Liste verfügt zudem über zwei Funktionen, mit dem auf Werte aus einer Liste zugegriffen werden kann — `head` und `tail`:
  - Angewandt auf eine nicht leere Liste `L` liefert `head` das erste (d.h. das am weitesten links stehende) Element von `L`.
  - Angewandt auf eine nicht leere Liste `L` liefert `tail` die Liste, die sich aus `L` ergibt, wenn das erste Element von `L` gestrichen wird.

  Bsp.: `head( [5, 9, 8] ) = 5` und `tail( [5, 9, 8] ) = [9, 8]`.

- Die Funktionen `head` und `tail` sind auf Listen nicht total, weil sie für die leere Liste nicht definiert sind.

- Weil sie ermöglichen, Listen zu zerlegen (*to decompose*), werden die Funktionen `head` und `tail` oft „decomposers“ genannt.

  [Manchmal auch „destructors“ — im Gegensatz zu „constructors“]

Der Begriff „Liste“ . . . (IV)

Wie kann man unter Verwendung von `head` und `tail` eine Funktion spezifizieren, die das letzte Element einer nichtleeren Liste liefert? $\leadsto$ Rekursion!

Das letzte Element `E` einer nichtleeren Liste `L` ist definiert als:

- Falls `L` die einelementige Liste `[A]` ist, so ist `A` das letzte Element.
- Andernfalls ist das letzte Element von `L` das letzte Element der Liste `tail(L)`.

Der Test, ob eine Liste nur ein Element enthält, lässt sich nun wie folgt spezifizieren:

- Eine Liste `L` enthält (genau) ein Element genau dann, wenn `tail(L) = []` ist.

Zusammengesetzte Ausdrücke/Werte als Listenelemente sind möglich:

z.B. Listen von Listen wie etwa `[[1,2],[1,5]]`

Die Gleichheit für Listen ist elementweise definiert:

`[a,b,c] = [d,e,f]` genau dann, wenn `a=d`, `b=e` und `c=f` gilt.

5.4.2. Die Listen in SML

- Eine SML-Liste ist eine endliche Folge von Werten desselben Typs.

  $\rightsquigarrow$ also nicht möglich: Listen von Werten mit verschiedenen Typen
- Für jeden gegebenen Typ `'a` (oft *alpha* ausgesprochen) bezeichnet `'a list` in SML den Typ der Listen von Werten vom Typ `'a`.

  Bsp.: Ist `'a` der Typ `int`, so ist `int list` der Typ „Listen von ganzen Zahlen“.

SML bietet zwei Notationen für Listen:

1. mit dem Listkonstruktor `cons` (in SML `::` geschrieben; rechtsassoziativ) und der leeren Liste (in SML `nil`):

   `0 :: (1 :: (2 :: (3 :: nil)))` kann auch als `0 :: 1 :: 2 :: 3 :: nil` geschrieben werden (wg. der Rechtsassoziativität von `::`)

```
- 0 :: 1 :: 2 :: 3 :: nil;
val it = [0,1,2,3] : int list

- 0 :: (1 :: (2 :: (3 :: nil)));
val it = [0,1,2,3] : int list
```

Die Listen in SML (II)

2. unter Verwendung der Listenklammern „[“ und „]“ mit Trennzeichen „,“ zwischen den Listenelementen:

   Bsp.: `[0,1,2,3]` und die leere Liste `[]`

```
- [0,1,2,3];
val it = [0,1,2,3] : int list
```

Selbstverständlich dürfen beide Notationen zusammen verwendet werden:

```
- 0 :: 1 :: [2, 3];
val it = [0,1,2,3] : int list
```

Die Notation mit den Listenklammern „[“ und „]“ und dem Trennzeichen „,“ ist lediglich „syntaktischer Zucker“, d.h. eine Kurzform für die Notation mit dem `::`.

Die Listen in SML (III)

Der SML-Typ der leeren Liste `nil` (oder `[]`) ist `'a list` („Liste von Elementen eines beliebigen Typs“):

```
- nil;
val it = [] : 'a list

- [];
val it = [] : 'a list
```

- Dabei ist `'a` (*alpha*) eine *Typvariable*, d.h. eine Variable, die als Wert einen Typ erhalten kann.

  `nil` ist ein *polymorphes Objekt*, d.h. ein „Objekt“, das mehreren Typen angehört.

- Da `nil` ein polymorphes Objekt ist, wird nur eine leere Liste benötigt.

  Wäre `nil` kein polymorphes Objekt, dann müsste für jeden möglichen Typ `'a` eine leere Liste (mit Typ `'a list`) geben, was ziemlich umständlich wäre.

Die Listen in SML (IV)

SML bietet die Gleichheit für Listen:

```
- val eins = 1;
val eins = 1 : int

- val zwei = 2;
val zwei = 2 : int

- [eins, 2] = [1,zwei];
val it = true : bool
```

5.4.3. Mono- und Polytypen

Ein Typausdruck wie `'a` oder `'a list` wird *polymorpher Typ* oder *Polytyp* genannt, weil der Ausdruck für mehrere (griechisch „poly“) Typen steht:

- Mögliche Instanzen von `'a` sind z.B. `int`, `bool` oder `int list`.
- Mögliche Instanzen von `'a list` sind z.B. `int list`, `bool list`, `(int list) list` oder `(int * bool) list`.

Ein Typ, der kein Polytyp ist, wird *Monotyp* genannt.

5.5. Beispiele: Grundlegende Listenfunktionen

- Länge einer Liste
- Letztes Element einer (nichtleeren) Liste
- Kleinstes Element einer (nichtleeren) Liste von ganzen Zahlen
- n-tes Elemente einer Liste
- `head`
- `tail`
- `append`
- `naive_reverse` und `reverse`

5.5.1. Länge einer Liste

```
- fun laenge(nil)    = 0
    | laenge(_ :: L) = 1 + laenge(L);
val laenge = fn : 'a list -> int

 - laenge([0,1,2,3]);
val it = 4 : int
```

5.5.2. Letztes Element einer nichtleeren Liste

```
- fun letztes_element(x :: nil) = x
    | letztes_element(_ :: L)   = letztes_element(L);
Warning: match nonexhaustive
          x :: nil => ...
          _ :: L   => ...
val letztes_element = fn : 'a list -> 'a

- letztes_element([0,1,2,3]);
  val it = 3 : int
```

- Die Deklaration der Funktion `letztes_element` hat keinen Fall für die leere Liste.
- Das SML-System erkennt dies und gibt eine Warnung aus, lehnt aber die Deklaration nicht ab.
- Die Warnung soll daran erinnern zu prüfen, ob nicht ein Fall vergessen wurde.

5.5.3. Kleinstes Element einer nichtleeren Liste von ganzen Zahlen

```
- fun kleinstes_element(x ::nil) = x : int
    | kleinstes_element(x :: L)  = let val y = kleinstes_element(L)
                                   in
                                       if x <= y then x else y
                                   end;
Warning: match nonexhaustive
           x :: nil => ...
           x :: L   => ...
val kleinstes_element = fn : int list -> int
```

Das Typ-Constraint `x : int` ist notwendig, weil der Boole'sche Operator <= überladen ist.

5.5.4. n-tes Element einer Liste

```
- fun ntes_element(1, x :: _) = x
    | ntes_element(n, _ :: L) = ntes_element(n-1, L);
```

Frage: Über welcher Menge ist diese Funktion total?

5.5.5. head

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a
```

SML bietet die vordefinierte Funktion `hd` an:

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

5.5.6. tail

```
- fun tail(_ :: L) = L;
Warning: match nonexhaustive
          _ :: L => ...
val tail = fn : 'a list -> 'a list
```

SML bietet die vordefinierte Funktion `tl` an:

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

5.5.7. append

Die vordefinierte SML-Funktion append, infix „@“ notiert, dient dazu, zwei Listen aneinander zu fügen:

```
- [1,2,3] @ [4,5];
val it = [1,2,3,4,5] : int list
```

Die Funktion append kann wie folgt in SML implementiert werden:

```
- fun append(nil,    L) = L
    | append(h :: t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list

- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

`append` (II)

Berechnungsschritte zur Auswertung von `append([1,2,3], [4,5])`:

```
append(1::(2::(3::nil)), 4::(5::nil))
1 :: append(2::(3::nil), 4::(5::nil))
1 :: (2 :: append(3::nil, 4::(5::nil)))
1 :: (2 :: (3 :: append(nil, 4::(5::nil))))
1 :: (2 :: (3 :: (4::(5::nil))))
```

Es gibt keinen weiteren Berechnungsschritt mehr:

`1 ::(2 ::(3 ::(4 ::(5 ::nil))))` ist die Liste `[1, 2, 3, 4, 5]`.

Zeitbedarf von `append`

Als Zeiteinheit kann die Anzahl der Aufrufe der Funktion `cons` (`::`) oder aber die Anzahl der rekursiven Aufrufe von `append` gewählt werden.

Beide Zahlen stehen einfach miteinander in Verbindung:

Wird zur Berechnung von `append(L, L')` die `append`–Funktion $n + 1$ mal rekursiv aufgerufen, so wird die Funktion `cons` (`::`) n mal aufgerufen.

(*) Um eine Liste `L` der Länge n mit $n \geq 1$ vor einer Liste `L'` einzufügen, ruft die Funktion `append` die Funktion `cons` (`::`) genau n-mal auf.

- Die Länge des zweiten Parameters `L'` beeinflusst nicht den Zeitbedarf von `append`.
- Ist n die Länge des ersten Parameters von `append`, so gilt: `append` $\in O(n)$.

5.5.8. `naive-reverse`

Mit der vordefinierten SML-Funktion `reverse` (`rev` notiert) kann aus einer Liste eine Liste in umgekehrter Reihenfolge erzeugt werden:

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

Eine Funktion `reverse` kann in SML wie folgt implementiert werden:

```
- fun naive_reverse(nil)    = nil
    | naive_reverse(h :: t) = append(naive_reverse(t), h :: nil);

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

naive-reverse (II)

Berechnungsschritte zur Auswertung von `naive_reverse([1,2,3])`:

```
naive_reverse(1::(2::(3::nil)))
append(naive_reverse(2::(3::nil)), 1::nil)
append(append(naive_reverse(3::nil), 2::nil), 1::nil)
append(append(append(naive_reverse(nil),3::nil),2::nil),1::nil)
append(append(append(nil,               3::nil),2::nil),1::nil)
append(append(3::nil,                          2::nil),1::nil)
append(3::append(nil, 2::nil),                          1::nil)
append(3::(2::nil),                                     1::nil)
3::append(2::nil,                                       1::nil)
3::(2::append(nil,                                      1::nil))
3::(2::(1::nil))
```

Zeitbedarf von `naive_reverse`

Zur Schätzung der Größe des Problems „Umkehrung einer Liste“ bietet es sich an, die Länge der Liste zu wählen.

Als Zeiteinheit wählen wir die Anzahl der rekursiven Aufrufe von `naive_reverse` oder die Anzahl der Aufrufe der Funktion `cons` (`::`). [wie bei `append`]

- Gegeben sei eine Liste `L` der Länge n mit $n \geq 1$.
- Während des Aufrufes von `naive_reverse(L)` wird die Funktion `naive_reverse` n-mal rekursiv aufgerufen:
  - zunächst mit einer Liste der Länge $n-1$ als Parameter,
  - dann bei jedem weiteren Aufrufmit einer jeweils um ein Element kürzeren Liste

  Wegen (*) (siehe Zeitbedarf von `append`) ergibt sich die Anzahl der Aufrufe der Funktion `cons` (`::`) zur Zerlegung der Eingabeliste als:

$$(n-1) + (n-2) + \ldots + 1$$

Zeitbedarf von `naive_reverse` (II)

Zum Aufbau der auszugebenden Liste wird `cons` (`::`) zudem n-mal aufgerufen. Die Gesamtanzahl der Aufrufe von `cons` (`::`) lautet also:

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n * (n+1)}{2}$$

Ist n die Länge des Parameters von `naive_reverse`, so gilt:

$$\texttt{naive\_reverse} \in O(\frac{n * (n+1)}{2})$$

also

$$\texttt{naive\_reverse} \in O(n^2)$$

5.5.9. `reverse`

Der quadratische Zeitbedarf von `naive_reverse` ist nicht zufriedenstellend. Gesucht ist ein Algorithmus mit möglichst linearen Zeitbedarf. Idee:

- Man fängt mit zwei Listen an: die linke Liste, die die Eingabeliste ist, und die rechte Liste, die anfangs leer ist.
- Nach und nach wird das erste Element der linken Liste von dieser Liste entfernt und am Anfang der rechten Liste eingefügt.
- Dabei werden nur Operationen verwendet, die der Typ Liste anbietet.
- Nach soviel Schritte, wie die Eingabeliste lang ist, ist die linke Liste leer und die rechte Liste die Liste in umgekehrten Reihenfolge, die aufzubauen war:

Schritt	Linke Liste	Rechte Liste
0	`[1, 2, 3]`	`[]`
1	`[2, 3]`	`[1]`
2	`[3]`	`[2, 1]`
3	`[]`	`[3, 2, 1]`

`reverse` (II)

Dieses Verfahren lässt sich einfach in SML wie folgt implementieren:

```
- fun aux_reverse(nil,  R) = R
    | aux_reverse(h::t, R) = aux_reverse(t, h::R);
 val aux_reverse = fn : 'a list * 'a list -> 'a list

- aux_reverse([1,2,3], []);
val it = [3,2,1] : int list
- aux_reverse([1,2], [8,9]);
val it = [2,1,8,9] : int list
```

Die gewünschte unäre `reverse`–Funktion folgt unmittelbar aus der Spezifikation von `aux_reverse`.

reverse (III)

```
- fun reverse(L) = let fun aux_reverse(nil, R) = R
                         | aux_reverse(h::t,R) = aux_reverse(t,h::R)
                   in
                       aux_reverse(L, nil)
                   end;
val reverse = fn : 'a list -> 'a list

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

Berechnungsschritte zur Auswertung von reverse([1,2,3]):

```
reverse(1::(2::(3::nil)))
aux_reverse(1::(2::(3::nil)),  nil)
aux_reverse(2::(3::nil),    1::nil)
aux_reverse(3::nil,     2::(1::nil))
aux_reverse(nil,    3::(2::(1::nil)))
3::(2::(1::nil)))
```

Zeitbedarf von `reverse`

Ist $n \geq 0$ die Länge einer Liste `L`, so bedarf die Auswertung von `aux_reverse(L, nil)` n rekursiver Aufrufe sowie n Aufrufe der Funktion `cons` (`::`). Es gilt also:

$$\texttt{aux\_reverse} \in O(n)$$

Folglich gilt auch:

$$\texttt{reverse} \in O(n)$$

- Der zweite Parameter der Funktion `aux_reverse`, der der rechten Liste aus unserem Beispiel entspricht, ist ein sogenannter *Akkumulator*.
- Wie die Funktion `fak_iter` (Abschnitt 4.3) ist die Funktion `reverse` endrekursiv.
- Dies ermöglicht einen iterativen Berechnungsprozess.

5.6. Hinweis auf die Standardbibliothek von SML

Die Standardbibliothek von SML

(siehe unter: http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/)

bietet für die Basistypen von SML Funktionen an, die herkömmliche Operationen über diesen Typen in SML implementieren.

6. Typprüfung

- 6.1 Die Typprüfung: Eine nützliche Abstraktion für die Entwicklung von korrekten Programmen
- 6.2 Statische versus dynamische Typprüfung
- 6.3 Die Polymorphie: Eine wünschenswerte Abstraktion
- 6.4 Polymorphie versus Überladung
- 6.5 Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
- 6.6 Typkonstruktor versus Wertkonstruktor
- 6.7 Schlussregeln für die Typinferenz
- 6.8 Der Unifikationsalgorithmus
- 6.9 Ein Verfahren zur automatischen Typinferenz

6.1. Die Typprüfung: Eine nützliche Abstraktion für die Entwicklung von korrekten Programmen

Die Typprüfung (*type checking*) umfasst zwei komplementäre Aufgaben:

1. Ermittlung der Typen von Ausdrücken aus gegebenen Typ-Constraints
   z.B. die Ermittlung der Typen von `x` (`int`) und `zweimal` (`int -> int`) in:

   ```
   - fun zweimal(x) = 2 * x;
   ```

2. Überprüfung der Typen, die die Typ-Constraints eines Programms angeben
   z.B. die Überprüfung der Korrektheit der angegebene Typen in:

   ```
   - fun zweimal'(x:int):real = 2.0 * x;
   ```

   [Typ-Constraints für `x` und der `real`–Konstante `2.0` sind nicht kompatibel]

Typen ermöglichen formale (aber nicht semantische) Korrektheitsprüfungen vor oder während der Ausführung von Programmen.

6.2. Statische versus dynamische Typprüfung

- Typprüfung zur Laufzeit:
  (d.h. wenn das Programm ausgeführt wird und Ausdrücke ausgewertet werden)

  $\rightsquigarrow$ *dynamische Typprüfung*

  Beispiele: die Programmiersprachen Lisp, Pascal und Smalltalk

  Die meisten *stark* (oder *streng*) *typisierten* Programmiersprachen führen eine dynamische Typprüfung durch.

- Typprüfung zur Übersetzungszeit:
  (, d.h. wenn das Programm in Maschinensprache umgewandelt wird, also bevor es ausgeführt wird)

  $\rightsquigarrow$ *statische Typprüfung*

  Beispiele: die Programmiersprachen SML, Miranda, C++ und Java

  Besonderheit von SML: es wird ausschließlich eine statische Typprüfung durchführt.

Statische versus dynamische Typprüfung (II)

- Eine Programmiersprache ist *dynamisch typisiert*, wennn sie eine dynamische Typprüfung durchführt.

- Eine Programmiersprache, die eine statische Typprüfung und keine dynamische Typprüfung durchführt, heißt *statisch typisiert*

- Eine Programmiersprachen ohne Typprüfung sind *nicht typisiert*.

Einige „moderne“ Programmiersprachen wie C sind nicht typisiert bzw. erlauben nur eine partielle Typprüfung — dies führt häufig zu Programmierfehlern!

Statische versus dynamische Typprüfung (III)

SML führt keine dynamische Typprüfung durch. Ausschließlich eine statische Typprüfung durchzuführen hat die folgenden Vorteile:

1. Die statische Typprüfung trägt zur frühen Erkennung von Programmier– oder Konzeptionsfehler schon während der Programmentwicklung bei.
2. Eine ausschließlich statische Typprüfung entlastet die Laufzeit von einer zeitaufwendigen Aufgabe und trägt zur Einfachheit des Übersetzers (bzw. des Auswerters) bei.
3. Bei statisch typisierten Programmiersprachen können keine typbedingten „Laufzeitfehler“ vorkommen, d.h. Fehler im Programm in der Maschinensprache, in das das ursprüngliche Programm übersetzt wird.

   [Dies setzt natürlich voraus, dass der Übersetzer selbst fehlerfrei ist.]

Statische versus dynamische Typprüfung (IV)

- In der Praxis ist die statische Typprüfung eine sehr große Hilfe zur Entwicklung von fehlerfreien Programmen.

  Es ist zu erwarten, dass die statische Typprüfung sich unter den zukünftigen industriellen Programmiersprachen verbreiten wird.
- Die Typprüfung beruht immer auf denselben Techniken — unabhängig von Programmiersprache und Berechnungsmodell (funktional, imperativ, logisch . . .).
- Viele dieser Techniken sind in Zusammenhang mit SML entstanden und daher in SML in besonders ausgereifter Form realisiert.
- SML ist eine sogenannte „polymorphe“ Programmiersprache:

  Die Polymorphie benötigt fortgeschrittene Techniken zur Typprüfung, wie die Unifikation.

6.3. Die Polymorphie: Eine wünschenswerte Abstraktion

- Polymorphe Funktionen, Konstanten und Typen
- Typen von Vorkommen eines polymorphen Ausdrucks
- Vorteile der Polymorphie

6.3.1. Polymorphe Funktionen, Konstanten und Typen

Für den Algorithmus zum Aneinanderhängen zweier Listen spielt der Typ der Listenelemente eigentlich keine Rolle.

Dieser Algorithmus kann z.B. durch die Funktion `append` implementiert werden:

```
- fun append(nil, l) = l
    | append(h :: t, l) = h :: append(t, l);
val append = fn : 'a list * 'a list -> 'a list
```

Polymorphe Funktionen, Konstanten und Typen (II)

Die SML-Funktion `append` ist auf Listen von Objekten beliebigen Typs anwendbar:

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list

- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it = [#"a",#"b",#"c",#"d",#"e"] : char list

- append([10e~1,20e~1],[30e~1]);
val it = [1.0,2.0,3.0] : real list

- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]] : int list list
```

- Eine Funktion oder Prozedur, die wie die Funktion `append` auf aktuellen Parametern unterschiedlicher Typen anwendbar ist, heißt *polymorph*.
- Diese Bezeichnung bezieht sich auf die Parameter, die unterschiedlicher (griechisch *poly*) Gestalt (griechisch *morph*) sein können.

Polymorphe Funktionen, Konstanten und Typen (III)

- Den Typ einer polymorphen Funktion nennt man einen *polymorphen Typ* oder kurz *Polytyp*.

  Beispiele von polymorphen Typen sind in der SML-Syntax:

`'a`	irgendein Typ
`'a list`	Liste von Objekten eines beliebigen Typs (aber allen von demselben Typ)

- Nicht nur Funktionen und Typen sondern auch beliebige Ausdrücke können polymorph sein:

  Beispiel: die leere Liste `nil` (auch `[]`) ist eine polymorphe Konstante.

- Eine Programmiersprache, die wie SML polymorphe Funktionen und Prozeduren ermöglicht, wird *polymorph* genannt.
- Man sagt auch, dass sie die *Polymorphie* anbietet.

6.3.2. Typen von Vorkommen eines polymorphen Ausdrucks

Ist ein Ausdruck polymorph, so dürfen unterschiedlichen Vorkommen dieses Ausdrucks im selben Programm unterschiedliche Typen erhalten.

Wenn z.B. die polymorphe Funktion `append` auf Listen von ganzen Zahlen angewendet wird, so erhält die polymorphe Konstante `nil` im Rumpf von `append` den Typ `int list`.

Ein weiteres Beispiel einer polymorphen Funktion ist die Identitätsfunktion:

```
- val id = fn x => x;
val id = fn : 'a -> 'a
```

Im Ausdruck `id(id)(2)` erhält das innere Vorkommen von `id` den Typ `int -> int`, das äußere den Typ `('a -> 'a) -> ('a -> 'a)`:

```
- id(id)(2);
val it = 2 : int
```

6.3.3. Vorteile der Polymorphie

Nicht alle Programmiersprachen sind polymorph.

Viele Programmiersprachen wie z.B. Pascal und Basic verlangen, dass für jede Art von Listen eine spezielle `append`-Funktion programmiert wird.

Das führt zu folgenden Problemen:

1. Die Nichtpolymorphie vergrößert die Programme unnötig und trägt damit dazu bei, sie unübersichtlich zu machen.
2. Die Nichtpolymorphie erschwert die Wartung von Programmen, weil derselbe Algorithmus in verschiedenen Prozeduren verbessert werden muss.

$\leadsto$ Polymorphie stellt also eine wünschenswerte Abstraktion dar!

Die Polymorphie ist eine Verbesserung von Programmiersprachen, die erst in den 80er Jahren vorgeschlagen wurde (im Zusammenhang mit SML).

6.4. Polymorphie versus Überladung

Wie die Funktion `append` können einige vordefinierte Funktionen von SML wie z.B. die Addition auf aktuelle Parameter von unterschiedlichen Typen angewandt werden:

```
- 2 + 5;
val it = 7 : int

- 2.1 + 4.2;
val it = 6.3 : real
```

Diese Eigenschaft der Addition hatten wir als „Überladung“ bezeichnet.

Sind Überladung und Polymorphie prinzipiell unterschiedliche Konzepte?

Polymorphie versus Überladung (II)

Addition

Bei der Addition handelt es sich um die Verwendung desselben Namens (Bezeichner) zum Aufruf von verschiedenen (System-)Prozeduren:

- der Prozedur zur Addition von ganzen Zahlen einerseits,
- der Prozedur zur Addition von Gleitkommazahlen andererseits.

Funktion `append`

Bei der Funktion `append` handelt es sich um die Verwendung derselben Prozedur, folglich auch desselben Namens, mit Aufrufparametern mit unterschiedlichen Typen.

Polymorphie versus Überladung (III)

Polymorphie und Überladung sind völlig unterschiedlich:

- Polymorphie bezeichnet die Verwendung derselben Prozedur mit Aufrufparametern mit unterschiedlichen Typen.
- Überladung bezeichnet die Verwendung desselben Namens zur Bezeichnung von verschiedenen Prozeduren.

Schon FORTRAN (eine der ältesten Programmiersprachen) kannte die Überladung; Polymorphie wurde erst später für stark typisierte Programmiersprachen entwickelt.

Untypisierte Programmiersprachen benötigen keine Polymorphie — die automatische Typanpassung ermöglicht in den vielen praktischen Fällen einen vergleichbaren Effekt.

ad hoc– und parametrische Polymorphie

Überladung wird auch „*ad hoc*–Polymorphie“ genannt. Dann spricht man von „parametrischer Polymorphie“ für das, was hier „Polymorphie“ genannt wird.

6.5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML

- Typvariablen
- Typinferenz
- Typausdrücke
- Typkonstanten
- Typ-Constraints
- zusamengesetzte Typausdrücke und Typkonstruktoren
- die ''–Typvariablen

6.5.1. Typvariablen

Der Typ der polymorphen Funktion `append` lautet:

```
'a list * 'a list -> 'a list
```

wie das SML-System nach der Auswertung der Deklaration von `append` mitteilt:

```
- fun append(nil, l) = l
    | append(h :: t, l) = h :: append(t, l);
val append = fn : 'a list * 'a list -> 'a list
```

Dabei ist `'a` (oft *alpha* gesprochen) eine „Typvariable“. Die Polymorphie der Funktion `append` macht es nötig, dass eine Variable im Typ dieser Funktion vorkommt.

- Wird die Funktion `append` auf aktuelle Parameter eines Typs `t` angewandt, so wird die Typvariable `'a` an `t` gebunden.
- Dies bestimmt den aktuellen Typ der polymorphen Funktion `append` in der gegebenen Funktionsanwendung.

$\rightsquigarrow$ *Typinferenz*

6.5.2. Typinferenz

Die Typinferenz ist die Schlussfolgerung, durch die der Typ eines Ausdrucks ermittelt wird oder die Erfüllung der Typ-Constraints eines Programms überprüft wird.

Beispiel 1:

```
- fun zweimal(x) = 2 * x;
```

Der Typ der Funktion `zweimal` kann wie folgt ermittelt werden:

Da 2 eine ganze Zahl ist, steht der überladene Name `*` für die Multiplikation zweier ganzen Zahlen.

- Folglich hat `x` den Typ `int`.
- Daraus folgt der Typ `int -> int` der Funktion `zweimal`.

Typinferenz (II)

Beispiel 2:

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list
```

- Die aktuellen Parameter der Funktionsanwendung haben den Typ `int list`.
- Aus dem polymorphen Typ `'a list * 'a list -> 'a list` von `append` und dem Typ `int list` der aktuellen Parameter der Funktionsanwendung folgt der aktuelle Typ von `append` in der Funktionsanwendung:

```
int list * int list -> int list
```

Typinferenz (III)

Beispiel 3:

```
- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it = [#"a",#"b",#"c",#"d",#"e"] : char list
```

- Hier wird `'a` an `char` gebunden.
- Aus dem polymorphen Typ `'a list * 'a list -> 'a list` folgt der aktuelle Typ von `append` in der Funktionsanwendung:

```
char list * char list -> char list
```

Typinferenz (IV)

Beispiel 4:

```
- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]] : int list list
```

- Hier wird die Typvariable `'a` an den Typ `int list` gebunden.
- Der aktueller Typ von `append` in der Funktionsanwendung ist damit:

```
int list list * int list list -> int list list
```

Da der Postfix-Operator `list` linksassoziativ ist und der Operator `*` (Kartesisches Produkt) stärker bindet als ->, steht der obige Typausdruck für:

```
((int list) list * (int list) list) -> ((int list) list)
```

6.5.3. Typausdrücke

In den betrachteten Beispielen kommen „Typausdrücke“ vor.

Es muss also einen Formalismus geben, mit dem Typausdrücke gebildet werden können und der damit festlegt, welches korrekte bzw. zulässige Typausdrücke sind.

Typvariablen sind herkömmlichen Variablen ähnlich. Sie werden an Typausdrücke gebunden.

Ein Typausdruck ist entweder

- *atomar*, wie z.B. `int` und `char`, oder
- *zusammengesetzt*, wie z.B. `int list` oder `(int list) list`.

6.5.4. Typkonstanten

- Atomare Typausdrücke, die keine Typvariablen sind, heißen „Typkonstanten“.

- Beispiele für Typkonstanten sind:

  `int`, `real`, `bool`, `string`, `char` und `unit`.

- Typkonstanten bezeichnen Typen, die nicht zusammengesetzt sind.

  [siehe Kapitel 8 für die Definition neuer Datentypen]

6.5.5. Typ-Constraints

Ein Typ-Constraint ist ein Ausdruck der Gestalt:

```
Ausdruck : Typausdruck
```

wie z.B.:

```
x : int
l : char list
(fn x => x * x) : int -> int
```

Typ-Constraints werden auch *Typisierungsausdrücke* genannt.

6.5.6. Zusammengesetzte Typausdrücke und Typkonstruktoren

Zusammengesetzte Typausdrücke werden ähnlich wie Funktionsanwendungen, oft mit Postfix- oder Infix-Operatoren (den *Typkonstruktoren*), gebildet, z.B.:

```
'a * 'a
int list
int -> int
int list * int list -> int list
{ Vorname:string, Nachname:string }
```

Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionsnamen. Typkonstruktoren werden aber anders als Funktionsnamen verwendet:

- Wird eine Funktion wie `append` auf die Listen `[1,2]` und `[3]` angewandt, so geschieht dies, damit die Funktionsanwendung `append([1,2],[3])` ausgewertet wird, d.h. damit der Wert `[1,2,3]` berechnet wird.
- Wird der Typkonstruktor `*` auf die Typausdrücke `int` und `'a list` angewandt, so geschieht dies lediglich, damit der Typausdruck `(int * 'a list)` gebildet wird.

Zusammengesetzte Typausdrücke und Typkonstruktoren (II)

Die Anwendung eines Typkonstruktors auf „Parameter“ führt zu keiner Auswertung.

Eine solche Auswertung könnte in der Regel nicht berechnet werden, z.B.:

- Die Anwendung von `*` (Kartesisches Produkt) auf `int` und `bool` bezeichnet die Menge aller Paare (n, w) mit einer ganzen Zahl n und einem Boole'schen Wert w.
- Mathematisch gesehen bildet die Anwendung von `*` (Kartesisches Produkt) die Typen `int` (also $\mathbb{Z}$) und `bool` (also $\{true, false\}$) auf die folgende Menge ab:

  $$\{\ (n, w) \mid n \in \mathbb{Z} \text{ und } w \in \{true, false\}\ \}$$

  Diese Menge kann nicht „ausgerechnet“ werden, weil sie unendlich ist.

In anderen Fällen (etwa `bool * bool`) wäre die Berechnung des zusammengesetzten Typs möglich, weil er endlich ist.

Typen werden dazu verwendet, Programmierfehler zu vermeiden, aber nicht um alle möglichen Werte zu berechnen.

6.5.7. Die `''`-Typvariablen zur Polymorphie für Typen mit Gleichheit

Die Gleichheitsfunktion = ist überladen, weil dasselbe Symbol = für viele verschiedene Typen wie etwa `bool`, `int`, die polymorphen Typen List, Vektor und Verbund verwendet werden kann.

Viele Algorithmen, für die eine Implementierung als polymorphe Funktion nahe liegt, beziehen sich auf die Gleichheit.

- Damit die Spezifikation von solchen Funktionen in SML möglich ist, bietet SML die `''`-Typvariablen.

  Eine `''`-Typvariable wird `''Name` geschrieben.

- `''`-Typvariablen stehen immer für Typen mit Gleichheit.

Die ''-Typvariablen . . . (II)

Ein Beispiel der Verwendung von ''-Typvariablen ist das polymorphe Prädikat `member` zum Testen, ob ein Element in einer Liste vorkommt:

```
- fun member(x, nil)          = false
    | member(x, head::tail) = if x = head
                               then true
                               else member(x, tail);
val member = fn : ''a * ''a list -> bool

- member(3,[1,2,3,4]);
val it = true : bool

- member(#"c",[#"a",#"b",#"c",#"d"]);
val it = true : bool

- member([1,2],[[1,2,3],[1,2],[1,2,3,4]]);
val it = true : bool
```

Die ''-Typvariablen . . . (III)

''-Typvariablen können nur an Typausdrücke gebunden werden, die Typen mit Gleichheit bezeichnen:

```
- member((fn x => x), [(fn x => x)]);
Error: operator and operand don't agree [equality type required]
   operator domain: ''Z * ''Z list
   operand:         ('Y -> 'Y) * ('X -> 'X) list
   in expression:
      member ((fn x => x),(fn <pat> => <exp>) :: nil)
```

Die Gleichheit ist auf dem Typ `'x -> 'x` nicht definiert.

6.6. Typkonstruktor versus Wertkonstruktor

Typkonstruktoren (z.B. `list`) und (Wert-)Konstruktoren (z.B. `cons` bzw. `::`) dürfen nicht verwechselt werden:

- Mit `list` und einem Typausdruck wie z.B. `'a` oder `int * bool` werden der polymorphe Listentyp `'a list` und der Listentyp `(int * bool) list` gebildet.

  `list` und `*` sind in diesen Beispielen Typkonstruktoren.

- Mit dem (Wert-)Konstruktor `cons` (`::`) und einem (herkömmlichen) Ausdruck wie z.B. 1, `"abc"` oder `(3, false)` werden wie folgt Listen gebildet:

```
- 1 :: [];
val it = [1] : int list

- "abc" :: [];
val it = ["abc"] : string list

- (3, false) :: [];
val it = [(3,false)] : (int * bool) list
```

Typkonstruktor versus Wertkonstruktor (II)

- Die Unterscheidung gilt für alle zusammengesetzten Typen.
- Typkonstruktoren und (Wert-) Konstruktoren können wie folgt einander gegenübergestellt werden (die Argumente sind durch `.` dargestellt):

Typkonstruktor	**(Wert-)Konstruktor**
`. list`	`. :: .`
	`nil`
`. * .`	`( . , . )`
`{ . : . , . : . }`	`{ . = . , . = . }`
`. -> .`	`fn . => .`

- Der allgemeine Fall ist, dass es zu *einem* Typkonstruktor *mehrere*, aber endlich viele (Wert-)Konstruktoren des Typs geben kann.

  Zum Beispiel sind `cons` (`::`) und `nil` die beiden (Wert-)Konstruktoren eines Listentyps. `nil` ist ein 0-stelliger (Wert-)Konstruktor, d.h. eine Konstante.

6.7. Schlussregeln für die Typinferenz

Die Typinferenz wurde bis jetzt nur informell eingeführt.

Im Folgenden soll sie formal und als Algorithmus für eine Vereinfachung von SML definieren. Dies erlaubt dann Korrektheitsbeweise und eine eindeutige Implementierung.

- Eine Vereinfachung von SML: SMalL
- Logischer Kalkül
- Gestalt der Schlussregeln eines logischen Kalküls
- Beweisbegriff in logischen Kalkülen
- Schlussregeln für die Typinferenz oder „Typisierungsregeln“
- Typisierungsbeweise
- Beispiele für Typisierungsbeweise

6.7.1. Eine Vereinfachung von SML: SMalL

Die Programmiersprache SMalL ist eine Vereinfachung von SML:

- SMalL enthält
  - die `val`–Deklarationen,
  - die vordefinierten Typen `bool`, `int`, `real`,
  - die Listen und Vektoren mit ihren vordefinierten Funktionen,
  - das `fn`–Konstrukt zur Funktionsbildung,
  - die Fallunterscheidung mit `if-then-else`
  - und die Typ-Constraints.
- SMalL lässt kein Pattern Matching, keine Verbunde, keine `let`– und keine `local`–Ausdrücke zu.
- Der Einfachheit halber und o.B.d.A. wird angenommen, dass alle Funktionen in SMalL präfix notiert werden.

Da eine n-stellige SMalL-Funktion als einstellige Funktion auf n-Vektoren angesehen werden kann, ist jede Funktionsanwendung `F(P1,...,Pn)` als `F P` darstellbar.

6.7.2. Logischer Kalkül

Zur Spezifikation eines Algorithmus für die Typinferenz in SMalL bedienen wir uns des Ansatzes eines „logischen Kalküls“. Ein logischer Kalkül besteht aus:

1. einer (*formalen*) *Sprache*, in der Aussagen über die zu beweisenden Eigenschaften ausgedrückt werden;
2. *Schlussregeln*, womit weitere Aussagen aus bereits festgestellten Aussagen geschlossen werden können;
3. *Axiome* und *Annahmen*, d.h. Aussagen, deren Gültigkeit nicht bewiesen werden muss, weil sie immer gelten (Axiome) oder weil sie angenommen werden (Annahme).

Logischer Kalkül (II)

Im Fall des (logischen) Kalküls für die Typinferenz gilt:

- die (formale) Sprache ist der Formalismus, in dem die Typisierungsaussagen (Typ-Constraints) wie etwa `(fn x => x) : 'a -> 'a` ausgedrückt werden;
- die Axiome sind die Typ-Constraints für die vordefinierten Typen wie etwa

  `34 : int`, `~98.67e4 : real`, `false : bool`, `nil : 'a list`, und
  `@ : ('a list * 'a list) -> 'a list`

  [`@` ist die vordefinierte Infixfunktion `append` von SML]
- die Annahmen sind die Typ-Constraints aus dem Programm

  — wie etwa: `x : int` oder `(fn x => a*x + b*x**2) : real -> real` —

  sowie für jeden (Teil-) Ausdruck `A` aus dem Programm ohne vorgegebenen Typ-Constraint ein Typ-Constraint `A : V` mit `V` Typvariable, so dass für jedes Paar mit verschiedenen Ausdrücken `A` die Typvariablen `V` paarweise verschieden sind.

Die Axiome werden *Typisierungsaxiome*, Annahmen *Typisierungsannahmen* genannt.

6.7.3. Gestalt der Schlussregeln eines logischen Kalküls

Eine Schlussregel hat die folgende Gestalt (mit $n \geq 1$):

$$\frac{Pr_1 \ \ldots \ Pr_n}{Sch}$$

- $Pr_1, \ldots, Pr_n$ und Sch Ausdrücke in der zugrunde liegenden formalen Sprache (des logischen Kalküls) sind.
- $Pr_1, \ldots, Pr_n$ sind die *Prämissen*, Sch der *Schluss*.
- Eine solche Regel bedeutet: Aus Pr_1 und ... und Pr_n folgt logisch Sch.

Verlangt eine Schlussregel eine Annahme A, so hat sie die Gestalt:
(Annahmen (A) können über jeder Prämisse stehen)

$$\frac{\begin{array}{ccc} (A) & & \\ Pr_1 & \ldots & Pr_n \end{array}}{Sch}$$

Gestalt der Schlussregeln eines logischen Kalküls (II)

Mögliche Schlussregeln wären z.B.:

$$\mathtt{R1}: \frac{\mathtt{append:('a\ list * 'a\ list) \text{->} 'a\ list} \quad \mathtt{[1,2]:int\ list} \quad \mathtt{[3,4]:int\ list}}{\mathtt{append([1,2],\ [3,4]):int\ list}}$$

$$\mathtt{R2}: \frac{\mathtt{1\ :\ int} \qquad \mathtt{2\ :\ int}}{\mathtt{[1,2]\ :\ int\ list}}$$

$$\mathtt{R3}: \frac{\mathtt{0\ :\ int} \qquad \mathtt{[1,2]\ :\ int\ list}}{\mathtt{[0,1,2]\ :\ int\ list}}$$

Diese Schlussregeln sind sinnvoll und korrekt gebildet.

Für einen logischen Kalkül für die Typinferenz werden aber Schlussregeln bevorzugt, die weder für eine besondere Prozedur noch für bestimmte konkrete aktuelle Parameter definiert sind.

Gestalt der Schlussregeln eines logischen Kalküls (III)

Eine bekannte Schlussregel ist der „*modus ponens*“ für Formeln der Aussagenlogik:

$$\frac{a \qquad a \Rightarrow b}{b}$$

Eine andere bekannte Schlussregel ist die „*Kontraposition*“, ebenfalls für Formeln der Aussagenlogik:

$$\frac{a \Rightarrow b}{(not\ b) \Rightarrow (not\ a)}$$

In den Schlussregeln *modus ponens* und *Kontraposition* stehen a und b für aussagenlogische Formeln.

a und b sind also Variablen der Metasprache, die zur Definition der Schlussregeln verwendet werden, aber keine aussagenlogischen Variablen.

6.7.4. Beweisbegriff in logischen Kalkülen

Ist ein logischer Kalkül durch eine formale Sprache, Schlussregeln und Axiome sowie möglichen Annahmen definiert, so werden Beweise als *Bäume* definiert, deren Blätter Axiome und Annahmen sind und wie folgt aufgebaut werden können:

1. Axiome und Annahmen sind Beweise.
2. Sind $B_1, \ldots, B_n$ Beweise mit Wurzel $P_1, \ldots$ bzw. P_n und ist

$$\frac{P_1 \quad \ldots \quad P_n}{S}$$

   eine Schlussregel, so ist der Baum

$$\frac{B_1 \quad \ldots \quad B_n}{S}$$

   ein Beweis (mit S als Wurzel, $B_1, \ldots, B_n$ als Unterbäume).

Beweisbegriff in logischen Kalkülen (II)

Oft werden zusätzliche Bedingungen gestellt, die so gebildete Bäume erfüllen müssen, um Beweise des logischen Kalküls zu sein.

Die Bäume, die aus den Axiomen und Annahmen unter Verwendung der Schlussregeln gebildet werden, haben ein paar bemerkenswerte Merkmale, die charakteristisch für die Beweise eines logischen Kalküls sind:

1. Ihre Kanten werden durch waagerechte Linien repräsentiert;
2. Ihre Wurzel befindet sich unten, ihre Blätter befinden sich oben.

Anmerkung: Letzteres ist deshalb bemerkenswert, weil in der Informatik Bäume üblicherweise „verkehrt herum“, also mit der Wurzel nach oben und den Blättern nach unten, dargestellt werden.

Beweisbegriff in logischen Kalkülen (III)

Beispiel:

Mit den zuvor angegebenen Schlussregeln R2 und R3 und den Axiomen `0:int`, `1:int` und `2:int` lässt sich der folgende Beweis bilden:

$$\mathtt{R3}: \dfrac{\mathtt{0:int} \qquad \mathtt{R2}: \dfrac{\mathtt{1:int} \quad \mathtt{2:int}}{\mathtt{[1,2]:int\ list}}}{\mathtt{[0,1,2]:int\ list}}$$

Verlangt eine Schlussregel eine Annahme A, so kann sie nur dann auf Beweise B_1, . . . , B_n zur Bildung eines Beweises B angewandt werden, wenn einer der Beweise B_1, . . . , B_n diese Annahme A als Blatt besitzt.

6.7.5. Die Schlussregeln für die Typinferenz oder „Typisierungsregeln“

Im Folgenden stehen $\mathtt{T}$, $\mathtt{T_i}$ für Typausdrücke, $\mathtt{V}$ für eine Typvariable, $\mathtt{F}$, $\mathtt{B}$, $\mathtt{P}$, $\mathtt{A}$, $\mathtt{A_i}$ für Ausdrücke:

T1 (Funktionsanwendung): $$\frac{\mathtt{F} : \mathtt{T}_1\text{->}\mathtt{T}_2 \qquad \mathtt{P} : \mathtt{T}_1}{\mathtt{F\ P} : \mathtt{T}_2}$$

T2 (if-then-else): $$\frac{\mathtt{B} : \mathtt{bool} \quad \mathtt{A}_1 : \mathtt{T} \quad \mathtt{A}_2 : \mathtt{T}}{(\mathtt{if\ B\ then\ A}_1\ \mathtt{else\ A}_2) : \mathtt{T}}$$

T3 (Funktionsbildung oder Lambda-Abstraktion): $$\frac{\begin{array}{c}(\mathtt{P} : \mathtt{T}_1)\\ \mathtt{A} : \mathtt{T}_2\end{array}}{(\mathtt{fn\ P\ =>\ A}) : \mathtt{T}_1\ \text{->}\ \mathtt{T}_2}$$

T4 (Instanziierung): $$\frac{\mathtt{A} : \mathtt{V}}{\mathtt{A} : \mathtt{T}}$$ mit $\mathtt{V}$ Typvariable und $\mathtt{T}$ Typausdruck, wobei $\mathtt{V}$ nicht in $\mathtt{T}$ vorkommt

Die Schlussregeln für die Typinferenz (II)

Die Bedingung für T4 ist notwendig, um z.B. `'a` $=$ `'a list` auszuschließen.

Eine Bindung einer Typvariablen `V` an einem Typausdruck `T` unter Verwendung der Schlussregel T4 gilt nicht nur für *ein*, sondern für *alle* Vorkommen von `V`, die gemäß der Blockstruktur dasselbe Objekt bezeichnen.

T5 (Einführung von Konstruktoren): $$\frac{\mathtt{A_1 : T} \qquad \mathtt{A_2 : T\ list}}{(\mathtt{A_1::A_2})\ \mathtt{:\ T\ list}}$$

T5-n-Vektor: $$\frac{\mathtt{A_1 : T} \quad \ldots \quad \mathtt{A_n : T}}{(\mathtt{A_1}, \ldots, \mathtt{A_n}) : \underbrace{\mathtt{T} * \ldots * \mathtt{T}}_{(\mathtt{n}\ \text{mal})}}$$

Für jedes $n \geq 1$ gibt es eine Regel T5-n-Vektor.
Da es eine maximale Stelligkeit n für n-Vektoren in einem Programm gibt, kann man sich auf eine endliche Anzahl von Schlussregeln beschränken.

6.7.6. Typisierungsbeweise

Typisierungsbeweise werden wie folgt definiert:

1. Typisierungsaxiome und -annahmen sind Typisierungsbeweise.
2. Ist
$$\frac{P_1 \quad \ldots \quad P_n}{S}$$
eine Typisierungsregel und sind $B_1, \ldots B_n$ Typisierungsbeweise von $P_1, \ldots$ bzw. P_n, so ist
$$C = \frac{B_1 \quad \ldots \quad B_n}{S}$$
ein Typisierungsbeweis, wenn in C für alle Ausdrücke A, die gemäß der Blockstruktur dasselbe Objekt bezeichnen, alle Anwendungen der Instanziierungsregel T4 mit Prämisse $A : V$ denselben Schluss $A : T$ (also mit demselben Typausdruck T) haben.

6.7.7. Beispiele für Typisierungsbeweise

Beispiel der Festlegung des Typs einer Funktion:

Seien die SMalL-Funktion

```
val f = fn x => +(*(a, x), 2.0)        (*   a*x + 2.0   *)
```

und die folgende Typisierungsannahme gegeben: `a : real`.

Der Typ `real -> real` von `f` lässt sich durch den folgenden Typisierungsbeweis feststellen:

$$
\text{T3: }\dfrac{\text{T1: }\dfrac{\texttt{+:(real*real)->real} \qquad \text{T5-2: }\dfrac{\text{T1: }\dfrac{\texttt{*:(real*real)->real} \qquad \text{T5-2: }\dfrac{\overset{\textit{(Ann.)}}{\texttt{a:real}} \qquad \text{T4: }\dfrac{\texttt{x:'x}}{\texttt{x:real}}}{\texttt{(a,x): real * real}}}{\texttt{*(a,x): real}} \qquad \overset{\textit{(Axiom)}}{\texttt{2.0:real}}}{\texttt{(*(a, x),2.0): real * real}}}{\texttt{+(*(a, x),2.0) : real}}}{\texttt{(fn x => +(*(a, x),2.0)) : real -> real}}
$$

Beispiele für Typisierungsbeweise (II)

Beispiel einer Überprüfung eines angegebenen Typs:

Sei nun die Funktion `f` wie folgt definiert:

```
val f = fn (x:int) => +(*(a, x), 2.0)        (*  a*x + 2.0  *)
```

- Der zuvor geführte Beweis lässt sich nicht mehr aufbauen, weil sich mit der Typisierungsannahme `x : int` kein Beweis von `*(a, x) : real` bilden lässt.

Beispiele für Typisierungsbeweise (III)

- Die Struktur des Ausdrucks `(fn x => +(*(a, x), 2.0))` wird im Beweis des ersten Beispiels genau widergespiegelt:

  Wird der Beweis von seiner Wurzel zu seinen Blätter, d.h. Annahmen oder Axiome durchlaufen, so zerlegt der Beweis den Ausdruck in seinen Teilausdrücke.

(*) Der Beweis könnte daher wie folgt aufgebaut werden:

`(fn x => +(*(a, x), 2.0)) : 'p1 -> 'p2`	da es ein `fn`-Ausdruck ist
`+(*(a, x), 2.0) : 'p2`	
`+ : ('p2 * 'p2) -> 'p2`	Axiom
`2.0 : real`	Axiom
daraus folgt: `'p2` = `real` und `*(a, x): real`	
`* : (real * real) -> real`	Axiom (bereits instanziiert)
daraus folgt: `x : real` und `'p1 : real`	
und damit: `(fn x => +(*(a, x), 2.0)) : real -> real`	

Beispiele für Typisierungsbeweise (IV)

Das Prinzip, das sich hier anhand eines Beispiels erkennen lässt, gilt allgemein. Darauf beruht die Automatisierung der Typprüfung.

Der Beweis des ersten Beispiels beinhaltet einen „Eureka-Schritt“, der schwer zu automatisieren ist:

- die Anwendung der Instanziierungsschlussregel T4 auf `x : 'x`, um `x : real` zu erhalten;

  Da es unendlich viele Typen gibt, die Kandidaten zur Bindung von `'x` sind, lässt sich daraus nur schwer ein brauchbarer Algorithmus entwickeln.

- Die Lösung dieses Problems liegt darin, Typvariablen nur soweit wie nötig an Typausdrücke zu binden — wie bereits im Beweis (*) angewendet.

- Für die Bindung von Typvariablen „nur soweit wie nötig“ wird der sogenannte Unifikationsalgorithmus verwendet.

6.8. Der Unifikationsalgorithmus

Im Folgenden bezeichnen V, V_i Typvariablen, T, T_i Typausdrücke, K, K_i Typkonstanten.

Zur Unifikation zweier Typausdrücke TA_1 und TA_2 gehe wie folgt vor:

Initialisierung: $M := \{(TA_1, TA_2)\}$ und $U := \{\}$

- M stellt eine Menge von Paaren dar, die Gleichungen repräsentieren.

  Anfangs enthält M nur ein Paar, das die beiden Typausdrücke enthält, deren Unifizierbarkeit überprüft werden soll.

- U ist eine Menge von Paaren, die Gleichungen zur Unifikation von TA_1 und TA_2 darstellen.

  Anfangs ist U leer.

  Terminiert der Unifikationsalgorithmus mit dem Ergebnis, dass die beiden Typausdrücke des Aufrufes unifizierbar sind, so enthält U die Gleichungen (repräsentiert als Paare), die diese Ausdrücke gleich machen, d.h. unifizieren.

Der Unifikationsalgorithmus (II)

Die Mengen M und U werden so lange wie möglich und so lange keine erfolglose Terminierung gemeldet wird wie folgt verändert:

Wähle (willkürlich) ein Paar (T_1, T_2) aus M und verändere M und U, je nach dem, welche Gestalt T_1 und T_2 haben:

1. Falls $(T_1, T_2) \in M$ und T_1 eine Typvariable ist, dann streiche (T_1, T_2) aus M, ersetze in M jedes Vorkommen von T_1 durch T_2 und füge (T_1, T_2) in U ein.

2. Falls $(T_1, T_2) \in M$ und T_1 eine Typkonstante ist
   (a) Wenn T_2 dieselbe Typkonstante ist, dann streiche (T_1, T_2) aus M.
   (b) Andernfalls wenn T_2 keine Typvariable ist, dann terminiere erfolglos.

Der Unifikationsalgorithmus (III)

3. Falls $(T_1, T_2) \in M$ und T_1 ein zusammengesetzter Typausdruck ist, der aus einem (Präfix-, Infix- oder Postfix-)Typoperator `Op` und den Typausdrücken $T_{11}, \ldots, T_{1n}$ (in dieser Reihenfolge) besteht

   (a) Wenn T_2 ebenfalls aus demselben (Präfix-, Infix- oder Postfix-)Typoperator `Op` und den Typausdrücken $T_{21}, \ldots, T_{2n}$ (in dieser Reihenfolge) besteht, dann ersetze (T_1, T_2) in M durch die n Paare $(T_{11}, T_{21}), \ldots, (T_{1n}, T_{2n})$.

   (b) Andernfalls wenn T_2 keine Typvariable ist, dann terminiere erfolglos.

4. Falls $(T_1, T_2) \in M$, T_1 keine Typvariable ist und T_2 eine Typvariable ist, dann ersetze (T_1, T_2) in M durch (T_2, T_1).

Ist kein Fall mehr anwendbar, dann liefere U als Unifikator von T_1 und T_2 und terminiere.

Anwendung des Unifikationsalgorithmus

Beispiel:

[Im Folgenden wird immer das erste Paar von M ausgewählt.]

$M =$	$U =$	
{(`'a -> 'a list`, `int list -> 'b`)},	{}	Fall 3
{(`'a`, `int list`), (`'a list`, `'b`)},	{}	Fall 1
{(`int list list`, `'b`)},	{(`'a`, `int list`)}	Fall 4
{(`'b`, `int list list`)},	{(`'a`, `int list`)}	Fall 1
{},	{(`'a`, `int list`), (`'b`, `int list list`)}	

U wird als Unifikator von `'a -> 'a list` und `int list -> 'b` geliefert.

6.9. Ein Verfahren zur automatischen Typinferenz

- Prinzip des Verfahrens
- Behandlung der Überschattung
- Beispiele

6.9.1. Prinzip des Verfahrens

Das Verfahren besteht aus zwei Phasen, deren Ausführungen beliebig ineinander verzahnt werden dürfen.

Das Verfahren erhält als Parameter

- einen Ausdruck `A` (mit oder ohne Typ-Constraint), dessen Typisierung überprüft oder festgestellt werden soll, und

- eine Menge M von Typ-Constraints.

  Beim ersten Aufruf des Verfahrens ist M leer.

Prinzip des Verfahrens (II)

Phase 1:

Hat der Ausdruck `A` einen (vorgegebenen) Typ-Constraint `C`, so füge `A : C` in M ein.

1. Hat `A` die Gestalt `val N = W` (oder `val rec N = W`), so
   (a) gebe `N` (bzw. `W`) alle in M vorkommenden Typ-Constraints von `W` (bzw. von `N`).
   (b) Wende das Verfahren mit Ausdruck `W` und Menge M an.

2. Andernfalls gehe nach der Gestalt von `A` (gemäß den Typisierungsregeln T1–T5) wie folgt vor (mit $\mathtt{V}_1$, $\mathtt{V}_2$, `W` Typvariablen, die weder in `A` noch in M vorkommen):
   (a) Falls `A` von der Form `B C` ist, so füge in M die folgenden Typ-Constraints ein:
   $$\mathtt{A} : \mathtt{V}_2, \quad \mathtt{B} : \mathtt{V}_1 \;\mathtt{->}\; \mathtt{V}_2, \quad \mathtt{C} : \mathtt{V}_1$$
   Wende das Verfahren auf Ausdruck `B` und Menge M an.

   Wende das Verfahren auf Ausdruck `C` und die Menge an, die sich aus der Anwendung des Verfahrens auf `B` und `M` ergeben hat.

Prinzip des Verfahrens (III)

(b) Falls `A` von der Form `(if B then` $\mathtt{C}_1$ `else` $\mathtt{C}_2$`)` ist, so füge in M die folgenden Typ-Constraints ein:

$$\mathtt{A} : \mathtt{V}, \quad \mathtt{B} : \mathtt{bool}, \quad \mathtt{C}_1 : \mathtt{V}, \quad \mathtt{C}_2 : \mathtt{V}$$

Wende das Verfahren auf Ausdruck `B` und Menge M an.

Wende dann das Verfahren auf Ausdruck $\mathtt{C}_1$ und die Menge M_1 an, die sich aus der Anwendung des Verfahrens auf `B` und M ergeben hat.

Wende das Verfahren auf Ausdruck $\mathtt{C}_2$ und die Menge an, die sich aus der vorangegangenen Anwendung des Verfahrens auf $\mathtt{C}_1$ ergeben hat.

(c) Falls `A` von der Form `(fn P => B)` ist, so füge in M die folgenden Typ-Constraints ein:

$$\mathtt{A} : \mathtt{V}_1 \texttt{ -> } \mathtt{V}_2, \quad \mathtt{P} : \mathtt{V}_1, \quad \mathtt{B} : \mathtt{V}_2$$

Wende das Verfahren mit Ausdruck `B` und Menge M an.

Prinzip des Verfahrens (IV)

Phase 2:

Kommen in M zwei Typ-Constraints $\mathtt{A} : \mathtt{T}_1$ und $\mathtt{A} : \mathtt{T}_2$ für denselben Ausdruck `A` vor, so unifiziere $\mathtt{T}_1$ und $\mathtt{T}_2$.

Wenn die Unifikation von $\mathtt{T}_1$ und $\mathtt{T}_2$ erfolglos terminiert, dann melde, dass der Ausdruck `A` einen Typfehler enthält, und terminiere.

Andernfalls binde die Typvariablen, wie es sich aus der Unifikation von $\mathtt{T}_1$ und $\mathtt{T}_2$ ergibt.

6.9.2. Behandlung der Überschattung

Auch ohne Deklarationen von lokalen Variablen mit `let`– oder `local`–Ausdrücken (die wir in SMalL nicht erlauben) kann es zu Überschattungen kommen:

- Die formalen Parameter einer Funktionsdefinition (`fn . => .`) können Namen überschatten.
- Deklarationen (mit `val`) können Namen überschatten.

Zur Behandlung der Überschattung wird eine Namensliste oder „Typisierungsumgebung“ verwendet, die wie eine Umgebung verwaltet wird:

Behandlung der Überschattung (II)

in Phase 1:

- Ist ein Ausdruck auszuwerten, der eine `val`–Deklaration oder ein `fn`–Ausdruck ist, dann wird der im `val`–Ausdruck deklarierte Name oder die formalen Parameter des `fn`–Ausdrucks am Anfang der „Typisierungsumgebung“ eingefügt.
- Wird die Auswertung eines solchen Ausdrucks abgeschlossen, so werden alle Namen von der Typisierungsumgebung entfernt, die beim Eintritt in die Typisierungsumgebung eingefügt wurden.

in Phase 2:

- Die Typ-Constraints, die während der Phase 2 einem Ausdruck zugeordnet werden, werden dem zutreffenden Ausdruck der Typisierungsumgebung zugeordnet.

  Dazu wird die Typisierungsumgebung vom Anfang her durchsucht.

Da zur Typinferenz keine Auswertung stattfindet, werden auch keine Werte an Namen gebunden. Folglich reicht es aus, Namen statt Gleichungen der Gestalt „Namen = Wert“ in die Typisierungsumgebung aufzunehmen.

6.9.3. Beispiele

Beispiel 1: Sei der Ausdruck A:

```
val f = fn x => +(*(a, x), 2.0)        (* d.h. a*x + 2.0 *)
```

Zerlegung des Ausdrucks: Typisierungsumgebung (Anfang: unten)

Zerlegung des Ausdrucks	Typisierungsumgebung
`val f = fn x => +(*(a, x), 2.0)`	
	`f                              : 'f`
	`fn x => +(*(a, x), 2.0)        : 'f`
`fn x => +(*(a, x), 2.0)`	`f                              : 'f`
	`fn x => +(*(a, x), 2.0)        : 'f    'x->'v1`
	`x                              : 'x`
`+(*(a, x), 2.0)`	`f                              : 'f`
	`fn x => +(*(a, x), 2.0)        : 'f    'x->'v1`
	`x                              : 'x`
	`+(*(a, x), 2.0))               : 'v1`

Beispiele (II)

Unifikation: `'f = 'x->'v1`

`+`	`f`	`: 'x->'v1`
	`fn x => +(*(a, x), 2.0)`	`: 'x->'v1`
	`x`	`: 'x`
	`+(*(a, x), 2.0)`	`: 'v1`
	`+`	`: 'v1*'v1->'v1`
`*(a, x)`	`f`	`: 'x->'v1`
	`fn x => +(*(a, x), 2.0)`	`: 'x->'v1`
	`x`	`: 'x`
	`+`	`: 'v1*'v1->'v1`
	`*(a, x)`	`: 'v1`
`*`	`f`	`: 'x->'v1`
	`fn x => +(*(a, x), 2.0)`	`: 'x->'v1`
	`x`	`: 'x`
	`+`	`: 'v1*'v1->'v1`
	`*(a, x)`	`: 'v1`
	`*`	`: 'v2*'v2->'v2`

Beispiele (III)

Unifikation: `'v2 = 'v1`

a	f	: 'f
	fn x => +(*(a, x), 2.0)	: 'x->'f
	x	: 'x
	+	: 'v1*'v1->'v1
	*(a, x)	: 'v1
	*	: 'v1*'v1->'v1
	a	: 'v1
x	f	: 'x->'v1
	fn x => +(*(a, x), 2.0)	: 'x->'v1
	x	: 'x 'v1
	+	: 'v1*'v1->'v1
	*(a, x)	: 'v1
	*	: 'v1*'v1->'v1
	a	: 'v1

Beispiele (IV)

Unifikation: `'x = 'v1`

`2.0`	`f`	`: 'v1->'v1`
	`fn x => +(*(a, x), 2.0)`	`: 'v1->'v1`
	`x`	`: 'v1`
	`+`	`: 'v1*'v1->'v1`
	`*(a, x)`	`: 'v1`
	`*`	`: 'v1*'v1->'v1`
	`a`	`: 'v1`
	`2.0`	`: 'v1   real`

Unifikation: `'v1 = real`

Ergebnis:	`f`	`: real->real`
	`fn x => +(*(a, x), 2.0)`	`: real->real`
	`x`	`: real`
	`+`	`: real*real->real`
	`*(a, x)`	`: real`
	`*`	`: real*real->real`
	`a`	`: real`
	`2.0`	`: real`

Beispiele (V)

Beispiel 2: Sei der Ausdruck A:

```
val not = fn x => if x = true than false else true;
```

Dies führt zu folgenden Ablauf:

```
val not = fn x => if x = true than false else true

            not                                     : 'a
            fn x => if x = true than false else true : 'a
```

```
fn x => if x = true than false else true

            not                                     : 'a
            fn x => if x = true than false else true : 'a   'b->'c
            x                                       : 'b
            if x = true than false else true        : 'c
```

Unifikation: `'a = 'b->'c`

Beispiele (VI)

```
if x = true than false else true
            not                                       : 'b->'c
            fn x => if x = true than false else true  : 'b->'c
            x                                         : 'b
            if x = true than false else true          : 'c   'd
            x = true                                  : bool
            false                                     : 'd   bool
            true                                      : 'd   bool
```

Unifikation: `'c = 'd  und  'd = bool`

```
x = true    not                                       : 'b->bool
            fn x => if x = true than false else true  : 'b->bool
            x                                         : 'b   'e
            if x = true than false else true          : bool
            x = true                                  : bool
            false                                     : bool
            true                                      : bool
            =                                         : 'e*'e->bool
            true                                      : 'e   bool
```

Beispiele (VII)

Unifikation: `'b = 'e  und  'e = bool`

```
not                                                : bool->bool
fn x => if x = true than false else true : bool->bool
x                                                  : bool
if x = true than false else true           : bool
x = true                                           : bool
false                                              : bool
true                                               : bool
=                                                  : bool*bool->bool
true                                               : bool
```

7. Abstraktionsbildung mit Prozeduren höherer Ordnung

- 7.1 Prozeduren als Parameter und Wert von Prozeduren
- 7.2 Currying
- 7.3 Funktionskomposition
- 7.4 Grundlegende Funktionen höherer Ordnung
- 7.5 Beispiel: Ein Rekursionsschema zur Akkumulation

7.1. Prozeduren als Parameter und Wert von Prozeduren

- In SML und anderen funktionalen Programmiersprachen sind Funktionen Werte.

  $\leadsto$ Funktionen können wie andere Werte auch als aktuelle Parameter von Prozeduren, u.a. von Funktionen, auftreten

- Prozeduren (Funktionen), die als Parameter oder als Wert Funktionen haben, werden Prozeduren (Funktionen) höherer Ordnung genannt.

Die Bezeichnung „Prozeduren höherer Ordnung“ basiert auf folgender Hierarchie:

- Die Ordnung 0 umfasst die Konstanten.
- Die Ordnung 1 umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung 0 sind.
- Die Ordnung $n+1$ umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung n sind.

Funktionen höherer Ordnung werden manchmal *functionals* oder *Operatoren* genannt.

Prozeduren als Parameter und Wert von Prozeduren (II)

Funktionen höherer Ordnung stellen eine nützliche Abstraktion dar.

Beispiel: Ist $f : \mathbb{R} \rightarrow \mathbb{R}$ eine (ableitbare) mathematische Funktion, so kann die Ableitung f' von f wie folgt geschätzt werden (wobei $\Delta \in \mathbb{R}$ eine kleine Zahl ist):

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

Dies lässt sich wie folgt in SML implementieren:

```
- val delta = 1e~5;
val delta = 1E~5 : real

- fun abl(f) = let
                   fun f'(x) = (f(x+delta) - f(x)) / delta
               in
                   f'
               end;
val abl = fn : (real -> real) -> real -> real
```

Prozeduren als Parameter und Wert von Prozeduren (III)

- Die Funktion `abl` hat als Parameter eine Funktion `f`.
- Innerhalb von `abl` wird eine neue Funktion `f’` mit Hilfe von `f` und `delta` definiert.
- Das Ergebnis von `abl(f)` ist selbst eine Funktion ($\rightarrow$ `f’`):
  d.h., dass `abl(f)` wiederum auf reelle Zahl angewandt werden kann.

Der Typausdruck `(real -> real) -> real -> real` bezeichnet wegen der Rechtsassoziativität von „->“ den Typ `(real -> real) -> (real -> real)` :

- `abl` hat also als Parameter eine Funktion vom Typ `real -> real` und
- liefert als Ergebnis ebenfalls eine Funktion vom Typ `real -> real` liefert.

Prozeduren als Parameter und Wert von Prozeduren (IV)

Anwendungsbeispiele:

```
- abl( fn x => x*x )(5.0);
val it = 10.0000099994 : real

- abl( fn x => x*x*x )(5.0);
val it = 75.0001499966 : real
```

- Ableitung der Funktion $x \mapsto x^2$ ist die Funktion $x \mapsto 2x$ (und $2 * 5 = 10$).
- Ableitung der Funktion $x \mapsto x^3$ ist die Funktion $x \mapsto 3x^2$ (und $3 * 5^2 = 75$).

Die mit SML berechneten Zahlen sind Schätzungen:

- Mit anderen Werten für `delta` kann man die Schätzung verbessern.
- Da die Schätzung somit auch von `delta` abhängt, bietet es sich an, `delta` zu einem zweiten Parameter von `abl` zu machen.

Prozeduren als Parameter und Wert von Prozeduren (V)

```
- fun abl(f, delta:real) =
                  let
                      fun f'(x) = (f(x+delta) - f(x)) / delta
                  in
                      f'
                  end;
val abl = fn : (real -> real) * real -> real -> real

- abl( fn x => x*x, 1e~10 )(5.0);
val it = 10.0000008274 : real

- abl( fn x => x*x*x, 1e~10 )(5.0);
val it = 75.0000594962 : real
```

Der Name der neuen Funktion `f'` ist nur lokal in der Definition von `abl` deklariert. Dies kann aber genauso gut durch eine anonyme Funktionsdeklaration ersetzt werden:

```
- fun abl(f, delta:real) = fn x => (f(x+delta) - f(x)) / delta;
val abl = fn : (real -> real) * real -> real -> real
```

Prozeduren als Parameter und Wert von Prozeduren (VI)

Unabhängig von der Form der Deklaration kann man mit `abl` erzeugte Funktionen selbst als Parameter von `abl` verwenden:

```
- abl( abl(fn x => x*x, 1e~5), 1e~5 )(5.0);
val it = 2.00003569262 : real

- abl( abl(fn x => x*x*x, 1e~5), 1e~5 )(5.0);
val it = 30.0002511722 : real
```

- Die Ableitung der Ableitung von $x \mapsto x^2$ ist die Funktion $x \mapsto 2$.
- Die Ableitung der Ableitung von $x \mapsto x^3$ ist die Funktion $x \mapsto 6x$.

Prozeduren als Parameter und Wert von Prozeduren (VII)

Ein weiteres Beispiel einer Funktionen höherer Ordnung ist die Identitätsfunktion:

```
- val id = fn x => x;
val id = fn : 'a -> 'a

- id(2);
val it = 2 : int

- id(id)(2);
val it = 2 : int
```

- Im Teilausdruck `id(id)` hat die Funktion namens `id` sogar sich selbst als Parameter, und sie liefert auch sich selbst als Ergebnis.
- `id` ist eine Funktion höherer Ordnung (da Funktionen als Parameter erlaubt sind).
- `id` ist polymorph, da `id` auf Objekte unterschiedlicher Typen anwendbar ist.
- `id` ist eine Funktion der Ordnung n — für jedes $n > 0$, da `id` auch auf Funktionen beliebiger Ordnung angewendet werden darf.

7.2. Currying

- Prinzip
- andere Syntax zur Deklaration von curried Funktionen
- einfache Deklaration von curried Funktionen
- die Funktion `curry`
- Umkehrung von `curry`
- Nicht-curried und curried Funktionen im Vergleich

7.2.1. Prinzip

Gegeben sei die folgende mathematische Funktion:

$$\begin{aligned} f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z} \\ (n_1, n_2, n_3) &\mapsto n_1 + n_2 + n_3 \end{aligned}$$

f kann wie folgt in SML implementiert werden:

```
- fun f(n1, n2, n3) : int = n1 + n2 + n3;
val f = fn : int * int * int -> int
```

Im Folgenden bezeichnen wir mit $F(A, B)$ die Menge der Funktionen von A in B ($A \rightarrow B$).

Currying — Prinzip (II)

Aus der Funktion f lässt sich die folgende Funktion definieren:

$$\begin{array}{rccl} f1: & \mathbb{Z} & \to & F(\mathbb{Z} \times \mathbb{Z}, \mathbb{Z}) \\ & n_1 & \mapsto & f1(n_1) \end{array}$$

mit:

$$\begin{array}{rccl} f1(n1): & \mathbb{Z} \times \mathbb{Z} & \to & \mathbb{Z} \\ & (n_2, n_3) & \mapsto & f(n_1, n_2, n_3) = n_1 + n_2 + n_3 \end{array}$$

Die Funktion $f1$ kann wie folgt in SML implementiert werden:

```
- fun f1(n1)(n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- f1(1)(1, 1);
val it = 3 : int

- f1(1);
val it = fn : int * int -> int
```

Currying — Prinzip (III)

`int -> int * int -> int` entspricht `int -> ((int * int) -> int)` wegen der Rechtsassoziativität von „`->`“ und der Präzedenzen zwischen „`*`“ und „`->`“.

- Die Funktion `f1` bildet also eine ganze Zahl auf eine Funktion vom Typ `((int * int) -> int)` ab.

In ähnlicher Weise lässt sich die folgende Funktion definieren:

$$
\begin{array}{rccl}
f11(n_1): & \mathbb{Z} & \rightarrow & F(\mathbb{Z},\mathbb{Z}) \\
 & n_2 & \mapsto & f11(n_1)(n_2)
\end{array}
$$

mit:

$$
\begin{array}{rccl}
f11(n_1)(n_2): & \mathbb{Z} & \rightarrow & \mathbb{Z} \\
 & n_3 & \mapsto & f1(n_1)(n_2,n_3) = f(n_1,n_2,n_3) = n_1+n_2+n_3
\end{array}
$$

Die Funktion $f11$ lässt sich wie folgt in SML implementieren:

```
- fun f11(n1)(n2)(n3) = f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

Currying — Prinzip (IV)

`int -> int -> int -> int` entspricht `int -> (int -> (int -> int))`, da der Typkonstruktor „->“ rechtsassoziativ ist.

```
- f11(1)(1)(1);
val it = 3 : int

- f11(1)(1);
val it = fn : int -> int

- f11(1);
 val it = fn : int -> int -> int
```

So lässt sich jede n-stellige Funktion durch einstellige (unäre) Funktionen darstellen.

Dies ermöglicht oft eine wesentlich kompaktere und übersichtlichere Schreibweisen von Funktionsdefinitionen.

7.2.2. Andere Syntax zur Deklaration von „curried“ Funktionen

In SML ist ein einelementiger Vektor mit seinem Element identisch.
Daher lassen sich `f1` und `f11` auch wie folgt deklarieren:

```
- fun f1 n1 (n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- f1(1)(1, 1);
val it = 3 : int

- f1 1 (1, 1);
val it = 3 : int

- f1(1);
val it = fn : int * int -> int

- f1 1;
val it = fn : int * int -> int
```

Andere Syntax zur Deklaration von „curried“ Funktionen (II)

```
- fun f11 n1 n2 n3 = f1 n1 (n2, n3);
val f11 = fn : int -> int -> int -> int

- f11(1)(1)(1);
val it = 3 : int

-  f11 1 1 1;
val it = 3 : int

- f11(1)(1);
val it = fn : int -> int

- f11 1 1;
val it = fn : int -> int

- f11(1);
val it = fn : int -> int -> int

- f11 1;
val it = fn : int -> int -> int
```

Andere Syntax zur Deklaration von „curried“ Funktionen (III)

Unter Verwendung des `fn`-Konstrukts werden `f1` und `f11` wie folgt deklariert:

```
- val f1 = fn n1 => fn (n2, n3) => f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- val f11 = fn n1 => fn n2 => fn n3 => f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

- Das Prinzip, das der Bildung von `f1` aus `f` (und `f11` aus `f1`) zugrunde liegt, wird *Currying* genannt (nach dem Logiker Haskell B. Curry).
- Die n–stellige Funktion, die sich durch „Currying“ aus einer $(n+1)$–stelligen Funktion f ergibt, wird „curried“ Form von f genannt.

7.2.3. Einfache Deklaration von curried Funktionen

Wie die Beispiele

```
- fun f11 n1 n2 n3 = ... ;
```

oder

```
- val f11 = fn n1 => fn n2 => fn n3 => ... ;
```

zeigen, lassen sich curried Funktionen einfach dadurch deklarieren, dass ihre Parameter nicht als Vektor angegeben werden:

```
- fun f' n1 n2 n3 : int = n1 + n2 + n3;
val f' = fn : int -> int -> int -> int
```

Nun ist `f' n1` die Funktion, die `f1(n1)` entspricht, `f' n1 n2` die Funktion, die `f11(n1)(n2)` entspricht.

Einfache Deklaration von curried Funktionen (II)

Curried Funktionen dürfen auch rekursiv sein; z.B.:

```
- fun ggT a b = if a < b then ggT b a
                           else if b = 0 then a
                                          else ggT b (a mod b);
val ggT = fn : int -> int -> int

- ggT 150 60;
val it = 30 : int

- ggT 150;
val it = fn : int -> int
```

Bemerkung:
Die Schreibweise `ggT b a mod b` statt `ggT b (a mod b)` wäre inkorrekt: Wegen der Präzedenzen bezeichnet sie `((ggT b) a) mod b`.

7.2.4. Die Funktion höherer Ordnung `curry` zur Berechnung der curried Form einer binären Funktion

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

oder

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

Man beachte, dass `curry` eine polymorphe Funktion ist.

```
- curry(fn (a,b) => 2*a + b);
val it = fn : int -> int -> int

- curry(fn (a,b) => 2*a + b) 3;
val it = fn : int -> int

- curry(fn (a,b) => 2*a + b) 3 1;
val it = 7 : int
```

7.2.5. Umkehrung der Funktion `curry`

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

oder

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

`uncurry` ist eine polymorphe Funktion:

```
- fun f x y = 2 + x + y;
val f = fn : int -> int -> int

- fun g x y = 2.0 + x + y;
val g = fn : real -> real -> real

- uncurry f;
val it = fn : int * int -> int

- uncurry f (1,1);
val it = 4 : int
```

Umkehrung der Funktion `curry` (II)

`uncurry` ist die Umkehrung von `curry`:

```
- curry(uncurry(f));
val it = fn : int -> int -> int

- curry(uncurry(f)) 1 1;
val it = 4 : int

- uncurry(curry(uncurry(f)))(1,1);
val it = 4 : int
```

7.2.6. Nicht-curried und curried Funktionen im Vergleich

Darstellung der Unterschiede am Beispiel einer einfachen Funktion:
(Annahme: die vordefinierte Funktion „*“ hat den Typ `int * int -> int`)

	nicht-curried	**curried**
mögliche Deklaration	`fun mal(x, y) = x * y` `val mal = (fn (x, y) => x * y)`	`fun mal x y = x * y` `val mal = (fn x y => x * y)` `fun mal x = (fn y => x * y)` `val mal = (fn x => (fn y => x * y))`
Typ	`int * int -> int`	`int -> int -> int`
Aufruf	`mal(2,3)` (hat Wert 6)	`mal 2 3` (hat Wert 6)
zu wenige Aufrufparameter	—	`mal 2` (hat eine Funktion als Wert)
	`val doppelt = fn y => mal(2, y)`	`val doppelt = mal 2`

7.3. Funktionskomposition

- Funktionskomposition

- die Kombinatoren `I`, `S` und `K`

7.3.1. Funktionskomposition

Die SML-Standardbibliothek enthält eine Funktion höherer Ordnung, die wie folgt definiert werden kann:

```
- infix o;
infix o

- fun (f o g)(x) = f(g(x));
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- ((fn x => 2*x) o (fn x => x+1))(3);
val it = 8 : int

- Math.sqrt o ~;
val it = fn : real -> real

- val f = Math.sqrt o ~;
val f = fn : real -> real

- f(~4.0);
val it = 2.0 : real
```

Funktionskomposition (II)

Die Infixfunktion $\mathtt{o}$ leistet eine Funktionskomposition in sogenannter „Anwendungsreihenfolge“ (oder Funktionalreihenfolge), d.h.:

$$(\mathtt{f} \circ \mathtt{g})(\mathtt{x}) = \mathtt{f}(\mathtt{g}(\mathtt{x}))$$

In der Mathematik ist auch ein Funktionskomposition in sogenannter „Diagrammreihenfolge“ üblich, die wie folgt definiert ist:

$$(\mathtt{f} \circ_{\mathtt{d}} \mathtt{g})(\mathtt{x}) = \mathtt{g}(\mathtt{f}(\mathtt{x}))$$

Um Verwechslung mit dem SML-Operator $\mathtt{o}$ zu vermeiden, verwenden wir hier die Notation $\circ_{\mathtt{d}}$ für den Funktionskompositionsoperator in Diagrammreihenfolge, der in der Mathematik üblicherweise $\circ$ notiert wird.

Funktionskomposition (III)

Die Bezeichnung „Diagrammreihenfolge“ bezieht sich auf die folgende Darstellung der Komposition zweier Funktionen $f : A \to B$ und $g : B \to C$ sind:

$$A \xrightarrow{f} B \xrightarrow{g} C$$

$(f \circ_{\mathtt{d}} g)$ ist also eine Funktion von A in C.

Die Reihenfolge f vor g in der Notation $(f \circ_{\mathtt{d}} g)$ folgt der Reihenfolge im Diagramm, aber nicht der Reihenfolge der Funktionsanwendungen:

$$(f \circ_{\mathtt{d}} g)(x) = g(f(x))$$

Im Gegensatz dazu entspricht die Anwendungsreihenfolge der Reihenfolge der Funktionsanwendungen:

$$(f \circ g)(x) = f(g(x))$$

[Der Funktionskompositionsoperator o ist in SML vordefiniert.]

7.3.2. Die Kombinatoren `I`, `K` und `S`

Die polymorphe Identitätsfunktion „`id`“ wird auch als „`I`“ notiert und „Identitätskombinator“ genannt:

```
- fun I x = x;
val I = fn : 'a -> 'a

- I 5;
val it = 5 : int

- I [1,2];
val it = [1,2] : int list

- I (fn x => 2 * x);
val it = fn : int -> int
```

Die Kombinatoren I, K und S (II)

Der „Kompositionskombinator“ S verallgemeinert die Funktionskomposition mit o:

```
- fun S x y z = x z (y z);
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Der „Konstantenkombinator“ K entspricht der Projektion auf die erste Komponente:

```
- fun K x y = x;
val K = fn : 'a -> 'b -> 'a
```

Mittels S und K lässt sich der Identitätskombinator I definieren:

```
- S K K 5;
val it = 5 : int

- S K K [1,2];
val it = [1,2] : int list
```

Die Kombinatoren `I`, `K` und `S` (III)

Bedeutung dieser Kombinatoren:

- Alleine mittels `S` und `K` können alle Funktionen des sogenannten Lambda-Kalküls ausgedrückt werden (der Lambda-Kalkül ist eine/die formale Grundlage der rein funktionalen Programmierung).

  In anderen Worten: Man kann jeden Algorithmus nur unter Verwendung von `S` und `K` ausdrücken.

- Diese Kombinatoren spielen eine Rolle in der theoretischen Informatik.

  Wenn alle Konzepte einer funktionalen Programmiersprache aus `S` und `K` abgeleitet werden können, reicht es Eigenschaften dieser Sprache nur für `S` und `K` zu beweisen.

7.4. Grundlegende Funktionen höherer Ordnung

- `map`
- `filter`
- `foldl` und `foldr`
- `exists` und `all`
- `repeat`

7.4.1. `map`

Die Funktion höherer Ordnung `map` wendet eine unäre Funktion auf alle Elementen einer Liste an und liefert als Wert die Liste der (Werte dieser) Funktionsanwendungen:

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

- map quadrat [2, 3, 5];
val it = [4, 9, 25]

- map Math.sqrt [4.0, 9.0, 25.0];
val it = [2.0, 3.0, 5.0] : real list
```

Die Funktion `map` kann wie folgt in SML definiert werden:

```
- fun map f nil        = nil
    | map f (h :: t) = f(h) :: map f t;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

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

`map` (II)

Die curried Form von `map` hat den Vorteil, dass `map` einfach auf Listen von Listen anzuwenden ist.

Beipsiel:

```
map (map quadrat) [[1,2], [3,4], [5,6]]
```

Die Funktion `(map quadrat)` wird hier auf jedes (Listen-)Element der Liste `[[1,2], [3,4], [5,6]]` angewandt.

```
- map quadrat;
val it = fn : int list -> int list

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

7.4.2. Vorteil der curried Form am Beispiel der Funktion `map`

Wäre `map` nicht in curried Form definiert, müsste man statt `(map quadrat)` einen komplizierteren Ausdruck mit einer anonymen Funktion verwenden.

Sei `map'` eine Version von `map`, die nicht in curried Form ist:

```
- fun map'(f, nil)    = nil
    | map'(f, h :: t) = f(h) :: map'(f, t);
val map' = fn : ('a -> 'b) * 'a list -> 'b list

- map'(quadrat, [2, 3, 5]);
val it = [4,9,25] : int list
```

Vorteil der curried Form am Beispiel der Funktion `map` (II)

Unter Verwendung von `map’` müsste der Ausdruck

```
map (map quadrat) [[1,2], [3,4], [5,6]]
```

zu dem komplizierteren Ausdruck

```
map’(fn L => map’(quadrat, L), [[1,2], [3,4], [5,6]])
```

umformuliert werden:

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

Die Verwendung der anonymen Funktion `(fn L => map’(quadrat, L))` ist notwendig, weil im Gegensatz zu `map` die Funktion `map’` *zwei* Parameter verlangt.

7.4.3. filter

Diese Funktion höherer Ordnung filtert aus einer Liste alle Elemente heraus, die ein Prädikat erfüllen:

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool

- filter ist_gerade [1,2,3,4,5];
val it = [2,4] : int list

- fun filter pred nil      = nil
    | filter pred (h :: t) = if pred(h)
                             then h :: filter pred t
                             else filter pred t;
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

`filter` (II)

```
- filter (fn x => (x mod 2) = 0) [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- filter (not o (fn x => (x mod 2) = 0)) [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

Man beachte die Verwendung des (vordefinierten) Operators `o` für die Funktionskomposition.

`filter` (III)

Auch `filter` ist in curried Form definiert, so dass kompaktere und übersichtlichere Definitionen möglich sind:

```
- val gerade_elemente = filter ist_gerade;
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente = filter (not o ist_gerade);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

7.4.4. Vorteil der curried Form am Beispiel der Funktion `filter`

Sei `filter'` eine Version von `filter`, die nicht in curried Form ist:

```
- fun filter'(pred, nil)       = nil
    | filter'(pred, (h :: t)) = if pred(h)
                                then h :: filter'(pred, t)
                                else filter'(pred, t);
val filter' = fn : ('a -> bool) * 'a list -> 'a list
```

Die zuvor definierten Funktionen würden mit `filter'` wie folgt aussehen:

```
- val gerade_elemente = fn L => filter'(ist_gerade, L);
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente = fn L => filter'(not o ist_gerade, L);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

7.4.5. `foldl` und `foldr`

Die Funktionen höherer Ordnung `foldl` und `foldr` (Zusammenfalten von links bzw. rechts) wenden wie folgt eine Funktion auf die Elemente einer Liste an:

`foldl f z` $[x_1, x_2, \ldots, x_n]$ entspricht $f(x_n, \ldots, f(x_2, f(x_1, z))\ldots)$

`foldr f z` $[x_1, x_2, \ldots, x_n]$ entspricht $f(x_1, f(x_2, \ldots, f(x_n, z)\ldots))$

Beispiele (op erlaubt Infix-Operatoren als Präfix-Operatoren zu verwenden):

```
- foldl (op +) 0 [2,3,5];      (* entspricht 5 + (3 + (2 + 0)) *)
val it = 10 : int

- foldr (op +) 0 [2,3,5];      (* entspricht 2 + (3 + (5 + 0)) *)
val it = 10 : int

- foldl (op -) 0 [7,10];       (* entspricht 10 - (7 - 0) *)
val it = 3 : int

- foldr (op -) 0 [7,10];       (* entspricht 7 - (10 - 0) *)
val it = ~3 : int
```

`foldl` und `foldr` (II)

Die Funktionen können wie folgt in SML implementiert werden:

```
- fun foldl f z nil      =  z
    | foldl f z (x::L)   =  foldl f (f(x,z)) L;
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- fun foldr f z nil      =  z
    | foldr f z (x::L)   =  f(x, foldr f z L);
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Die Klammerung um `f(x,z)` ist bei `foldl` notwendig, weil `foldl` eine curried Funktion ist. Im Ausdruck `foldl f f (x,z) L` würde `foldl` auf `f` angewandt werden und eine Funktion liefern, die dann auf `f` angewandt werden würde und nicht wie beabsichtigt auf `f(x,z)`.

`foldl` **und** `foldr` **(III)**

- Viele binäre Funktionen haben neutrale Elemente, z.B. ist `0` das neutrale Element für `+`, `1` für `*`, `nil` für `@` (Listenkonkatenation) und `""` für `^` (Stringkonkatenation).
- Die Funktionen `foldl` und `foldr` werden typischerweise so verwendet, dass das zweite Argument `z` das neutrale Element des ersten Arguments `f` ist.
- Die wichtigste Anwendung von `foldl` und `foldr` ist die Definition von neuen Funktionen auf Listen mit Hilfe von binären Funktionen auf den Elementen der Listen.

`foldl` und `foldr` (IV)

```
- val listsum = fn L => foldl (op +) 0 L;
val listsum = fn : int list -> int

- val listprod = fn L => foldl (op * ) 1 L;
val listprod = fn : int list -> int

- val listconc = fn L => foldr (op ^) "" L;
val listconc = fn : string list -> string

- val listapp = fn L => foldr (op @) nil L;
val listapp = fn : ‘a list list -> ‘a list

- listsum [1,2];
val it = 3 : int

- listsum [1,2,3,4];
val it = 10 : int
```

`foldl` und `foldr` (V)

```
- listprod [1,2];
val it = 2 : int

- listprod [1,2,3,4];
val it = 24 : int

- listconc ["abc", "de", "fghi", "j"];
val it = "abcdefghij" : string

- listapp [[1,2], [10,20,30], [100]];
val it = [1,2,10,20,30,100] : int list
```

Syntaktischer Hinweis: da `*)` das Symbol für das Ende eines Kommentars ist, führt der Ausdruck `(op *)` zu der folgenden Fehlermeldung:

```
Error: unmatched close comment
```

$\leadsto$ Zwischen `*` und `)` im Ausdruck `(op * )` muss mindestens ein Leerzeichen stehen!

`foldl` und `foldr` (VI)

- Neue Funktionen wie `listsum`, `listprod` usw. könnten auch explizit rekursiv definiert werden:

  Die Definitionen wären sehr ähnlich zueinander und würden sich nur an wenigen Stellen unterscheiden.

- `foldl` und `foldr` sind Abstraktionen, die die Gemeinsamkeiten solcher Definitionen darstellen, damit man sie nicht bei jeder neuen Funktion wiederholen muss.

- `foldl` und `foldr` sind damit sehr mächtige und grundlegende Hilfsmittel.

- Da `foldl` und `foldr` curried Funktionen sind, kann man mit ihnen neue Funktionen definieren, ohne Namen für jedes Argument der neuen Funktionen erfinden zu müssen.

`foldl` **und** `foldr` **(VII)**

Die kompakteste (und übersichtlichste) Definition der obigen Funktionen lautet:

```
- val listsum = foldl (op +) 0;
val listsum = fn : int list -> int

- val listprod = foldl (op * ) 1;
val listprod = fn : int list -> int

- val listconc = foldr (op ^) "";
val listconc = fn : string list -> string
```

`foldl` **und** `foldr` **(VIII)**

Welche bekannten Listenfunktionen werden wie folgt mittels `foldl` und `foldr` definiert?

```
- val a = fn L => foldl (op ::) nil L;
val a = fn : 'a list -> 'a list

- a [1,2,3];
val it = [3,2,1] : int list

- val b = fn L => foldr (op ::) nil L;
val a = fn : 'a list -> 'a list

- b [1,2,3];
val it = [1,2,3] : int list
```

7.4.6. `exists` und `all`

Die Funktion höherer Ordnung `exists` überprüft, ob ein Prädikat für manche Elemente einer Liste erfüllt ist:

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool

- exists ist_gerade [1,2,3,4,5];
val it = true : bool

- exists ist_gerade [1,3,5];
val it = false : bool
```

Die Funktion `exists` kann wie folgt in SML implementiert werden:

```
- fun exists pred nil      =  false
    | exists pred (h::t)   =  (pred h) orelse exists pred t;
val exists = fn : ('a -> bool) -> 'a list -> bool
```

`exists` und `all` (II)

Die Funktion höherer Ordnung `all` überprüft, ob ein Prädikat für alle Elemente einer Liste erfüllt ist:

```
- all ist_gerade [1,2,3,4,5];
val it = false : bool

- all ist_gerade [2,4];
val it = true : bool
```

Die Funktion `all` kann wie folgt in SML implementiert werden:

```
- fun all pred nil      =  true
    | all pred (h::t)   =  (pred h) andalso all pred t;
val all = fn : ('a -> bool) -> 'a list -> bool
```

7.4.7. Wiederholte Funktionsanwendung

Angewandt auf eine Funktion `f` und eine natürliche Zahl `n` ist die Funktion höherer Ordnung `repeat` eine Funktion, die `n` mal die Funktion `f` anwendet:

```
- fun repeat f 0 x = x
    | repeat f n x = repeat f (n-1) (f x);
val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a

- repeat (fn x => x+1) 3 4;
val it = 7 : int

- repeat (fn x:int => x*x) 3 2;
val it = 256 : int
```

7.5. Beispiel: Ein Rekursionsschema zur Akkumulation

- `summe`
- `produkt`
- das gemeinsame Rekursionsschema
- Anwendung zur Integralschätzung

7.5.1. `summe`

Die folgende Funktion höherer Ordnung `summe` entspricht dem Summenzeichen, für das in der Mathematik die Sigma-Notation (Σ) üblich ist:

```
- fun summe(von, bis, schritt, funktion, akk) =
    if von > bis
    then akk
    else summe(von+schritt, bis, schritt, funktion,
                funktion(von)+akk );
val summe = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x), 0);
val it = 10 : int
```

7.5.2. `produkt`

Die folgende Funktion höherer Ordnung `produkt` entspricht dem Produktzeichen, für das in der Mathematik die Pi-Notation (Π) üblich ist:

```
- fun produkt(von, bis, schritt, funktion, akk) =
      if von > bis
      then akk
      else produkt(von+schritt, bis, schritt, funktion,
                   funktion(von)*akk );
val produkt = fn : int * int * int * (int -> int) * int -> int

- produkt(1,4,1, (fn x => x), 1);
val it = 24 : int
```

7.5.3. Das gemeinsame Rekursionsschema

Das gemeinsame Rekursionsschema der Definitionen von `summe` und `produkt` kann wie folgt unabhängig von der verwendeten arithmetischen Operation definiert werden:

```
- fun akkumulieren(operation)(von, bis, schritt, funktion, akk) =
        if von > bis
        then akk
        else akkumulieren(operation)(von+schritt, bis, schritt,
                                     funktion,
                                     operation(funktion(von),akk) );
val akkumulieren = fn
    ('a * 'b -> 'b) -> int * int * int * (int -> 'a) * 'b -> 'b
```

Das gemeinsame Rekursionsschema (II)

```
- val summe = akkumulieren (op +);
val summe = fn : int * int * int * (int -> int) * int -> int

- val produkt = akkumulieren (op * );
val produkt = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x), 0);
val it = 10 : int

- produkt(1,4,1,(fn x => x), 1);
val it = 24 : int
```

Welche Funktion wird wie folgt mittels `akkumulieren` definiert?

```
fun a n = akkumulieren (op * ) (1, n, 1, (fn x => x), 1);
```

7.5.4. Anwendung zur Integralschätzung

Funktion `real_akkumulieren` zum Abschätzen des Integrals einer Funktion $\mathbb{R} \to \mathbb{R}$

Das Integral einer Funktion f zwischen a und b (unter gewissen Voraussetzungen wie der Stetigkeit von f) kann wie folgt abgeschätzt werden:

$$\Delta * f(a + \Delta) + \Delta * f(a + 2\Delta) + \Delta * f(a + 3\Delta) + \ldots$$

```
- fun real_akkumulieren(operation)(von:real,     bis:real,
                                   schritt:real, funktion,
                                   akk:real)  =
  if von > bis
  then akk
  else real_akkumulieren(operation)(von+schritt, bis, schritt,
                                    funktion,
                                    operation(funktion(von),akk) );
val real_akkumulieren = fn
:('a*real->real) -> real * real * real * (real->'a) * real -> real
```

Anwendung zur Integralschätzung (II)

```
- fun integral(f, von, bis, delta) =
   real_akkumulieren(op +)( von+delta, bis, delta,
                            (fn x => delta*f(x)), 0.0);
val integral = fn : (real -> real) * real * real * real -> real
- integral((fn x => 1.0), 0.0, 3.0, 0.5);

val it = 3.0 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.5);
val it = 10.5 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.1);
val it = 8.7 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.0001);
val it = 8.9997 : real
```

Mit `delta` kann die Güte der Schätzung gesteuert werden ($\int_0^3 2x\,dx = 3^2 - 0^2 = 9$).

Anmerkung zum Beispiel im Abschnitt 7.5.4

Die Auswertungen der mit der Funktion `integral` gebildeten Ausdrücke liefern überraschende Werte:

- für `delta = 0.5` werden 6 Iterationen durchlaufen;
- für `delta = 0.1` werden 29 (statt 30!) Iterationen durchlaufen;
- für `delta = 0.0001` werden 29999 (statt 30000!) Iterationen durchlaufen.

Warum? $\Rightarrow$ Genauigkeit der `real`-Arithmetik

```
- val a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
val a = 1.0 : real

- a >= 1.0;
val it = false : real

- Real.compare(a, 1.0);
val it = LESS : order
```

Anmerkung zum Beispiel im Abschnitt 7.5.4 (II)

```
- val b = 1.00000000001;
val b = 1.00000000001 : real

- val b = 1.000000000001;
val b = 1.0 : real

- b - 1.0;
val it = 1.00008890058e~12 : real
```

$\leadsto$ intern werden mehr Stellen gespeichert als ausgegeben werden;

$\leadsto$ offensichtlich werden die Zahlen intern nicht im Dezimalsystem gespeichert und verarbeitet

8. Abstraktionsbildung mit neuen Typen

- 8.1 Typen im Überblick
- 8.2 Deklarationen von Typabkürzungen in SML: type-Deklarationen
- 8.3 Definition von Typen: datatype-Deklarationen
- 8.4 Definition von rekursiven Typen
- 8.5 Beweisprinzip der strukturellen Induktion
- 8.6 Beispiele: Implementierungen des Konzepts der „Menge“
- 8.7 Beispiele: Grundlegende Funktionen für binäre (Such-)Bäume

8.1. Typen im Überblick

- Typ als Wertemenge
- Typen mit atomaren und zusammengesetzten Werten
- Typen in Programmiersprachen mit erweiterbaren Typsystemen
- Monomorphe und Polymorphe Typen
- (Wert-)Konstruktoren und (Wert-)Selektoren eines Typs
- Typkonstruktoren

8.1.1. Typ als Wertemenge

Ein Typ (oder Datentyp) bezeichnet eine Menge von Werten. Diese Werte können

- *atomar* (z.B. die Werte des Typs „ganze Zahlen": `int`) oder
- *zusammengesetzt* (z.B. die Werte des Typs „Listen von ganzen Zahlen": `int list`) sein.

Die Wertemenge, die ein Typ repräsentiert, kann

- *endlich* (wie z.B. im Falle des Typs „Boole'sche Werte": `bool`) oder
- *unendlich* (wie im Falle des Typs „ganze Zahlen" `int`) sein.

Mit einem Typ werden üblicherweise Prozeduren zur Bearbeitung der Daten des Typs angeboten.

8.1.2. Typen mit atomaren und zusammengesetzten Werten

Ein Typ kann atomare Werte haben wie z.B. die Typen `bool` und `int`.

Ein Typ kann auch zusammengesetzte Werte haben wie z.B. die Typen `int * int` und der Typ `'a list`.

Typen mit zusammengesetzten Werten werden auch zusammengesetzte Typen genannt, weil sie mit zusammengesetzten Typausdrücken wie etwa `real * real` oder `int list` oder `'a list` bezeichnet werden.

8.1.3. Typen in Programmiersprachen mit erweiterbaren Typsystemen

Ein Typ kann *vordefiniert* sein, d.h. von der Programmiersprache angeboten werden (z.B. `int` und `bool` in SML).

Moderne Programmiersprachen ermöglichen auch die Definition von neuen Typen nach den Anforderungen einer Programmieraufgabe:

$\leadsto$ Sprachen mit „erweiterbarem Typsystem“

Dies ermöglicht z.B. die Definition von Typen „Wochentag“, „Uhrzeit“, „komplexe Zahl“ oder „Übungsgruppe“ (vgl. Abschnitt 5.1).

8.1.4. Monomorphe und Polymorphe Typen

Ein (atomarer oder zusammengesetzter) Typ kann *monomorph* sein.

$\longrightarrow$ keine Typvariablen im Typausdruck

Beispiele: `bool` und `int * int` sind monomorphe Typen

Ein (atomarer oder zusammengesetzter) Typ kann auch *polymorph* sein.

$\longrightarrow$ eine oder mehrere Typvariablen im zugehörigen Typausdruck

Beispiel: `'a list` ist ein polymorpher Typ

8.1.5. (Wert-)Konstruktoren und (Wert-)Selektoren eines Typs

- Zusammen mit einem Typ werden (Wert-)Konstruktoren (manchmal auch Operatoren genannt) definiert.

  Bsp.: Der vordefinierte (polymorphe) Typ „Liste“ hat zwei (Wert-)Konstruktoren: die leere Liste `nil` und den Operator `cons` (`::`).

- (Wert-)Konstruktoren können 0-stellig sein (auch Konstanten genannt) oder eine beliebige andere Stelligkeit haben:
  - Der (Wert-)Konstruktor `nil` des polymorphen Typs „Liste“ ist 0-stellig.
  - Der (Wert-)Konstruktor `cons` (`::`) desselben Typs ist zweistellig (binär).

- Für zusammengesetzte Typen werden auch sogenannte *(Wert-)Selektoren* definiert.

  Die (Wert-)Selektoren des vordefinierten Typs „Liste“ sind die Funktionen `hd` (*head*) und `tl` (*tail*).

  (Wert-)Selektoren werden manchmal auch „Destruktoren“ genannt.

8.1.6. Typkonstruktoren

Zur Definition von Typen werden Typkonstruktoren angeboten, mit denen Typausdrücke zusammengesetzt werden können.

Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionen, werden aber anders verwendet:

- Wird eine Funktion auf aktuelle Parameter angewandt, so geschieht dies, um einen Wert zu berechnen.
- Wird ein Typkonstruktor auf Typausdrücke angewandt, so geschieht dies lediglich, um einen neuen Typausdruck zu bilden und so einen neuen Typ zu definieren.

  Dabei findet keine Auswertung statt!

Vorsicht:
Typkonstruktoren dürfen nicht mit (Wert-)Konstruktoren verwechselt werden.

8.2. Deklarationen von Typabkürzungen in SML: type-Deklarationen

- Typabkürzungen
- Grenzen der Nutzung von Typabkürzungen
- Nützlichkeit von Typabkürzungen
- Polymorphe Typabkürzungen

8.2.1. Typabkürzungen

Im Abschnitt 5.3.2 wurde das folgende Beispiel eines Vektortyps eingeführt:

```
- type punkt = real * real;
type punkt = real * real

- fun abstand(p1: punkt, p2: punkt) =
       let fun quadrat(z) = z * z
           val delta_x = #1(p2) - #1(p1)
           val delta_y = #2(p2) - #2(p1)
       in
           Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
       end;
val abstand = fn : punkt * punkt -> real

- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Typabkürzungen (II)

Mit der `type`-Deklaration

```
- type punkt = real * real;
```

wird die Typkonstante `punkt` als Abkürzung für den Vektortyp `real * real` vereinbart.

Eine solche Abkürzung, „Typabkürzung“ (*type abbreviation*) genannt, spezifiziert keinen neuen Typ, sondern lediglich ein Synonym für einen Typ.

Im Zusammenhang mit Typabkürzungen spricht man auch von „transparenter Typbindung“ (*tranparent type binding*).

8.2.2. Grenzen der Nutzung von Typabkürzungen

Benötigt man z.B. neben Kartesischen Koordinaten auch Polarkoordinaten, dann kann man die folgenden Typabkürzungen vereinbaren:

```
- type kartes_punkt = real * real;

- type polar_punkt = real * real;
```

Da `punkt` und `kartes_punkt` beide denselben Typ `real * real` bezeichnen, ist keine Anpassung der Definition der Funktion `abstand` nötig:

```
- val A = (0.0, 0.0) : kartes_punkt;
val A = (0.0,0.0) : kartes_punkt

- val B = (1.0, 1.0) : kartes_punkt;
val B = (1.0,1.0) : kartes_punkt

- abstand (A, B);
val it = 1.41421356237 : real
```

Grenzen der Nutzung von Typabkürzungen (II)

Dies ist aber gefährlich, weil die Funktion `abstand` auch auf Punkte in Polarkoordinaten angewendet werden kann:

```
- val C = (1.0, Math.pi/2.0) : polar_punkt;
val C = (1.0,1.57079632679) : polar_punkt

- abstand(B, C);
val it = 0.570796326795 : real
```

Der Punkt `C` hat ja die Kartesischen Koordinaten `(0.0, 1.0)`, der Abstand zwischen `B` und `C` ist also in Wirklichkeit `1.0`.

8.2.3. Nützlichkeit von Typabkürzungen: Erstes Beispiel

Würde SML Typabkürzungen nicht ermöglichen, so müsste die Funktion `abstand` wie folgt definiert werden:

```
- fun abstand(p1: real * real, p2: real * real) =
        let fun quadrat(z) = z * z
            val delta_x = #1(p2) - #1(p1)
            val delta_y = #2(p2) - #2(p1)
        in
            Math.sqrt(quadrat(delta_x) + quadrat(delta_y))
        end;
val abstand = fn : (real * real) * (real * real) -> real
```

Diese Definition ist etwas weniger lesbar. Vor allem der ermittelte Typ

```
(real * real) * (real * real) -> real
```

der Funktion `abstand` ist wesentlich schlechter lesbar als

```
punkt * punkt -> real
```

8.2.4. Nützlichkeit von Typabkürzungen: Zweites Beispiel

Ein zweites Beispiel für die Nützlichkeit von Typabkürzungen ist:

```
type person = {Name      : string,
               Vorname   : string,
               Anschrift : string,
               Email     : string,
               Tel       : int};

type kurz_person = {Name    : char,
                    Vorname : char,
                    Tel     : int};

fun kuerzen(x : person) : kurz_person=
     {Name    = String.sub(#Name(x), 0),
      Vorname = String.sub(#Vorname(x), 0),
      Tel     = #Tel(x)};
```

Nützlichkeit von Typabkürzungen: Zweites Beispiel (II)

Obwohl die Typausdrücke

```
person -> kurz_person
```

und

```
{Anschrift:string, Email:string, Name:string,
 Tel:int, Vorname:string}
-> {Name:char, Tel:int, Vorname:char}
```

denselben Typ bezeichnen, ist der erste Ausdruck lesbarer.

Ein Beispiel für die Verwendung der Funktion:

```
- kuerzen {Name="Bry",  Vorname="Francois", Anschrift="D1.02",
           Tel=2210, Email="bry@informatik.uni-muenchen.de"};
val it = {Name=#"B",Tel=2210,Vorname=#"F"} : kurz_person
```

8.2.5. Polymorphe Typabkürzungen

Typvariablen dürfen in `type`-Deklarationen vorkommen:

$\leadsto$ *polymorphe Typabkürzungen* in „parametrische `type`-Deklarationen“:

```
type 'a menge = 'a -> bool;

val ziffer_menge : int menge =
    fn 0 => true
     | 1 => true
     | 2 => true
     | 3 => true
     | 4 => true
     | 5 => true
     | 6 => true
     | 7 => true
     | 8 => true
     | 9 => true
     | _ => false;
```

[eine Menge wird hier durch eine charakteristische Funktion realisiert]

8.3. Definition von Typen: datatype-Deklarationen

- Eine Typabkürzung definiert keinen neuen Typ.

- Neue Typen können in SML mit `datatype`- und `abstype`-Deklarationen vereinbart werden.

Im Folgenden werden Definitionen von neuen Typen mit `datatype` behandelt.

8.3.1. Definition von Typen mit endlich vielen atomaren Werten

Ein Typ „Farbe“ bestehend aus der Wertemenge $\{\texttt{Rot}, \texttt{Gelb}, \texttt{Blau}\}$ kann in SML wie folgt definiert werden:

```
- datatype Farbe = Rot | Gelb | Blau;
datatype Farbe = Blau | Gelb | Rot

- Rot;
val it = Rot : Farbe
```

Diese Deklaration legt das Folgende fest:

- Der Name `Farbe` ist eine Typkonstante.
- Die Typkonstante `Farbe` wird an die Wertemenge $\{\texttt{Rot}, \texttt{Gelb}, \texttt{Blau}\}$ gebunden.
- Die Namen `Rot`, `Gelb` und `Blau` sind 0-stellige (Wert-)Konstruktoren.

$\rightsquigarrow$ auch *Aufzählungstyp* genannt

Definition von Typen mit endlich vielen atomaren Werten (II)

- Mit `datatype` deklarierte Typen haben keine Entsprechung in den vordefinierten Typen.
- Ihre (Wert-)Konstruktoren können genauso wie die von vordefinierten Typen verwendet werden, z.B. für Pattern Matching:

```
 - fun farbname(Rot)  = "rot"
     | farbname(Gelb) = "gelb"
     | farbname(Blau) = "blau";
 val farbname = fn : Farbe -> string

- farbname(Gelb);
val it = "gelb" : string

- [Rot, Blau];
val it = [Rot,Blau] : Farbe list
```

Definition von Typen mit endlich vielen atomaren Werten (III)

In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, stets über die Gleichheit, die implizit bei der `datatype`-Deklaration mit definiert wird:

```
- Blau = Blau;
val it = true : bool

- Blau = Rot;
val it = false : bool
```

Definition von Typen mit endlich vielen atomaren Werten (IV)

In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, über keine implizit definierte Ordnung:

```
- Rot < Gelb;
Error: overloaded variable not defined at type
   symbol: <
   type: Farbe
```

Die folgenden Typdeklarationen sind also in SML austauschbar:

```
- datatype Farbe = Rot | Gelb | Blau;

- datatype Farbe = Gelb | Rot | Blau;
```

Andere Programmiersprachen würden aus der Reihenfolge der (Wert-)Konstruktoren in der Typdefinition die Ordnung `Rot` $<$ `Gelb` $<$ `Blau` implizit definieren.

8.3.2. Definition von Typen mit zusammengesetzten Werten

Nicht nur Typen, die Mengen von atomaren Werte bezeichnen, können definiert werden, sondern auch Typen mit zusammengesetzten Werten:

```
- datatype Preis = DM of real | EURO of real;
datatype Preis = DM of real | EURO of real
```

Diese Deklaration legt das Folgende fest:

- Der Name `Preis` ist eine Typkonstante.
- Die Typkonstante `Preis` wird an die folgende Wertemenge gebunden:

$$\{\ \mathtt{DM}(\mathtt{x}) \mid \mathtt{x} \in \mathbb{R}\ \} \cup \{\ \mathtt{EURO}(\mathtt{x}) \mid \mathtt{x} \in \mathbb{R}\ \}$$

- Die Namen `DM` und `EURO` sind unäre (einstellige) (Wert-)Konstruktoren, beide vom Typ `real -> Preis`.

Definition von Typen mit zusammengesetzten Werten (II)

```
- DM(4.5);
val it = DM 4.5 : Preis

- EURO(2.0);
val it = EURO 2.0 : Preis

- DM(1);
Error: operator and operand don't agree [literal]
   operator domain: real
   operand:         int
   in expression:
     DM 1
```

Man beachte die Syntax „`k of t2`" in der Definition eines (Wert-)Konstruktors für einen Typ `t1` mit Bereich `t2` anstelle der Schreibweise `k(t2)` oder `k t2`:

```
datatype t1 = ... | k of t2 | ...
```

[Diese Syntax unterstreicht, dass ein (Wert-)Konstruktor eines Typs keine gewöhnliche Funktion ist.]

Definition von Typen mit zusammengesetzten Werten (III)

Pattern Matching ist die bevorzugte Weise, Funktionen auf benutzerdefinierten Typen mit zusammengesetzten Werten zu definieren:

```
- fun wechseln(DM(x))   = EURO(0.51129 * x)
    | wechseln(EURO(x)) = DM(1.95583 * x);
val wechseln = fn : Preis -> Preis
```

Es ist empfehlenswert, die Fälle beim Pattern Matching in der Reihenfolge der Typdefinition aufzulisten.

Unter Verwendung der vordefinierten Funktionen `real` und `round` kann `wechseln` verbessert werden, um auf die zweite Nachkommastelle zu runden:

```
fun wechseln(DM(x))   = EURO(real(round(0.51129 * x * 1e5)) * 1e~5)
  | wechseln(EURO(x)) = DM(real(round(1.95583 * x * 1e5)) * 1e~5);
```

8.3.3. Gleichheit für Typen mit zusammengesetzten Werten

In SML verfügen auch benutzerdefinierte Typen, die Mengen von zusammengesetzten Werten bezeichnen, über die Gleichheit. Sie ist komponentenweise definiert:

```
- datatype zeitpunkt = Sek of int | Min of int | Std of int;
datatype zeitpunkt = Min of int | Sek of int | Std of int

- Sek(30) = Sek(0030);
val it = true : bool

- Min(60) = Std(1);
val it = false : bool

- Sek 2 = Sek(2);
val it = true : bool
```

Gleichheit für Typen mit zusammengesetzten Werten (II)

Die Gleichheit ist auf benutzerdefinierten Typen mit zusammengesetzten Werten komponentenweise definiert:

Zwei Werte sind gleich, wenn

- ihre (Wert-)Konstruktoren gleich sind (was für `Min(60)` und `Std(1)` nicht der Fall ist) und
- diese (Wert-)Konstruktoren auf identische Vektoren angewandt werden.

  Dabei muss berücksichtigt werden, dass in SML ein einstelliger Vektor mit seiner einzigen Komponente gleich ist (z.B. sind in SML `2` und `(2)` gleich).

Gleichheit für Typen mit zusammengesetzten Werten (III)

Ist die Gleichheit auf dem Typ einer Komponente nicht definiert, so ist sie es auch nicht auf dem benutzerdefinierten Typ:

```
- DM(2.0) = DM(2.0);
Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         Preis * Preis
  in expression:
    DM 2.0 = DM 2.0
```

Wir erinnern daran, dass in SML die Gleichheit über den Gleitkommazahlen nicht vordefiniert ist (siehe Abschnitt 5.2.2).

8.3.4. „Typenmix“

Typdeklarationen ermöglichen es, Funktionen mit Parametern unterschiedlicher Grundtypen zu definieren:

```
- datatype nb = Int of int | Real of real;
datatype nb = Int of int | Real of real

- fun round(Int(x))  = Int(x)
    | round(Real(x)) = Int(trunc(x));
val round = fn : nb -> nb

- round(Int(56));
val it = Int 56 : nb

- round(Real(56.8976));
val it = Int 56 : nb
```

„Typenmix“ (II)

Ein weiteres Beispiel der Verwendung des Typs nb:

```
- datatype nb = Int of int | Real of real;

- local fun real_abl f x  =
              let
                  val delta = 1E~10
              in
                  (f(x + delta) - f(x)) / delta
              end;
        fun konvertieren(Int(x))  = real(x)
          | konvertieren(Real(x)) = x
  in
    fun abl f (Int(x))    = Real(real_abl f  (real(x)))
      | abl f (Real(x)) =  Real(real_abl f x)
  end;
val abl = fn : (real -> real) -> nb -> nb
```

„Typenmix“ (III)

```
- abl (fn x => 2.0 * x) (Int(5));
val it = Real 2.00000016548 : nb

- abl (fn x => 2.0 * x) (Real(5.0));
val it = Real 2.00000016548 : nb
```

Man beachte, dass Ausdrücke, die mit den (Wert-)Konstruktoren `Int` und `Real` des Typs `nb` aufgebaut sind, in Klammern vorkommen.

Dies ist wegen der Präzedenzen in SML notwendig!

8.4. Definition von rekursiven Typen

Ein Typ kann auch Werte unbegrenzter (aber endlicher) Größe haben.

- ein bereits bekanntes Beispiel: Listen
- ein anderes Beispiel: die sogenannten „Binärbäume“

8.4.1. Wurzel, Blätter, Äste, Bäume und Wälder

Zunächst definieren wir die benötigten Begriffe:

Definition ((gerichteter) Graph)

Ein *(gerichteter) Graph* G ist ein Paar (Kn, Ka) mit $Ka \subseteq Kn \times Kn$.

- Die Elemente von Kn werden *Knoten* von G genannt.
- Die Elemente von Ka sind die *Kanten* von G.
- Ist Kn leer, so sagt man, der Graph $G = (Kn, Ka) = (\{\}, \{\})$ sei leer.
- Ist $(k_1, k_2) \in Ka$, so heißt k_2 ein *Nachfolger* von k_1 (in G).

Wurzel, Blätter, Äste, Bäume und Wälder (II)

Definition (zusammenhängender Graph)

Ein (gerichteter) Graph $G = (Kn, Ka)$ heißt *zusammenhängend*, wenn es für jedes Paar (k_a, k_e) von Knoten eine Folge (Pfad genannt) von Knoten $k_1, \ldots, k_m \in Kn$ $(m \geq 1)$ gibt, so dass:

- $k_1 = k_a$
- $k_m = k_e$
- für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m - 1$ ist $(k_i, k_{i+1}) \in Ka$ oder $(k_{i+1}, k_i) \in Ka$.

In einem zusammenhängenden Graph G kann man von jedem Knoten von G über die Kanten von G jeden anderen Knoten von G erreichen (wobei die Kanten in beliebiger Richtung durchlaufen werden dürfen).

Wurzel, Blätter, Äste, Bäume und Wälder (III)

Definition (Zyklus in einem Graph)

Ein *Zyklus* in einem Graph $G = (Kn, Ka)$ ist eine endliche Folge $k_1, \ldots, k_m$ $(m \geq 1)$ von Knoten von G, so dass

- für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m - 1$ ist $(k_i, k_{i+1}) \in Ka$
- $(k_m, k_1) \in Ka$

Ein Zyklus ist ein Rundgang durch den Graph über die Kanten des Graphen, bei dem die Richtung der Kanten eingehalten wird.

Wurzel, Blätter, Äste, Bäume und Wälder (IV)

Definition (Baum, Wurzel)

Sei K eine Menge von Knoten. *Bäume (mit Knoten in K)* sind:

1. Der leere Graph $(\{\}, \{\})$ ist ein Baum. Dieser Baum hat keine Wurzel.
2. Für jedes $k \in K$ ist $(\{k\}, \{\})$ ein Baum. Die Wurzel dieses Baums ist der Knoten k.
3. Ist $m \in \mathbb{N} \setminus \{0\}$ und ist (Kn_i, Ka_i) für jedes $i \in \mathbb{N}$ mit $1 \leq i \leq m$ ein Baum mit Wurzel k_i, so dass die Knotenmengen Kn_i paarweise disjunkt sind, und ist $k \in K$ ein „neuer“ Knoten, der in keinem Kn_i vorkommt, so ist (Kn, Ka) ein Baum (mit Wurzel k):

$$\begin{aligned} Kn &= \{\, k \,\} \cup \bigcup_{i=1}^{n} Kn_i \\ Ka &= \{\, (k, k_i) \mid 1 \leq i \leq m \,\} \cup \bigcup_{i=1}^{n} Ka_i \end{aligned}$$

Wurzel, Blätter, Äste, Bäume und Wälder (V)

Definition (Blatt)

Die Knoten eines Baumes, die keine Nachfolger haben, heißen *Blätter*.

Definition (Binärbaum)

Ein *Binärbaum* B ist ein Baum mit der Eigenschaft:

Jeder Knoten von B ist entweder ein Blatt oder hat genau zwei Nachfolger.

Wurzel, Blätter, Äste, Bäume und Wälder (VI)

Man kann leicht (strukturell induktiv) beweisen:

- Jeder nichtleere Baum hat eine Wurzel hat.

  Von der Wurzel kann jeder andere Knoten des Baumes über die Kanten des Baumes erreicht werden, wobei die Richtung der Kanten eingehalten wird.

- In einem Baum kommt kein Zyklus vor.

- Jeder Baum enthält einen Pfad von seiner Wurzel zu jedem seiner Blätter enthält.

  Ein Wurzel-Blatt-Pfad in einem Baum heißt *Ast*.

Wurzel, Blätter, Äste, Bäume und Wälder (VII)

Definition (Wald)

Ein *Wald* ist eine Menge von Bäumen, deren Knotenmengen paarweise disjunkt sind.

- Bäume und Wälder werden in der Informatik häufig verwendet.
- Fast alle Bäume werden graphisch so dargestellt, dass ihre Wurzel oben und ihre Blätter unten sind.

  [Eine seltene Ausnahme stellen die Beweisbäume dar (siehe Abschnitt 6.7.4).]

8.4.2. Induktive Definition

Eine Definition wie die vorangehende Definition der Bäume heißt *induktive Definition*.

Induktive Definitionen setzen sich zusammen aus:

- einem oder mehreren Basisfällen,

  [wie der Fall 2 der vorangehenden Definition]

- und einem oder mehreren Induktionsfällen

  [wie der Fall 3 der vorangehenden Definition]

- und ggf. Sonderfällen

  [wie der Fall 1 der vorangehenden Definition]

Die Basisfälle bestimmen Anfangsstrukturen. Die Induktionsfälle sind Aufbauregeln.

8.4.3. Induktive Definition und rekursive Algorithmen

Für induktiv definierte Datenstrukturen lassen sich Algorithmen leicht rekursiv spezifizieren.

Die Anzahl der Knoten eines Baumes kann z.B. wie folgt rekursiv ermittelt werden:

1. Der leere Baum hat 0 Knoten.
2. Ein Baum der Gestalt $(\{k\}, \{\})$ hat 1 Knoten.
3. Ein Baum der Gestalt (Kn, Ka) mit

$$\begin{aligned} Kn &= \{\, k \,\} \cup \bigcup_{i=1}^{n} Kn_i \\ Ka &= \{\ (k, k_i) \mid 1 \leq i \leq m\ \} \cup \bigcup_{i=1}^{n} Ka_i \end{aligned}$$

   hat einen Knoten mehr als die Summe der Knotenanzahlen der Bäume (Kn_i, Ka_i) mit $1 \leq i \leq m$.

8.4.4. Darstellung von Bäumen: graphisch und durch Ausdrücke

Graphische Darstellung:

Die Knoten werden durch Symbole und die Kanten als Verbindung dazwischen darstellt (i.d.R. mit der Richtung von oben nach unten).

```
           k1
         /    \
      k2        k3
     /  \      /  \
   k4   k5   k6    k7
                  /  \
                k8   k9
```

Dieser Baum ist ein Binärbaum mit Wurzel `k1` und Blättern `k4`, `k5`, `k6`, `k8`, `k9`.

Sind die Bezeichner der Knoten irrelevant, werden sie einfach durch einen kleinen Kreis oder ein Symbol wie $*$ ersetzt.

Darstellung von Bäumen: graphisch und durch Ausdrücke (II)

Oft wird den Knoten im Baum ein Wert zugeordnet, dann zeichnet man einfach die Markierungen statt der Knoten oder $*$.

```
      4
    /   \
  1       5
 / \     / \
1   3   5   8
           / \
          8   9
```

Die Werte, mit denen die Knoten in diesem Fall markiert sind, können mehrfach im Baum vorkommen, während die Knoten selbst nicht mehrfach vorkommen können.

Darstellung von Bäumen: graphisch und durch Ausdrücke (III)

Es können auch Bäume vorkommen, bei denen nur die Blätter mit Werten markiert sind, so dass sich eine Mischform zur Darstellung ergibt:

```
        *
      /   \
    *       *
   / \     / \
  1   3   5   *
             / \
            8   9
```

Darstellung von Bäumen: graphisch und durch Ausdrücke (IV)

Darstellung durch Ausdrücke:

Verschachtelung von Teilausdrücken repräsentiert die Struktur eines Baums, z.B.:

```
Knt(Knt(Blt,
        Blt),
    Knt(Blt,
        Knt(Blt,
            Blt)))
```

Dieser Ausdruck ist mit zwei verschiedenen Symbolen für Blätter und Knoten gebildet.

- Oft entspricht die Reihenfolge der Argumente in den Ausdrücken der Reihenfolge der Nachfolger in der graphischen Darstellung.
- Durch Einrückungen kann man zudem die Struktur von verschachtelten Ausdrücken verdeutlicht.

Darstellung von Bäumen: graphisch und durch Ausdrücke (V)

Eine Darstellung für die Baum-Varianten mit Markierungen erfordert lediglich zusätzliche Argumente für die Markierungen:

```
Knt(4,
    Knt(1,
        Blt(1),
        Blt(3)),
    Knt(5,
        Blt(5),
        Knt(8,
            Blt(8),
            Blt(9))))
```

Natürlich kann man hier ebenso wie in der graphischen Darstellung auch Varianten betrachten, in denen nur die Blätter mit Werten markiert sind.

8.4.5. Rekursive Typen zum Ausdrücken von induktiven Definitionen — Der Binärbaum

In SML können nichtleere Binärbäume, deren Blätter mit ganzen Zahlen markiert sind, wie folgt spezifiziert werden:

```
- datatype binbaum1 = Blt of int                      (* Blatt  *)
                    | Knt of binbaum1 * binbaum1;  (* Knoten *)
datatype binbaum1 = Blt of int
                  | Knt of binbaum1 * binbaum1

- val b = Knt(Knt(Blt(1), Blt(3)),
              Knt(Blt(5), Knt(Blt(8), Blt(9))));
val b = Knt (Knt (Blt #,Blt #),
             Knt (Blt #,Knt #)) : binbaum1
```

Der Wert von `b` wird hier abgekürzt gedruckt; in der globalen Umgebung ist der richtige Wert abgespeichert.

Rekursive Typen — Der Binärbaum (II)

Dieser Binärbaum kann wie folgt graphisch dargestellt werden:

```
                          *
                        /   \
Knt(Blt(1), Blt(3))  *       *  Knt(Blt(5), Knt(Blt(8), Blt(9)))
                    / \     / \
                   1   3   5   *  Knt(Blt(8), Blt(9))
                              / \
                             8   9
```

Der (Wert-)Konstruktor Knt bildet aus zwei Binärbäumen einen neuen Binärbaum. Sein Typ ist also `binbaum1 * binbaum1 -> binbaum1`.

Ein solcher Binärbaum wird manchmal auch „beblättert“ genannt, weil seine Blätter und nur seine Blätter Markierungen (d.h. Werte) tragen.

Rekursive Typen — Der Binärbaum (III)

- Die obige Typdefinition der Binärbäume hat genau dieselbe Struktur wie die Definition der Bäume in Abschnitt 8.4.1.
- Zusätzlich besteht die Einschränkung für Binärbäume, dass jeder Knoten, der kein Blatt ist, genau zwei Nachfolger haben muss.
- Syntaktisch unterscheidet sie sich von der Definition in Abschnitt 8.4.1 in der Verwendung von SML-Konstrukten.
- Eine Typdefinition wie die Definition des Typs `binbaum1` ist also eine induktive Definition.
- Wegen ihrer syntaktischen Ähnlichkeit mit rekursiven Algorithmen werden aber solche Typdefinition manchmal *rekursive Typdefinitionen* genannt.
- Die Typen, die mittels induktiven (oder rekursiven) Typdefinitionen vereinbart werden, heißen *rekursive Typen*.

Rekursive Typen — Der Binärbaum (IV)

Die Funktion `blaetterzahl` zur Berechnung der Anzahl der Blätter eines Binärbaums vom Typ `binbaum1` kann in SML wie folgt implementiert werden:

```
- fun blaetterzahl  (Blt(_))      = 1
    | blaetterzahl (Knt(b1, b2)) =   blaetterzahl b1
                                   + blaetterzahl b2;
val blaetterzahl = fn : binbaum1 -> int

- val c = Knt(Blt(1), Blt(2));

- blaetterzahl(c);
val it = 2 : int

- blaetterzahl(Knt(c, Knt(c, c)));
val it = 6 : int
```

8.4.6. Polymorphe Typen

Es ist oft vorteilhaft, rekursive Typen polymorph zu definieren.
Ein polymorpher Typ `binbaum1` kann wie folgt definiert werden:

```
- datatype 'a binbaum2 = Blt of 'a
                       | Knt of 'a binbaum2 * 'a binbaum2;
```

Die Funktion `blaetterzahl` kann nun als polymorphe Funktion redefiniert werden:

```
- fun blaetterzahl  (Blt(_))       = 1
    | blaetterzahl (Knt(b1, b2)) =   blaetterzahl b1
                                   + blaetterzahl b2;
val blaetterzahl = fn : 'a binbaum2 -> int

- val d = Knt(Blt("ab"), Blt("cd"));
val d = Knt (Blt "ab",Blt "cd") : string binbaum2

- blaetterzahl(d);
val it = 2 : int
```

Polymorphe Typen (II)

```
- let val e = Knt(d, d);
      val f = Knt(e, e)
  in
      blaetterzahl(Knt(f, f))
  end;
val it = 16 : int

- val g = Knt(Blt([1,2,3]), Blt[4,5]);
val g = Knt (Blt [1,2,3],Blt [4,5]) : int list binbaum2

- let val h = Knt(g, g);
      val i = Knt(h, h)
  in
      blaetterzahl(Knt(i, i))
  end;
val it = 16 : int
```

8.4.7. Suche in Binärbäumen

Mit dem folgenden Prädikat kann überprüft werden, ob ein Element in einem Binärbaum vorkommt:

```
- fun suche(x, Blt(M))       = (x = M)
    | suche(x, Knt(B1, B2)) = suche(x, B1)
                              orelse suche(x, B2);
val suche = fn : 'a * binbaum2 -> bool

- val b = Knt(Knt(Blt(1), Blt(3)), Knt(Blt(5), Knt(Blt(8),Blt(9))));
val b = Knt (Knt (Blt #,Blt #),Knt (Blt #,Knt #)) : int binbaum2

- suche(5, b);
val it = true : bool

- suche(12, b);
val it = false : bool
```

8.4.8. Die Liste als Binärbaum

Die Liste ist ein Sonderfall des Binärbaums, wie der Vergleich der folgenden Darstellungen erkennen lässt:

```
    ::                    *
   /  \                  /   \
  1    ::               1     *
      /  \                   /   \
     2    ::                2     *
         /  \                    /   \
        3    nil                3     nil
```

Die Suche in einem Binärbaum einer solchen Gestalt besteht aus denselben Schritten wie die Suche in einer Liste mit der Funktion `member`.

8.5. Beweisprinzip der strukturellen Induktion

Wie können Eigenschaften von Funktionen bewiesen werden, die auf rekursiven Typen definiert sind?

Induktionsbeweise (vgl. Abschnitt 1.1.8) und induktive Definitionen haben eine ähnliche Gestalt:

- Induktionsbeweise und induktive Definitionen weisen Basisfälle und Induktionsfälle auf.
- Sowohl bei Induktionsbeweisen als auch bei induktiven Definitionen stellen die Induktionsfälle eine Art stufenweisen Aufbau dar.

Eigenschaften von Funktionen, die auf rekursiven Typen definiert sind, lassen sich oft induktiv beweisen.

$\leadsto$ *strukturelle Induktion*

Beweisprinzip der strukturellen Induktion (II)

Beweisschema für strukturelle Induktion:

Sei t ein rekursiver Typ mit den (Wert-)Konstruktoren $\mathtt{k}_i^{s_i}$ $(0 \leq i \leq I)$ wobei s_i die Stelligkeit des (Wert-)Konstruktors $\mathtt{k}_i^{s_i}$ ist.

Um zu zeigen, dass alle Werte des Typs t eine Eigenschaft $\mathbf{E}$ (z.B.: „die Auswertung einer Anwendung der Funktion f terminiert") besitzen, genügt es zu zeigen:

1. Basisfälle:
   Jeder 0-stellige (Wert-)Konstruktor $\mathtt{k}_i^0$ $(0 \leq i \leq I)$ besitzt die Eigenschaft $\mathbf{E}$.
2. Induktionsfälle:
   Für jeden (Wert-)Konstruktor $\mathtt{k}_i^{s_i}$ (der Stelligkeit s_i) mit $i \geq 1$ gilt: wenn s_i Werte $W_1, \ldots, W_{s_i}$ des Typs t die Eigenschaft $\mathbf{E}$ besitzen (Induktionsannahme), dann besitzt auch der Wert $\mathtt{k}_i^{s_i}(W_1, \ldots, W_{s_i})$ des Typs t die Eigenschaft $\mathbf{E}$.

Das Beweisprinzip der strukturellen Induktion lässt sich (allerdings nicht trivial) auf die vollständige Induktion zurückführen.

Beweisprinzip der strukturellen Induktion (III)

Beweis der Eigenschaft: Jede Anwendung der polymorphen Funktion `blaetterzahl` auf einen Binärbaum vom Typ `binbaum1` terminiert.

Basisfall: Hat ein Ausdruck `A` die Gestalt `Blt(x)`, so führt die Auswertung von `blaetterzahl(A)` nach Definition (der Funktion `blaetterzahl`) zur Auswertung von `1` — terminiert!

Induktionsfall: Seien `W1` und `W2` zwei Werte vom Typ `binbaum1`.

Induktionsannahme: Die Auswertungen von `blaetterzahl(W1)` und von `blaetterzahl(W2)` terminieren.

Nach Definition der Funktion führt `blaetterzahl(Knt(W1, W2))` zur Auswertung von `blaetterzahl(W1) + blaetterzahl(W2)`.

Nach Induktionsannahme terminiert die Auswertung der beiden Komponenten dieser Addition. Folglich terminiert auch die Auswertung dieses Ausdrucks.

qed.

8.6. Beispiele: Implementierungen des Konzepts der „Menge“

Am Beispiel des mathematischen Begriffs „Menge“ (der zur Lösung vieler praktischer Aufgaben nützlich ist) werden verschiedene Implementierungen diskutiert.

- Was ist eine „Menge“?
- Was ist eine „Datenstruktur”?
- Die Menge als charakteristische Funktion
- Die Menge als Liste
- Die Menge als sortierte Liste
- Die Menge als binärer Suchbaum

8.6.1. Was ist eine „Menge“

- Die „Menge“ ist ein grundlegender Begriff der Mathematik.
- Eine Menge stellt eine Zusammenfassung von Objekten (*Elemente*) dar.
- In einer Menge gibt es keine Reihenfolge oder sonstige Strukturierung der Elemente.

Referenzmenge

Eine Menge wird immer bezüglich einer „Referenzmenge“ definiert, d.h. einer „Urmenge“, woraus die Elemente der zu definierenden Mengen stammen.

Ohne eine Referenzmenge gäbe es Paradoxien wie etwa das folgende:

> Sei `M` die (Pseudo-)Menge, die alle Mengen umfasst, die nicht Element von sich selbst sind: Gilt $\mathtt{M} \in \mathtt{M}$?
>
> — Falls ja, dann gilt nach Definition von `M`: $\mathtt{M} \notin \mathtt{M}$, ein Widerspruch.
>
> — Falls nein, dann gilt nach Definition von `M`: $\mathtt{M} \in \mathtt{M}$, ein Widerspruch.

Die Bedingung, dass keine Menge `M` definiert werden kann, ohne eine Referenzmenge festzulegen, schließt einen solchen Fall aus.

In einer typisierten Programmiersprache stellen die Typen geeignete Kandidaten für etwaige Referenzmengen dar.

Extensional und intensional definierte Mengen

- Eine Menge wird *extensional* (oder *explizit*) definiert, wenn ihre Definition aus einer Auflistung ihrer Elemente besteht.

  $\{1.0, 23.5, 12.45\}$ ist z.B. eine extensional definierte Menge mit Referenzmenge $\mathbb{R}$.

- Eine Menge wird *intensional* (oder *implizit*) definiert, wenn ihre Elemente durch eine Eigenschaft charakterisiert werden.

  $\{\, x * x \mid x \in \mathbb{N} \,\}$ ist z.B. eine intensional definierte Menge mit Referenzmenge $\mathbb{N}$.

Funktionen sind in einer Programmiersprache das Gegenstück zu intensional definierten Mengen. Die Funktion

```
fun quadrat(x : int) = x * x
```

drückt in SML die Menge $\{\, x * x \mid x \in \mathbb{Z} \,\}$ als Menge möglicher Ergebniswerte aus.

Die Datenstruktur „Menge“, die es zu implementieren gilt, kann sinnvollerweise also nur extensional definierte, zudem endliche Mengen wiedergeben.

Mengenoperationen

Mit dem Begriff „Menge“ werden die folgenden grundlegenden Operationen definiert:

Elementbeziehung:	$\in$
Vereinigung:	$M_1 \cup M_2 = \{\, x \,\vert\, x \in M_1 \vee x \in M_2 \,\}$
Durchschnitt:	$M_1 \cap M_2 = \{x \,\vert\, x \in M_1 \wedge x \in M_2 \,\}$
Gleichheit:	Zwei Mengen sind gleich, wenn sie genau dieselben Elemente haben.
Teilmengenbeziehung:	M_1 ist eine Teilmenge von M_2 ($M_1 \subseteq M_2$), wenn jedes Element von M_1 auch Element von M_2 ist.

Die „leere Menge“ ist eine ausgezeichnete Menge, die keine Elemente hat.

8.6.2. Was ist eine „Datenstruktur”?

Unter „Datenstruktur“ versteht man in der Informatik

- eine Darstellung einer mathematischen Struktur (wie z.B. Mengen, Vektoren oder Listen) in einer Programmiersprache zusammen mit
- der Implementierung der grundlegenden Operationen dieser Struktur in derselben Programmiersprache.

In einer typisierten Programmiersprache wie SML ist die Definition eines Typs ein gewöhnlicher Teil der Implementierung einer Datenstruktur.

Die Definition eines Typs allein reicht i.d.R. nicht aus, weil damit die grundlegenden Operationen der betrachteten Struktur noch lange nicht gegeben sind.

In der Praxis besteht die Implementierung einer Datenstruktur typischerweise aus einem Typ und Prozeduren (die nicht immer Funktionen sind) für diesen Typ.

8.6.3. Die Menge als charakteristische Funktion

In Abschnitt 8.2.5 hatten wir die Menge der Ziffern wie folgt implementiert:

```
type 'a menge = 'a -> bool;

val ziffer_menge : int menge =  fn 0 => true
                                 | 1 => true
                                 | 2 => true
                                 | 3 => true
                                 | 4 => true
                                 | 5 => true
                                 | 6 => true
                                 | 7 => true
                                 | 8 => true
                                 | 9 => true
                                 | _ => false;
```

Eine solche Funktion nennt man *charakteristische Funktion der Menge* (aller Ziffern).

Die Menge als charakteristische Funktion (II)

Diese Implementierung gibt die Elementbeziehung unmittelbar wieder.

Die Vereinigung und der Durchschnitt lassen sich sehr einfach wie folgt realisieren:

```
- fun vereinigung(m1:'a menge, m2:'a menge) =
                              fn x => m1(x) orelse m2(x);
val vereinigung = fn : 'a menge * 'a menge -> 'a -> bool

- fun durchschnitt(m1:'a menge, m2:'a menge) =
                              fn x => m1(x) andalso m2(x);
val durchschnitt = fn : 'a menge * 'a menge -> 'a -> bool
```

Die Menge als charakteristische Funktion (III)

```
- val M1 : string menge =  fn "ab" => true
                            | "bc" => true
                            | "be" => true
                            | _    => false;
val M1 = fn : string menge

- val M2 : string menge =  fn "de" => true
                            | "fg" => true
                            | "be" => true
                            | _    => false;
val M2 = fn : string menge

- vereinigung(M1, M2);
val it = fn : string -> bool

- vereinigung(M1, M2)("be");
val it = true : bool

- durchschnitt(M1, M2)("ab");
val it = false : bool
```

Die Menge als charakteristische Funktion (IV)

- Diese Implementierung ist zur Implementierung der Gleichheit (von Mengen) und der Teilmengenbeziehung ungeeignet.

- Zur Implementierung von Gleichheit und Teilmengenbeziehung ist ein Vergleich der einzelnen Elemente erforderlich.

  In der obigen Implementierung ist ein solcher Vergleich schwierig, da die Auflistung im Programm selbst statt in einem Aufrufparameter vorkommt.

8.6.4. Die Menge als Liste

Es bietet sich an, eine extensional definierte, endliche Menge als Liste darzustellen.

Die Elementbeziehung wird durch das polymorphe Prädikat `member` implementiert:

```
- fun member(x, nil)          = false
    | member(x, head::tail)   = if x = head
                                  then true
                                  else member(x, tail);
val member = fn : ''a * ''a list -> bool

- member(3,[1,2,3,4]);
val it = true : bool
```

Die Menge als Liste (II)

Abschätzung des Zeitbedarfs von `member`:

- Als Schätzung der Problemgröße bietet sich die Größe (Kardinalität) der betrachteten Menge `M` an ($n = |\texttt{M}|$).
- Als Zeiteinheit nehmen wir die Anzahl der rekursiven Aufrufe des Prädikats `member`.
- Zur Überprüfung, ob `x` Element der Menge `M` ist, wird man bestenfalls `member` einmal, schlechtestenfalls $(n+1)$–mal aufrufen.

  Der Zeitbedarf einer Überprüfung einer Elementbeziehung ist also schlechtestenfalls $O(n)$.

Die Menge als Liste (III)

Die Vereinigung wird durch die polymorphe Funktion `append` (in SML vordefiniert als Infixfunktion „`@`") implementiert:

```
- fun append(nil,    L) = L
    | append(h :: t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list

- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

Diese Implementierung der Vereinigung mag für manche Anwendungen unbefriedigend sein, weil sie die Wiederholung von Elementen nicht ausschließt.

Die Menge als Liste (IV)

Der Durchschnitt kann wie folgt implementiert werden:

```
- fun durchschnitt(nil,    _) = nil
    | durchschnitt(h :: t, L) = if member(h, L)
                                then h :: durchschnitt(t, L)
                                else durchschnitt(t, L);
val durchschnitt = fn : ''a list * ''a list -> ''a list

- durchschnitt([1,2,3,4], [3,4,5,6]);
val it = [3,4] : int list
```

Die Menge als Liste (V)

Die Teilmengenbeziehung kann wie folgt implementiert werden:

```
- fun teilmenge(nil,    _) = true
    | teilmenge(h :: t, L) = if member(h, L)
                             then teilmenge(t, L)
                             else false;
val teilmenge = fn : ''a list * ''a list -> bool

- teilmenge([6,4,2],[1,2,8,4,9,6,73,5]);
val it = true : bool

- teilmenge([4,2,3,1], [3,6,5,4]);
val it = false : bool
```

Um die Menge zu verändern, stehen die Funktionen `cons` (in SML als Infix-Operator `::` vordefiniert), `head` (bzw. `hd`) und `tail` (bzw. `tl`) zur Verfügung.

8.6.5. Die Menge als sortierte Liste

Die zuvor betrachtete Implementierung der Menge setzt keine bestimmte Reihenfolge der Listenelemente voraus.

Eine Sortierung der Elemente ist nur möglich, wenn eine Ordnung (d.h. eine reflexive, transitive und antisymmetrische Relation) über der Referenzmenge vorhanden ist, die zudem total ist.

Im weiteren nehmen wir eine solche Ordnung als gegeben an.

Ist eine Sortierung der Elemente (nach der Ordnung der Referenzmenge) von Vorteil?

$\leadsto$ Sind die Elemente nach aufsteigenden (bzw. absteigenden) Werten sortiert, so kann die sequenzielle Suche durch die Liste abgebrochen werden, sobald ein Listenelement gefunden wird, das größer (bzw. kleiner) als das gesuchte Element ist.

Die Menge als sortierte Liste (II)

Unter der Annahme, dass die Listenelemente nach aufsteigenden Werten sortiert sind, kann die verbesserte Suche wie folgt implementiert werden:

```
- fun member_sort(x, nil)    = false
    | member_sort(x, h::t)  =
       if x < h
       then false
       else if x = h
            then true
            else member_sort(x, t);
val member_sort = fn : int * int list -> bool
```

Zeitbedarf dieser Funktion?

Schlechtestenfalls muss auch mit `member_sort` die (sortierte) Liste ganz durchlaufen werden: Zeitbedarf im schlechtesten Fall also auch $O(n)$ (mit n Länge der Liste).

Die Menge als sortierte Liste (III)

Frage:

Kann eine Sortierung der Elemente prinzipiell eine effizientere Suche als die sequenzielle Suche mit `member_sort` ermöglichen?

Antwort: (an einem Beispiel aus dem Alltag)

Suche einer Telefonnummer zu einem gegebenen Namen im Telefonbuch:

- Sucht man die Telefonnummer eines Herrn Zimmermann, schlägt man das Telefonbuch am Ende auf.
- Sucht man die Telefonnummer eines Herrn Klein, schlägt man das Telefonbuch ungefähr in der Mitte auf.
- Sucht man die Telefonnummer einer Frau Ackermann, schlägt man das Telefonbuch vorne auf.

Aufgrund der bekannten Sortierreihenfolge kann man vom gesuchten Namen auf die ungefähre Position im Telefonbuch schließen und damit die Suche erheblich einschränken.

Die Menge als sortierte Liste (IV)

Diese deutlich bessere Suche ist aber in einer sortierten Liste (als verwendete Datenstruktur) *nicht* möglich:

- Eine Liste kann immer nur linear vom ersten (am weitesten links stehenden) Element bis zum letzten Element durchlaufen werden.

- Ursache dafür sind, ist das Konstruktionsprinzip von Listen (mittels `cons`) und die (entsprechenden) zur Verfügung stehenden Selektoren für Listen ($\rightarrow$ `head` und `tail`).

Ein weiteres Problem der Verwendung sortierter Listen ist, dass die Listen im Allgemeinen zunächst sortiert werden müssen:

$\leadsto$ ein nicht unerheblicher zusätzlicher Aufwand

8.6.6. Die Menge als binärer Suchbaum

Prinzip am Beispiel der Suche im Telefonbuch:

- *Annahme:*

  Wir wissen nichts über die Verteilung der Daten, z.B. ob es zu jedem Anfangsbuchstaben überhaupt Einträge gibt bzw. wieviele es gibt.

- *dann ist das folgende Vorgehen am schnellsten:*

  Wir schlagen das Buch (ungefähr) in der Mitte auf.

  – Der gesuchte Name befindet auf der aufgeschlagenen Seite
    $\leadsto$ fertig!
  – Der gesuchte Name befindet in der vorderen Hälfte
    $\leadsto$ weitersuchen nach demselben Prinzip
  – Der gesuchte Name befindet in der hinteren Hälfte
    $\leadsto$ weitersuchen nach demselben Prinzip

  $\longrightarrow$ rekursives Verkleinern des Suchbereichs

Nichtleerer binärer Suchbaum

Eine diesem Suchprinzip entsprechende Datenstruktur ist ein (nichtleerer) Binärbaum, dessen Blätter und Knoten Werte beinhalten:

```
- datatype binbaum3 = Blt of int
                    | Knt of binbaum3 * int * binbaum3;
datatype binbaum3 = Blt of int
                  | Knt of binbaum3 * int * binbaum3
```

Dieser Datentyp schließt den leeren Baum und damit die leere Menge aus.

```
- val b1 = Knt(Blt(8),12,Blt(15));
val b1 = Knt (Blt 8,12,Blt 15) : binbaum3

- val b2 = Knt(Blt(32),45,Blt(50));
val b2 = Knt (Blt 32,45,Blt 50) : binbaum3

- val b3 = Knt(b1,21,b2);
val b3 = Knt (Knt (Blt #,12,Blt #),21,Knt (Blt #,45,Blt #)):binbaum3
```

Nichtleerer binärer Suchbaum (II)

Der Wert von `b3` kann auch wie folgt dargestellt werden:

```
                21
              /     \
           12          45
          /  \        /   \
        8     15   32      50
```

Dieser Baum hat folgende Eigenschaft:

Für jeden Knoten gilt, dass alle Markierungen im linken Teilbaum kleiner sind als die Markierung des Knotens und alle Markierungen im rechten Teilbaum größer.

Binärbäume vom Typ `binbaum3` können nicht alle endlichen Mengen darstellen:

$\{1, 2\}$ und $\{1, 2, 3, 4\}$ sind z.B. nicht als Binärbaum vom Typ `binbaum3` dargestellbar.

Nichtleerer binärer Suchbaum (III)

Mittels struktureller Induktion lässt sich beweisen, dass jeder Binärbaum vom Typ `binbaum3` eine ungerade Anzahl von Knotenmarkierungen hat:

Beweis:

Basisfall: Ein Binärbaum der Gestalt `Blt(W)` hat genau eine Knotenmarkierung, also eine ungerade Anzahl von Knotenmarkierungen.

Induktionsfall: Seien `B1` und `B2` zwei Binärbäume vom Typ `binbaum3` und `W` eine ganze Zahl.

Induktionsannahme: `B1` und `B2` haben jeweils eine ungerade Anzahl von Knotenmarkierungen.

Der Binärbaum `Knt(B1, W, B2)` hat $k = |\texttt{B1}| + 1 + |\texttt{B2}|$ Knotenmarkierungen.
Da $|\texttt{B1}|$ ungerade ist, gibt es $n_1 \in \mathbb{N}$ mit $|\texttt{B1}| = 2 * n_1 + 1$.
Da $|\texttt{B2}|$ ungerade ist, gibt es $n_2 \in \mathbb{N}$ mit $|\texttt{B2}| = 2 * n_2 + 1$.
Also $k = (2*n_1+1)+1+(2*n_2+1) = 2*(n_1+n_2+1)+1$, d.h. k ist ungerade.

qed.

Verbesserung des „nichtleeren binären Suchbaums“

Der folgende Typ `binbaum4` beseitigt diesen Mangel des Typs `binbaum3`:

```
- datatype binbaum4 = Knt1 of int
                    | Knt2 of int * int
                    | Knt3 of binbaum4 * int * binbaum4;
datatype binbaum4
     = Knt1 of int | Knt2 of int * int
             | Knt3 of binbaum4 * int * binbaum4

- val c0 = Knt2(1,2);
val b2 = Knt2 (1,2) : binbaum4

- val c1 = Knt3(Knt1(1), 2, Knt2(3,4));
val c1 = Knt3 (Knt1 1,2,Knt2 (3,4)) : binbaum4

- val c2 = Knt3(Knt2(1,2), 3, Knt1(4));
val c2 = Knt3 (Knt2 (1,2),3,Knt1 4) : binbaum4

- val d = Knt3(Knt2(1,2), 3, Knt3(Knt1(4), 5, Knt1(6)));
val d = Knt3 (Knt2 (1,2),3,Knt3 (Knt1 #,5,Knt1 #)) : binbaum4
```

Verbesserung des „nichtleeren binären Suchbaums“ (II)

Die Binärbäume `c1` und `c2` können wie folgt dargestellt werden:

```
      2                                   3
    /   \                               /   \
   1     4                             2     4
        /                             /
       3                             1
```

`c1` und `c2` sind die zwei Möglichkeiten, die Menge $\{1, 2, 3, 4\}$ als binärer Suchbaum vom Typ `binbaum4` darzustellen.

Der Binärbaum `d` kann wie folgt dargestellt werden:

```
      3
    /   \
   2     5
  /     / \
 1     4   6
```

Verbesserung des „nichtleeren binären Suchbaums“ (III)

Betrachten wir die folgenden binären Suchbäume vom Typ `binbaum4`:

```
- val b1 = Knt3(Knt1(8), 12, Knt1(15));
val b1 = Knt3 (Knt1 8,12,Knt1 15) : binbaum4

- val b2 = Knt3(Knt1(32), 45, Knt1(50));
val b2 = Knt3 (Knt1 32,45,Knt1 50) : binbaum4

- val b3 = Knt3(b1, 21, b2);
val b3 = Knt3 (Knt3 (Knt1 #,12,Knt1 #),21,
             Knt3 (Knt1 #,45,Knt1 #)) : binbaum4
```

```
                 21
               /    \
            12        45
           /  \      /  \
          8    15  32    50
```

Verbesserung des „nichtleeren binären Suchbaums“ (IV)

Die Suche nach 25 (bzw. nach 32) in `b3` kann wie folgt durchgeführt werden:

1. Da $25 > 21$ (bzw. $32 > 21$) wird die Suche im rechten Teilbaum `b2` von `b3` fortgesetzt.
2. Da $25 < 45$ (bzw. $32 < 45$) ist, wird die Suche im linken Teilbaum `Knt1`(32) von `b2` fortgesetzt.
3. Da $25 \neq 32$ ist, terminiert die Suche erfolglos
   (bzw. da $32 = 32$ ist, terminiert die Suche erfolgreich).

Das Verfahren macht nur dann Sinn, wenn

- für jeden (Teil-)Baum der Gestalt `Knt2(Markierung1, Markierung2)` gilt:

  ```
  Markierung1 < Markierung2
  ```

- und für jeden (Teil-)Baum der Gestalt `Knt3(LBaum, Markierung, RBaum)` für jeden Knoten `Kl` in `LBaum` und jeden Knoten `Kr` in `RBaum` gilt:

  ```
  Kl < Markierung < Kr
  ```

Die Suche in binären Suchbäumen vom Typ `binbaum4`

```
- fun suche(x, Knt1(M))                 = (x = M)
    | suche(x, Knt2(M1, M2))            = (x = M1) orelse (x = M2)
    | suche(x, Knt3(LBaum, M, RBaum)) =
                if x = M
                   then true
                   else if x < M
                           then suche(x, LBaum)
                           else suche(x, RBaum);
val suche = fn : int * binbaum4 -> bool

- suche(25, b3);
val it = false : bool

- suche(32, b3);
val it = true : bool

- suche(4, d);
val it = true : bool
```

Zeitbedarf der Suche in einem binären Suchbaum

Der Zeitbedarf der Suche nach einer Zahl in einem binären Suchbaum vom Typ `binbaum4` kann wie folgt geschätzt werden:

- Als Problemgröße bietet sich die Anzahl der Knotenmarkierungen an, d.h. die Kardinalität der Menge, die der binäre Suchbaum darstellt.
- Als Zeiteinheit bietet sich die Anzahl der Knoten an, die besucht werden, bis eine Antwort geliefert wird.

Die Suche wird also die folgenden Zeiten in Anspruch nehmen:

- 1 Zeiteinheit, wenn der Binärbaum die Gestalt `Knt1(M)` hat (d.h. aus einem Blatt besteht),
- 2 Zeiteinheiten, wenn der Binärbaum die Gestalt `Knt2(M1, M2)` hat (d.h. nur eine einzige Kante enthält);
- höchstens die Anzahl der Knoten entlang eines längsten Astes des Baumes.

Ausgeglichenheit

- Die Länge der Äste eines Baumes bestimmt offensichtlich die Suchzeit.
- Daher ist es wünschenswert, dass alle Äste des Baumes gleich lang sind.
- Längenunterschiede von einem Knoten zwischen zwei Ästen lassen sich aber i.Allg. nicht vermeiden.

Definition (Ausgeglichener Baum)

Ein (Binär-)Baum heißt *ausgeglichen*, wenn sich die Längen zweier beliebiger Äste dieses Baumes um höchstens 1 unterscheiden.

Definition (Tiefe)

Ein Knoten `k` eines Baumes hat die *Tiefe* t, wenn der Pfad von der Wurzel nach `k` in dem Baum t Knoten enthält.

Die *Tiefe eines Baumes* ist die maximale Tiefe eines seiner Knoten.

Ausgeglichenheit (II)

Satz A
Die Länge eines Astes eines ausgeglichenen Binärbaums, der n Knoten hat, ist $O(\ln n)$.

Aus dem Satz A folgt, dass die Suche in einem ausgeglichenen Binärbaum die Zeitkomplexität $O(\ln n)$ hat, wenn n die Kardinalität der Menge ist.

Der Satz A folgt aus der folgenden Beobachtung:

Satz B
Ein ausgeglichener Binärbaum der Tiefe t kann bis zu 2^t Knoten mit Tiefe t haben.

Ausgeglichenheit (III)

Beweis des Satzes B (induktiv):

Basisfall: Die Aussage gilt für Binärbäume der Tiefe 0 und 1.

Induktionsfall: Sei $t \in \mathbb{N}$.

Induktionsannahme: Ein ausgeglichener Baum der Tiefe t hat bis zu 2^t Knoten mit Tiefe t.

Ein Binärbaum der Tiefe $t + 1$ besteht aus einer Wurzel mit zwei Nachfolgern, die die Wurzeln von Teilbäumen der Tiefe t sind.

Jeder dieser Teilbäume kann nach Induktionsannahme bis zu 2^t Knoten mit Tiefe t in dem jeweiligen Teilbaum haben.

Diese Knoten sind genau die Knoten mit Tiefe $t+1$ im gesamten Baum, zusammen sind es also bis zu $2 * (2^t) = 2^{t+1}$ Knoten.

qed.

Ausgeglichenheit (IV)

Beweis des Satzes A:

Aus dem Satz B folgt, dass ein ausgeglichener Binärbaum der Tiefe t maximal $2^0 + 2^1 + \ldots + 2^t = 2^{t+1} - 1$ Knoten hat. Hat also ein ausgeglichener Binärbaum n Knoten, so gilt:

$$2^t - 1 \leq n \leq 2^{t+1} - 1$$

wobei t die Tiefe des Baumes ist. Daraus folgt:

$$t \leq \log_2(n+1) \leq t+1$$

Das heißt: $t \in O(\log_2 n + 1) = O(K * \ln n)$ für ein $K \in \mathbb{N}$, d.h. $t \in O(\ln n)$.

qed.

Preis der Ausgeglichenheit

Hinzufügen oder Löschen von Elementen einer Menge kann die Ausgeglichenheit des zugehörigen Binärbaums zerstören.

Beispiel:
Der folgende Baum `B1` ist ausgeglichen.
Nach Einfügen des Wertes 0 ist der entstehende Baum `B2` nicht mehr ausgeglichen:

```
B1:     4           B2:     4
       / \                 / \
      2   5               2   5
     / \                 / \
    1   3               1   3
                       /
                      0
```

Ein Verfahren ist also nötig, um die Ausgeglichenheit nach jeder oder nach einigen Änderungen der Menge wiederherzustellen. Der Algorithmus dazu ist nicht trivial.

8.7. Beispiele: Grundlegende Funktionen für binäre (Such-)Bäume

Gegeben sei der Datentyp eines „Binärbaums mit Knotenmarkierungen”:

```
- datatype 'a binbaum5 = Leer
                       | Knt1 of                'a
                       | Knt2 of 'a           * 'a
                       | Knt3 of 'a binbaum5 * 'a * 'a binbaum5;
```

In Ausdrücken `Knt2(M1,M2)` wird die Markierung `M1` als linker Nachfolger des Knotens mit Markierung `M2` verstanden.

Grundlegende Funktionen für binäre (Such-)Bäume (II)

Betrachten wir die Binärbäume mit folgenden graphischen Darstellungen:

```
B1:
  3
```

```
B2:
    3
   /
  2
```

```
B3:
    3
   / \
  2   5
```

```
B4:
      3
     / \
    2   5
   /
  1
```

```
B5:
    3
   / \
  2   5
     /
    4
```

```
B:
      3
     / \
    2   5
   /   / \
  1   4   6
```

```
A:
        "-"
       /   \
    "*"     "/"
    / \     / \
  "x" "y" "2" "+"
              / \
            "z" "1"
```

Der Baum `A` soll offenbar einen arithmetischen Ausdruck repräsentieren.

Grundlegende Funktionen für binäre (Such-)Bäume (III)

Darstellung als Ausdrücke des Typs `int binbaum5` bzw. `string binbaum5`:

```
- val B1 = Knt1(          3);
- val B2 = Knt2(2,        3);
- val B3 = Knt3(Knt1(2),   3, Knt1(5));
- val B4 = Knt3(Knt2(1,2), 3, Knt1(5));
- val B5 = Knt3(Knt1(2),   3, Knt2(4,5));
- val B  = Knt3(Knt2(1,2), 3, Knt3(Knt1(4), 5, Knt1(6)));

- val A = Knt3( Knt3( Knt1("x"),
                      "*",
                      Knt1("y")  ),
                "-",
                Knt3( Knt1("2"),
                      "/",
                      Knt3( Knt1("z"),
                            "+",
                            Knt1("1")  )  )
              );
```

8.7.1. Selektoren und Prädikate

(auf das hier verwendete Konzept der `exception` gehen wir in Kapitel 10 ein)

```
- exception binbaum5_leer;
- exception binbaum5_kein_nachfolger;

- fun wurzel(Leer)            = raise binbaum5_leer
    | wurzel(Knt1(M))         = M
    | wurzel(Knt2(_, M))      = M
    | wurzel(Knt3(_, M, _))   = M;

- fun linker_baum(Leer)              = raise binbaum5_kein_nachfolger
    | linker_baum(Knt1(_))           = raise binbaum5_kein_nachfolger
    | linker_baum(Knt2(_, _))        = raise binbaum5_kein_nachfolger
    | linker_baum(Knt3(B, _, _))     = B;

- fun rechter_baum(Leer)             = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt1(_))          = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt2(_, _))       = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt3(_, _, B))    = B;
```

Selektoren und Prädikate (II)

```
- fun ist_leer(Leer) = true
    | ist_leer(_)    = false;
```

Damit sind Zugriffe auf Bestandteile von Binärbäumen möglich:

```
- wurzel(B);
val it = 3 : int

- wurzel(rechter_baum(B));
val it = 5 : int

- wurzel(linker_baum(A));
val it = "*" : string
```

Die folgenden Definitionen sind alle mit Hilfe des Pattern Matching aufgebaut, so dass die obigen Funktionen darin nicht benötigt werden.

8.7.2. Durchlauf in Infix-Reihenfolge

Die Infix-Reihenfolge bedeutet, dass zunächst die Knoten des linken Teilbaums, dann die Wurzel und anschließend die Knoten des rechten Teilbaums aufgesammelt werden.

```
- fun infix_collect(Leer)              = nil
    | infix_collect(Knt1(M))           = [M]
    | infix_collect(Knt2(M1,M2))       = [M1,M2]
    | infix_collect(Knt3(B1,M,B2)) =
                 infix_collect(B1) @ [M] @ infix_collect(B2);
val infix_collect = fn : 'a binbaum5 -> 'a list

- infix_collect(B);
val it = [1,2,3,4,5,6] : int list

- infix_collect(A);
val it = ["x","*","y","-","2","/","z","+","1"] : string list
```

Durchlauf in Infix-Reihenfolge (II)

Die Bezeichnung „Infix-Reihenfolge" wird am Beispiel des Baums `A` leicht verständlich:

- Die Ergebnisliste entspricht dem durch den Baum `A` repräsentierten arithmetischen Ausdruck in Infix-Notation.
- Allerdings ist die Information über die Klammerung verloren gegangen.

Die rechte Seite des letzten Falls der Definition könnte auch so umformuliert werden:

```
infix_collect(B1) @ M :: infix_collect(B2)
```

Wir haben mit `infix_collect(B1) @ [M] @ infix_collect(B2)` eine umständlichere Form gewählt, die aber die Analogien und Unterschiede zu den folgenden Funktionen deutlicher macht.

8.7.3. Durchlauf in Präfix-Reihenfolge

Die Präfix-Reihenfolge bedeutet, dass die Wurzel vor den Knoten des linken Teilbaums, und diese vor den Knoten des rechten Teilbaums aufgesammelt werden.

```
- fun praefix_collect(Leer)              = nil
    | praefix_collect(Knt1(M))           = [M]
    | praefix_collect(Knt2(M1,M2))       = [M2,M1]
    | praefix_collect(Knt3(B1,M,B2)) =
              [M] @ praefix_collect(B1) @ praefix_collect(B2);
val praefix_collect = fn : 'a binbaum5 -> 'a list

- praefix_collect(B);
val it = [3,2,1,5,4,6] : int list

- praefix_collect(A);
val it = ["-","*","x","y","/","2","+","z","1"] : string list
```

Durchlauf in Präfix-Reihenfolge (II)

- Die Kenntnis der Stelligkeit der Operationen reicht aus, um aus der „Linearisierung“ in Präfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.

- Auch hier wäre es angebracht, `[M] @ praefix_collect(B1)` zu vereinfachen zu `M :: praefix_collect(B1)`.

8.7.4. Durchlauf in Postfix-Reihenfolge

In Postfix-Reihenfolge werden zunächst die Knoten des linken Teilbaums, dann des rechten Teilbaums, und schließlich die Wurzel aufgesammelt.

```
- fun postfix_collect(Leer)          = nil
    | postfix_collect(Knt1(M))       = [M]
    | postfix_collect(Knt2(M1,M2))   = [M1,M2]
    | postfix_collect(Knt3(B1,M,B2)) =
               postfix_collect(B1) @ postfix_collect(B2) @ [M];
val postfix_collect = fn : 'a binbaum5 -> 'a list

- postfix_collect(B);
val it = [1,2,4,6,5,3] : int list

- postfix_collect(A);
val it = ["x","y","*","2","z","1","+","/","-"] : string list
```

Die Kenntnis der Stelligkeit der Operationen reicht aus, aus der „Linearisierung“ in Postfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.

8.7.5. Infix-/Präfix-/Postfix-Reihenfolge mit Akkumulatortechnik

Durch den Aufruf von `append` in jedem rekursiven Aufruf summiert sich die Gesamtanzahl der Aufrufe von „`::`“ auf einen unnötig hohen Wert.

Die Funktionen sammeln zwar die Markierungen in der gewünschten Reihenfolge auf, aber verschachtelt in Listen, die dann erst wieder von `append` (bzw. „`@`“) zu einer „flachen“ Liste zusammengefügt werden müssen.

In ähnlicher Weise, wie mit Hilfe der Akkumulator-Technik aus der Funktion `naive-reverse` die Funktion `reverse` entwickelt wurde (vgl. Abschnitt 5.5.8), kann man auch hier mit Hilfe der Akkumulator-Technik bessere Definitionen entwickeln.

Infix-/Präfix-/Postfix-Reihenfolge mit Akkumulatortechnik (II)

am Beispiel des Durchlaufs in Infix-Reihenfolge:

```
- fun infix_collect(B) =
    let
     fun collect(Leer,          L) = L
       | collect(Knt1(M),       L) = M :: L
       | collect(Knt2(M1,M2),   L) = M1 :: M2 :: L
       | collect(Knt3(B1,M,B2),L) = collect(B1, M :: collect(B2,L))
    in
        collect(B, nil)
    end;
val infix_collect = fn : 'a binbaum5 -> 'a list
```

Die Reihenfolge, in der die Parameter `B1`, `M`, `B2` in den rekursiven Aufrufen weitergereicht werden, ist jeweils die gleiche wie in den vorhergehenden Definitionen.

Die lokale Hilfsfunktion `collect` ist trotz der Verwendung der Akkumulator-Technik *nicht* endrekursiv!

8.7.6. Tiefendurchlauf (Depth-First-Durchlauf)

- Infix-, Präfix- und Postfix-Reihenfolge haben gemeinsam, dass die Teilbäume des Baums unabhängig voneinander durchlaufen werden.
- Der Unterschied der Funktionen liegt nur darin, wie das Ergebnis jeweils zusammengesetzt wird.
- Das gemeinsame Prinzip ist als *Tiefendurchlauf* bekannt und kann durch Funktionen höherer Ordnung als gemeinsame Abstraktion definiert werden:

```
fun depth_first_collect f0 f1 f2 f3  Leer            = f0
  | depth_first_collect f0 f1 f2 f3 (Knt1(M))        = f1(M)
  | depth_first_collect f0 f1 f2 f3 (Knt2(M1,M2))    = f2(M1,M2)
  | depth_first_collect f0 f1 f2 f3 (Knt3(B1,M,B2)) =
                         f3(depth_first_collect f0 f1 f2 f3 B1,
                         M,
                         depth_first_collect f0 f1 f2 f3 B2);
```

Tiefendurchlauf (II)

```
fun infix_collect(B) =
            depth_first_collect nil
                                (fn M          => [M])
                                (fn (M1,M2)    => [M1,M2])
                                (fn (R1,M,R2) => R1 @ [M] @ R2)
                                B;
fun praefix_collect(B) =
            depth_first_collect nil
                                (fn M          => [M])
                                (fn (M1,M2)    => [M2,M1])
                                (fn (R1,M,R2) => [M] @ R1 @ R2)
                                B;
fun postfix_collect(B) =
            depth_first_collect nil
                                (fn M          => [M])
                                (fn (M1,M2)    => [M1,M2])
                                (fn (R1,M,R2) => R1 @ R2 @ [M])
                                B;
```

Tiefendurchlauf (III)

Aber auch andere nützliche Funktionen auf Binärbäumen lassen sich mit Hilfe des Tiefendurchlaufs leicht implementieren:

```
fun anzahl_knoten(B) =
          depth_first_collect 0
                              (fn M          => 1)
                              (fn (M1,M2)    => 2)
                              (fn (R1,M,R2) => R1 + 1 + R2)
                              B;

fun anzahl_blaetter(B) =
          depth_first_collect 0
                              (fn M          => 1)
                              (fn (M1,M2)    => 1)
                              (fn (R1,M,R2) => R1 + R2)
                              B;
```

Tiefendurchlauf (IV)

```
fun tiefe(B) =
        depth_first_collect 0
                            (fn M          => 1)
                            (fn (M1,M2)    => 2)
                            (fn (R1,M,R2) => 1 + Int.max(R1,R2))
                            B;

fun element(x,B) =
    depth_first_collect false
                        (fn M          => x=M)
                        (fn (M1,M2)    => x=M2 orelse x=M1)
                        (fn (R1,M,R2) => x=M orelse R1 orelse R2)
                        B;
```

8.7.7. Breitendurchlauf (Breadth-First-Durchlauf)

Bei einem Breitendurchlauf werden die Knoten nach wachsender Tiefe besucht.

Zuerst wird die Wurzel aufgesammelt, dann die Wurzeln der Nachfolger, dann die Wurzeln von deren Nachfolgern usw.

Das Ergebnis des Breitendurchlaufs ist also für Baum `B` die Liste `[3,2,5,1,4,6]` und für Baum `A` die Liste `["-","*","/","x","y","2","+","z","1"]`.

Zur Implementierung bedienen wir uns einer Hilfsfunktion `entwurzeln`, die angewandt auf eine Liste von Bäumen von jedem Baum die Wurzel aufsammelt und die Nachfolger der Wurzel am Ende der Liste für die spätere Weiterverarbeitung einfügt.

Breitendurchlauf (II)

So soll aus der einelementigen Liste von Binärbäumen

```
[ Knt3( Knt3( Knt1("a"),
              "*",
              Knt1("b") ),
        "+",
        Knt3( Knt1("e"),
              "-",
              Knt1("f") )  )   ]
```

nach dem Aufsammeln von „+“ die folgende zweielementige Liste entstehen:

```
[ Knt3( Knt1("a"),
        "*",
        Knt1("b") ),
  Knt3( Knt1("e"),
        "-",
        Knt1("f") )   ]
```

Breitendurchlauf (III)

```
- fun breadth_first_collect(B) =
   let
    fun entwurzeln nil                     = nil
      | entwurzeln(Leer::L)                = entwurzeln(L)
      | entwurzeln(Knt1(M)::L)             = M ::entwurzeln(L)
      | entwurzeln(Knt2(M1,M2)::L)         = M2::entwurzeln(L @ [Knt1(M1)])
      | entwurzeln(Knt3(B1,M,B2)::L)= M ::entwurzeln(L @ [B1,B2])
   in
       entwurzeln(B :: nil)
   end;
val breadth_first_collect = fn : 'a binbaum5 -> 'a list

-  breadth_first_collect(B);
val it = [3,2,5,1,4,6] : int list

- breadth_first_collect(A);
val it = ["-","*","/","x","y","2","+","z","1"] : string list
```

Breitendurchlauf (IV)

Die Funktion `entwurzeln` zerlegt also einen Baum von der Wurzel her, gerade entgegengesetzt zum Aufbau des Baums gemäß der mathematischen Definition oder mit den Konstruktoren des induktiv definierten Typs `'a binbaum5`.

Der Unterschied zu den Selektoren ist, dass die Teilbäume nicht einfach als Ergebnisse geliefert werden, sondern in einer Liste nach dem Prinzip *first-in-first-out* verwaltet werden.

Beim Breitendurchlauf werden die Teilbäume nicht unabhängig voneinander durchlaufen, sondern nach Tiefe verzahnt. Daher kann der Breitendurchlauf nicht mit Hilfe des Tiefendurchlaufs definiert werden kann.

9. Pattern Matching

- 9.1 Zweck des Pattern Matching
- 9.2 Prinzip des Pattern Matching
- 9.3 Pattern Matching und statische Typprüfung — Angleichfehler zur Laufzeit
- 9.4 Das Wildcard–Muster von SML
- 9.5 Das Verbund-Wildcard–Muster von SML
- 9.6 Die gestuften Muster von SML
- 9.7 Linearitätsbedingung für Muster
- 9.8 Der Pattern-Matching-Algorithmus

9.1. Zweck des Pattern Matching

- Muster in Wertdeklarationen
- Muster zur Fallunterscheidung in Funktionsdeklarationen
- Muster zur Fallunterscheidung in case–Ausdrücken

9.1.1. Muster in Wertdeklarationen

Pattern Matching dient zur Selektion der Komponenten eines zusammengesetzten Wertes und kann in Wertdeklarationen wie folgt verwendet werden:

```
- val paar = (1.52, 4.856);
val paar = (1.52,4.856) : real * real

- val (komp1, komp2) = paar;
val komp1 = 1.52 : real
val komp2 = 4.856 : real
```

- Der Ausdruck `(komp1, komp2)` stellt ein „Muster“ (*Pattern*) dar.
- Das Muster `(komp1, komp2)` wird mit dem Wert des Namens `paar`, d.h. mit dem Vektor `(1.52, 4.856)`, verglichen.

  Dieser Vergleich liefert Bindungen für die Namen `komp1` und `komp2`.

  $\leadsto$ *Matching*

Muster in Wertdeklarationen (II)

Ist der Angleich unmöglich, so scheitert die Wertdeklaration, wie etwa in:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y * 'X
  expression:    real * real
  in declaration:
    (k1,k2,k3) =
      (case (1.0,2.0)
        of (k1,k2,k3) => (k1,k2,k3))
```

9.1.2. Muster zur Fallunterscheidung in Funktionsdefinitionen

Muster und die Technik des Pattern Matching werden häufig in Fallunterscheidungen verwendet wie etwa in der folgenden polymorphen Funktionsdefinition:

```
- fun laenge nil      = 0
    | laenge(h :: L) = 1 + laenge(L);
val laenge = fn : 'a list -> int

- laenge [0,1,2,3];
val it = 4 : int
```

Die Muster `nil` und `(h :: L)` decken alle Listenarten ab:

- Das Muster `nil` deckt den Fall der leeren Liste ab;
- das Muster `(h :: L)` deckt den Fall aller nichtleeren Listen ab.

Muster zur Fallunterscheidung in Funktionsdefinitionen (II)

Die statische Typprüfung von SML erkennt, ob die verschiedenen Fälle einer Funktionsdeklaration etwaige Fälle nicht berücksichtigen, wie z.B. in:

```
- fun letztes_element(x :: nil) = x
    | letztes_element(h :: L)   = letztes_element(L);
Warning: match nonexhaustive
          x :: nil => ...
          h :: L => ...
val letztes_element = fn : 'a list -> 'a
```

- Der Fall „leere Liste“ fehlt hier, weil eine leere Liste kein letztes Element hat.
- Die Typprüfung (des SML-Systems) kennt die Semantik der Funktion aber nicht. Sie erkennt lediglich, dass der Fall „leere Liste“ nicht abdeckt wird.
- Die Warnung ist aber ein oft nützlicher Hinweis für den Programmierer.

Muster zur Fallunterscheidung in Funktionsdefinitionen (III)

Muster und Pattern Matching werden verwendet, wenn Funktionen über zusammengesetzten Typen, u.a. über rekursiven Typen, definiert werden.

Die polymorphen Selektoren `head` und `tail` für Listen sind Beispiele dafür:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

- fun tail(_ :: L) = L;
Warning: match nonexhaustive
          _ :: L => ...
val tail = fn : 'a list -> 'a list
```

Muster zur Fallunterscheidung in Funktionsdefinitionen (IV)

Dies ist natürlich auch bei selbstdefinierten rekursiven Typen möglich:

```
- datatype binbaum1 = Leer
                    | Blt of int                  (* f"ur Blatt  *)
                    | Knt of binbaum1 * binbaum1; (* f"ur Knoten *);
datatype binbaum1 = Blt of int
                  | Knt of binbaum1 * binbaum1
                  | Leer;

- val b = Knt(Knt(Blt(1), Blt(3)), Knt(Blt(5), Knt(Blt(8),Blt(9))));
val b = Knt (Knt (Blt #,Blt #),Knt (Blt #,Knt #)) : binbaum1
```

[eine graphische Repräsentation dieses Baumes findet sich in Kapitel 8]

Muster zur Fallunterscheidung in Funktionsdefinitionen (V)

```
- fun blaetterzahl Leer             = 0
    | blaetterzahl (Blt(_))         = 1
    | blaetterzahl (Knt(b1, b2)) =   blaetterzahl b1
                                   + blaetterzahl b2;
val blaetterzahl = fn : binbaum1 -> int

- blaetterzahl(b);
val it = 5 : int
```

Die Fallunterscheidung in der Funktionsdefinition enthält ein Muster für jeden der drei Fälle aus der Definition des Typs `binbaum1`.

Muster zur Fallunterscheidung in Funktionsdefinitionen (VI)

Muster und die Technik des Pattern Matching bieten sich an zur Definition von Funktionen über Typen mit endlichen Wertemengen.

Ein Beispiel ist die folgende Implementierung einer logischen Disjunktion `OR`:

```
- fun OR(true,  true)  = true
    | OR(true,  false) = true
    | OR(false, true)  = true
    | OR(false, false) = false;
val OR = fn : bool * bool -> bool

- OR(OR(false, true), true);
val it = true : bool
```

Anmerkung:

Die Auswertung dieser selbstdefinierten Funktion unterscheidet sich von der für `orelse`–Sonderausdrücke. Dies liegt aber nicht an der Verwendung von Mustern.

9.1.3. Muster zur Fallunterscheidung in `case`–Ausdrücken

Muster und Pattern Matching können auch in herkömmlichen Ausdrücken verwendet werden.

Die Funktion `OR` kann z.B. wie folgt neu definiert werden:

```
- fun OR'(x, y) = case x  of true  => true
                           | false => y;
val OR' = fn : bool * bool -> bool

- OR'(OR'(false, true), true);
val it = true : bool
```

[*Achtung:* auch hier werden in SML stets beide Argumente ausgewertet!]

Muster zur Fallunterscheidung in case–Ausdrücken (II)

In vielen SML-Implementierungen ist die Behandlung von Mustern in case–Ausdrücken Basis für alle anderen Verwendungen von Mustern.

Dies erkennt man z.B. an Fehlermeldungen wie:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y * 'X
  expression:    real * real
  in declaration:
    (k1,k2,k3) =
      (case (1.0,2.0)
        of (k1,k2,k3) => (k1,k2,k3))
```

Die Deklaration wurde offenbar intern in einen case–Ausdruck übersetzt.

9.2. Prinzip des Pattern Matching

- Angleichregel
- Prüfung einer Angleichregel gegen einen Wert
- Prüfung eines Angleichmodells gegen einen Wert
- Typkorrektheit eines Angleichmodells
- herkömmliche Angleichmodelle in SML

9.2.1. Angleichregel

Eine sogenannte *Angleichregel* ist ein Ausdruck der folgenden Gestalt:

```
<Muster> => <Ausdruck>
```

In einer `fun`–Deklarationen sind Angleichregeln „versteckt“, die im äquivalenten `val`–Ausdruck ersichtlich werden. So steht z.B. die `fun`–Deklaration der Funktion `OR`

```
fun OR(true,  true)  = true
  | OR(true,  false) = true
  | OR(false, true)  = true
  | OR(false, false) = false;
```

für die folgende `val`–Deklaration, in der Angleichregeln vorkommen:

```
val rec OR = fn (true,  true)  => true
              | (true,  false) => true
              | (false, true)  => true
              | (false, false) => false;
```

9.2.2. Prüfung einer Angleichregel gegen einen Wert

Eine Angleichregel

```
<Muster> => <Ausdruck>
```

wird wie folgt gegen einen Wert `W` geprüft:

- `W` wird mit dem Muster `<Muster>` verglichen.
- Gelingt der „Angleich“, so werden etwaige Namen, die im Muster vorkommen, gebunden.
- Beim Angleich werden die Strukturen von Muster und Wert gleichzeitig durchlaufen (bzw. zerlegt), indem die Teilausdrücke komponentenweise und rekursiv angeglichen werden.

Prüfung einer Angleichregel gegen einen Wert (II)

Die Basisfälle dieses rekursiven Algorithmus liefern Namen und Konstanten, die im Muster vorkommen.

- Ein in einem Muster vorkommender Name kann bei Typübereinstimmung mit jedem Wert angeglichen werden.
- Wird ein Name, der im Muster vorkommt, an einen Wert angeglichen, so wird der Wert an diesen Namen gebunden.
- Eine in einem Muster vorkommende Konstante kann nur mit derselben Konstante angeglichen werden.

Wie die vorangehende informelle Beschreibung des Pattern Matching suggeriert, stellen Konstanten und Namen hinsichtlich der Gestalt die einfachsten Muster dar.

Wie der Angleich genau durchgeführt wird, sehen wir in Abschnitt 9.8.

9.2.3. Prüfung eines Angleichmodells gegen einen Wert

Angleichregeln werden wie folgt zu einem *Angleichmodell* zusammengesetzt:

```
<Angleichregel> | ...| <Angleichregel>
```

- Die Angleichregeln werden sequenziell durchlaufen, wenn ein Wert `W` gegen sie geprüft wird.
- Ist der Angleich von `W` mit dem Muster einer Angleichregel des Angleichmodells erfolgreich, so wird `W` gegen die nachfolgenden Angleichregeln nicht mehr geprüft.

9.2.4. Typkorrektheit eines Angleichmodells

Ein Angleichmodells der Gestalt

$$\texttt{Muster}_1 \texttt{ => Ausdruck}_1 \texttt{ | ...| Muster}_n \texttt{ => Ausdruck}_n$$

ist nur dann korrekt typisiert, wenn:

1. $\texttt{Muster}_1, \ldots, \texttt{Muster}_n$ alle denselben Typ haben, und
2. $\texttt{Ausdruck}_1, \ldots, \texttt{Ausdruck}_n$ alle denselben Typ haben.

Die Muster können aber einen anderen Typ haben als die Ausdrücke.

Diese Bedingungen können während der statischen Typprüfung überprüft werden.

9.2.5. Herkömmliche Angleichmodelle in SML

Aus den bisher beschriebenen Prinzipien ergeben sich die beiden herkömmlichen Formen von Angleichmodellen in SML:

```
case <Ausdruck> of <Angleichmodell>
```

und

```
fn <Angleichmodell>
```

Anmerkung:
Die `fun`–Deklarationen mit Pattern Matching stellen lediglich eine „syntaktische Verzuckerung“ des vorangehenden Angleichmodells mit `fn`–Ausdrücken dar.

9.3. Pattern Matching und statische Typprüfung — Angleichfehler zur Laufzeit

- Pattern Matching und statische Typprüfung
- Angleichfehler zur Laufzeit

9.3.1. Pattern Matching und statische Typprüfung

Schon während der statischen Typprüfung kann ein Angleichmodell auf Typkorrektheit überprüft werden, wie etwa in:

```
- fun f (0, true) = true
    | f (1, 2)    = false;
Error: parameter or result constraints of clauses don’t agree [literal]
  this clause:      int * int -> ’Z
  previous clauses:      int * bool -> ’Z
  in declaration:
    f =
      (fn (0,true) => true
        | (1,2) => false)
```

Der Typfehler besteht darin, dass die beiden Muster `(0, true)` und `(1, 2)` des Angleichmodells nicht denselben Typ haben.

Pattern Matching und statische Typprüfung (II)

Auch unterschiedliche Typen bei Ausdrücken in einem Angleichmodell werden statisch erkannt:

```
- fun f true  = 1
    | f false = 0.0;
Error: right-hand-side of clause doesn't agree
with function result type [literal]
  expression:  real
  result type:  int
  in declaration:
    f =
      (fn true => 1
        | false => 0.0)
```

9.3.2. Angleichfehler zur Laufzeit

Da Angleichmodelle nicht alle Fälle abdecken müssen, können Angleichfehler vorkommen, die erst zur Laufzeit erkannt werden können:

```
- fun f(0) = 0
    | f(x) = if x > 0 then f(x-1) else f(~x-1);
val f = fn : int -> int

- fun g(1) = true;
Warning: match nonexhaustive
          1 => ...
val g = fn : int -> bool

- g(f(1));
uncaught exception nonexhaustive match failure
```

Um festzustellen, dass kein Muster der Deklaration von Fall `g` den Fall `g(f(1))` abdeckt, muss `f(1)` ausgewertet werden. Dies kann nicht statisch geprüft werden!

9.4. Das Wildcard–Muster von SML

Das sogenannte *Wildcard–Muster* von SML „_" hat zwei Eigenschaften:

- Zum einen kann es an jeden Wert angeglichen werden.
- Zum anderen wird es bei einem erfolgreichen Angleich an keinen Wert gebunden.

Typischerweise wird „_" als Fangfall verwendet werden, z.B. in:

```
val Ziffer = fn 0 => true
              | 1 => true
              | 2 => true
              | 3 => true
              | 4 => true
              | 5 => true
              | 6 => true
              | 7 => true
              | 8 => true
              | 9 => true
              | _ => false;
```

Das Wildcard–Muster von SML (II)

Das Wildcard–Muster darf auch in einem zusammengesetzten Ausdruck vorkommen, um an Teilausdrücke angeglichen zu werden, deren konkrete Werte nicht weiter von Belang sind:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

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

Der Rest der Liste spielt hier keine Rolle. Daher kann (und sollte!) das Wildcard–Muster anstatt eines Namens verwenden werden.

9.5. Das Verbund-Wildcard–Muster von SML

Verbunde sind zusammengesetzte Strukuren, deren Komponenten Namen tragen, wobei die Reihenfolge der Komponenten keine Rolle spielt:

```
val adressbucheintrag = {Nachname           = "Bry",
                          Vornamenbuchstabe = #"F",
                          Durchwahl         = "2210"};
```

Die vollständige Angabe eines Verbundes verlangt oft viel Schreibarbeit. Das sogenannte *Verbund-Wildcard–Muster* stellt hier eine erhebliche Hilfe dar:

```
- val {Nachname = NN, ...} = adressbucheintrag;
val NN = "Bry" : string
```

So kann der Name `NN` an den Wert `"Bry"` (vom Typ Zeichenfolge) gebunden werden, ohne dass im Muster die Struktur des Verbundes vollständig angeben werden muss.

Das Verbund-Wildcard–Muster von SML (II)

```
- val v1 = {a = 1, b = 2};
val v1 = {a=1,b=2} : {a:int, b:int}

- val v2 = {a = 0, b = 2, c = 3};
val v2 = {a=1,b=2,c=3} : {a:int, b:int, c:int}

- val {a = N, ...} = v1;
val N = 1 : int

- val {a = N, ...} = v2;
val N = 0 : int
```

In beiden Wertdeklarationen, die ein Verbund-Wildcard–Muster enthalten, ist eindeutig, an welchen Wert der Name `N` gebunden werden soll.

Das Verbund-Wildcard–Muster von SML (III)

Eine Verwendung des Verbund-Wildcard–Musters setzt aber voraus, dass der Typ des Verbundes im Muster statisch, also zur Übersetzungszeit, bestimmt werden kann.

Da das bei der folgenden Funktionsdeklaration nicht möglich ist, wird ein Fehler gemeldet:

```
- fun f({a = 1, ...}) = true
    | f(_)            = false;
  Error: unresolved flex record
(can't tell what fields there are besides #a)
```

9.6. Die gestuften Muster von SML

SML bietet zudem die sogenannten „gestuften Muster“ (*layered pattern*) an:

```
- val info1dozent = {Dozent = ("Bry", #"F"), Raum = "D1.04"};
val info1dozent = {Dozent=("Bry",#"F"),Raum="D1.04"}
  : {Dozent:string * char, Raum:string}

- val {Dozent = D as (N,V), Raum = R} = info1dozent;
val D = ("Bry",#"F") : string * char
val N = "Bry" : string
val V = #"F" : char
val R = "D1.04" : string
```

Mit dem Konstrukt

```
D as (N,V)
```

wird nicht nur der Wert `("Bry", #"F")` an den Namen `D` sondern auch die Werte `"Bry"` und `#"F"` an die Namen `N` bzw. `V` gebunden.

9.7. Linearitätsbedingung für Muster

Muster müssen *linear* sein; in dem Sinne, dass eine ungebundene Variable, die durch den Angleich des Musters an einen Wert gebunden werden soll, nur einmal — daher die Bezeichnung linear — im Muster vorkommt.

Die folgende polymorphe Funktion

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

kann also nicht wie folgt deklariert werden:

```
- val gleich = fn (x, x) => true
                | _      => false;
Error: duplicate variable in pattern(s): x
```

Diese Einschränkung erlaubt es, den Pattern-Matching–Algorithmus einfach und effizient zu halten. Ohne diese Einschränkung müssten bei Mehrfachvorkommen von Variablen in Mustern die entsprechenden Werte zu Gleichheitstypen gehören.

9.8. Der Pattern-Matching-Algorithmus

- Informelle Spezifikation des Pattern-Matching–Algorithmus
- Umgebung (Wiederholung aus Kapitel 2)
- Formale Spezifikation des Pattern-Matching–Algorithmus
- Beispiel einer Anwendung des Pattern-Matching–Algorithmus
- Korrektheit und Terminierung des Pattern-Matching–Algorithmus
- Pattern Matching und Unifikation
- Folgen der Linearätsbedingung für den Pattern-Matching–Algorithmus

9.8.1. Informelle Spezifikation des Pattern-Matching–Algorithmus

In Abschnitt 9.2.2 hatten wir den Pattern-Matching–Algorithmus informell eingeführt als „Angleich“ zwischen einen Wert `W` und einem Muster `M`:

- `W` wird mit dem Muster `M` verglichen.
- Gelingt der „Angleich“, so werden etwaige Namen, die im Muster vorkommen, gebunden.
- Beim Angleich werden die Strukturen von Muster und Wert gleichzeitig durchlaufen (bzw. zerlegt), indem die Teilausdrücke komponentenweise und rekursiv angeglichen werden.
- Basisfälle dieses rekursiven Algorithmus sind Namen und Konstanten, die im Muster vorkommen.

Informelle Spezifikation des Pattern-Matching–Algorithmus (II)

Pattern Matching erfolgt zwischen einem Muster `M` und einem Wert `W` — in einem Programm sind aber jeweils ein Muster `M` und ein Ausdruck `A` gegeben, z.B. in der Form

```
val M = A  oder  case A of M =>
```

- Bei der Auswertung wird zunächst `A` in der aktuellen Umgebung ausgewertet zu einem Wert `W`.
- Danach wird der Pattern-Matching-Algorithmus auf das Muster `M` und den Wert `W` angewandt.

9.8.2. Umgebung (Wiederholung aus Kapitel 2)

Die Umgebung zur Verwaltung von Bindungen von Werten an Namen hatten wir wie folgt eingeführt:

- Die Umgebung ist eine Liste von Gleichungen der Gestalt `Name = Wert`.
- Das SML-System verwaltet mit jeder Sitzung und jeder eingelesenen Datei (d.h. Programm) eine Umgebung.
- Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N = W` am Anfang der Umgebung.
- Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen.

  So gilt immer als Wert eines Namens `N` derjenige Wert, der bei der letzten Deklaration von `N` angegeben wurde.

Der Pattern-Matching–Algorithmus muss sein Ergebnis in einer Form liefern, die für die Erweiterung der aktuellen Umgebung um neue Einträge geeignet ist.

9.8.3. Formale Spezifikation des Pattern-Matching–Algorithmus

Der Einfachheit halber werden im folgenden Algorithmus die gestuften Muster und das Verbund-Wildcard–Muster nicht berücksichtigt.

Der Pattern-Matching-Algorithmus wird auf ein Muster `M` und einen Wert `W` angewandt.

- Der Pattern-Matching–Algorithmus soll, wenn möglich, `M` und `W` angleichen und dabei Werte für die Namen, die im Muster `M` vorkommen, ermitteln.
- Die Bindung dieser Werte an diese Namen erfolgt aber nicht während des Pattern Matching, sondern erst danach.

  Das heißt, der Pattern-Matching–Algorithmus verändert die aktuelle Umgebung nicht, er liefert nur die Bindungen, um die die Umgebung anschließend erweitert werden kann.

Der Pattern-Matching–Algorithmus soll auch feststellen, dass der Angleich von Muster `M` und Wert `W` unmöglich ist, d.h. *gescheitert* ist.

Formale Spezifikation des Pattern-Matching–Algorithmus (II)

Der Pattern-Matching-Algorithmus meldet also

- entweder einen Erfolg und liefert eine (endliche) Menge von Bindungen für die (endlich vielen) Namen, die im Muster `M` vorkommen (dabei bleibt die Umgebung unverändert); Jede dieser Bindungen wird dargestellt als ein Paar `(Name, Wert)`;
- oder ein Scheitern.

Zur Angleichung eines Musters `M` und eines Werts `W` gehe wie folgt vor:

1. Falls `M` eine Konstante `k` ist, dann:
   (a) Falls `W` ebenfalls die Konstante `k` ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.
   (b) Andernfalls terminiere erfolglos.
2. Falls `M` ein Name ist, dann: liefere die einelementige Menge $\{$`(M,W)`$\}$ von Bindungen und terminiere erfolgreich.

Formale Spezifikation des Pattern-Matching–Algorithmus (III)

3. Falls `M` das Wildcard–Muster _ ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.

4. Falls `M` zusammengesetzt ist mit (Wert-)Konstruktor `K` und Teilmustern $\mathtt{M}_1, \ldots, \mathtt{M}_n$, dann:

   (a) Falls `W` ebenfalls zusammengesetzt ist mit demselben Konstruktor `K` und Teilwerten $\mathtt{W}_1, \ldots, \mathtt{W}_n$, dann: wende für jedes $\mathtt{i} \in \{1, \ldots, \mathtt{n}\}$ den Pattern-Matching–Algorithmus auf das Muster $\mathtt{M}_\mathtt{i}$ und den Wert $\mathtt{W}_\mathtt{i}$ an.

      i. Falls eine dieser Anwendungen des Pattern-Matching–Algorithmus scheitert, dann terminiere erfolglos.

      ii. Andernfalls bilde die Vereinigung aller Bindungsmengen, die diese Anwendungen des Pattern-Matching–Algorithmus liefern; liefere die so erhaltene Menge von Bindungen und terminiere erfolgreich.

   (b) Andernfalls terminiere erfolglos.

9.8.4. Beispiel einer Anwendung des Pattern-Matching–Algorithmus

`M` sei das Muster `e1 :: (e2 :: _)`,
`W` sei der Wert `1 :: (2 :: (3 :: (4 :: (5 :: nil))))`:

Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:

`K` ist der Konstruktor `::`, `M1` ist das Muster `e1`, `M2` ist das Muster `e2 :: _`

Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:

`W1` ist der Wert `1`, `W2` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`

- Anwendung des Pattern-Matching-Algorithmus auf `M1` und `W1`:
  Nebenrechnung, in der gilt:
  `M` ist das Muster `e1`, `W` ist der Wert `1`
  Fall 2: `M` ist ein Name:
  liefere die Menge {`(e1,1)`} und terminiere erfolgreich
  Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e1,1)`}

. . . Anwendung des Pattern-Matching–Algorithmus (II)

- Anwendung des Pattern-Matching-Algorithmus auf `M2` und `W2`:
  Nebenrechnung, in der gilt:
  `M` ist das Muster `e2 :: _`, `W` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`
  Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:
  `K` ist der Konstruktor `::`, `M1` ist das Muster e2, `M2` ist das Muster `_`
  Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:
  `W1` ist der Wert 2, `W2` ist der Wert `3 :: (4 :: (5 :: nil))`
  - Anwendung des Algorithmus auf `M1` und `W1`:
    Nebenrechnung, in der gilt:
    `M` ist das Muster e2, `W` ist der Wert 2
    Fall 2: `M` ist ein Name:
    liefere die Menge {`(e2,2)`} und terminiere erfolgreich
    Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e2,2)`}

. . . Anwendung des Pattern-Matching–Algorithmus (III)

– Anwendung des Algorithmus auf `M2` und `W2`:

Nebenrechnung, in der gilt:

`M` ist das Muster _, `W` ist der Wert `3 :: (4 :: (5 :: nil))`

Fall 3: `M` ist das Wildcard–Muster _:

liefere die leere Menge und terminiere erfolgreich.

Ende der Nebenrechnung mit Erfolg, Ergebnis {}

Fall 4(a)ii: Beide Anwendungen waren erfolgreich:

Bilde die Vereinigung von {`(e2,2)`} und {};

liefere {`(e2,2)`} und terminiere erfolgreich

Ende der Nebenrechnung mit Erfolg, Ergebnis {`(e2,2)`}

Fall 4(a)ii: Beide Anwendungen waren erfolgreich:

Bilde die Vereinigung von {`(e1,1)`} und {`(e2,2)`};

liefere {`(e1,1), (e2,2)`} und terminiere erfolgreich.

Ende der gesamten Berechnung, Erfolg, Ergebnis (e1,1), (e2,2).

9.8.5. Korrektheit und Terminierung des Pattern-Matching–Algorithmus

Die Korrektheit des Pattern-Matching–Algorithmus bedeutet:

- Wenn der Pattern-Matching–Algorithmus eine Menge von Bindungen liefert, dann ergibt eine Ersetzung der Namen im Muster durch die an diese Namen gebundenen Werte genau den Wert, mit dem das Muster angeglichen wurde.
- Wenn der Pattern-Matching-Algorithmus ein Scheitern meldet, dann gibt es keine Bindungen mit dieser Eigenschaft.

Diese Aussage lässt sich durch strukturelle Induktion beweisen, wobei für jeden möglichen (Wert-)Konstruktor ein Induktionsfall nötig ist.

Korrektheit und Terminierung des Pattern-Matching–Algorithmus (II)

Die Terminierung lässt sich ebenfalls durch strukturelle Induktion beweisen. Entscheidend dabei sind die folgenden Beobachtungen:

- Die Fälle 1, 2 und 3 des Pattern-Matching-Algorithmus, die nicht zusammengesetzte Muster behandeln, terminieren offensichtlich.
- Die Terminierung des Falles 4 des Pattern-Matching-Algorithmus, der zusammengesetzte Muster behandelt, wird induktiv bewiesen.

  Dabei sind die Induktionsannahmen, dass die $\mathtt{n}$ Anwendungen des Pattern-Matching-Algorithmus auf $\mathtt{M_i}$ und $\mathtt{W_i}$ für jedes $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{n}\}$ terminieren.

9.8.6. Pattern Matching und Unifikation

Der Pattern-Matching-Algorithmus erinnert an den Unifikationsalgorithmus.

- Der Pattern-Matching-Algorithmus stellt einen Sonderfall der Unifikation dar:

  Nur in einem der beiden Parameter, dem Muster, können ungebundene Namen vorkommen.

- Offenbar führt diese Einschränkung zu einem wesentlich einfacheren Algorithmus.

- Die Linearitätsbedingung trägt ebenfalls zur Vereinfachung gegenüber dem Unifikationsalgorithmus bei.

9.8.7. Folgen der Linearätsbedingung für den Pattern-Matching–Algorithmus

- Lineare Muster ermöglichen, dass im Algorithmus die erzeugten Bindungen unbesehen in die Ergebnismenge übernommen werden können:
  - Der Ablauf des Pattern-Matching-Algorithmus hängt an keiner Stelle davon ab, welche Bindungen erzeugt wurden.
  - Die Bindungsmengen, die im Schritt 4(a)ii vereinigt werden, sind garantiert disjunkt, so dass die Vereinigung durch ein triviales Aneinanderhängen implementiert werden kann.
- Ohne die Linearitätsbedingung könnten Namen mehrfach vorkommen, so dass es bei der Vereinigung der Bindungsmengen im Schritt 4(a)ii mehrere Bindungen für denselben Namen geben könnte.

  Dann müssten bei der Vereinigung mehrfache Bindungen zusammengefasst werden. Falls diese zudem verschieden sind, liegt eine Inkonsistenz vor.

Folgen der Linearätsbedingung (II)

- Aufgrund der Linearitätsbedingung hängt der Ablauf des Pattern-Matching–Algorithmus nur vom Muster ab, das bereits zur Übersetzungszeit bekannt ist.

  Aufgrund der Linearitätsbedingung spielt die Erzeugungsreihenfolge der Bindungen keine Rolle, so dass man die rekursiven Aufrufe bereits zur Übersetzungszeit in eine einzige Schachtelung von `if-then-else` „entfalten“ kann.

  Die Struktur dieser Schachtelung hängt nur vom Muster ab.

- Neuere funktionale Sprachen mit Pattern Matching verlangen die Linearitätsbedingung noch aus einem anderen Grund:

  Sie haben wesentlich differenziertere Gleichheitstypen als SML, die z.B. jeweils mit eigenen Gleichheitsprädikaten verbunden sein können.

  $\leadsto$ Angabe des Gleichheitsprädikats bei Mehrfachvorkommen einer Variablen im Muster erforderlich ($\leadsto$ Syntax der Muster wird komplizierter!)

- Die Linearität von Mustern vereinfacht also nicht nur den Pattern-Matching-Algorithmus, die Verwendung von Muster wird einfacher und übersichtlicher.

10. Auswertung und Ausnahmen

- 10.1 Die Programmiersprache SMalL
- 10.2 Die abstrakte Syntax von SMalL
- 10.3 Ein Auswerter für SMalL: Datenstrukturen
- 10.4 Ein Auswerter für SMalL: Programm `eval1`
- 10.5 Behandlung von fehlerhaften SMalL-Ausdrücken — `eval2` und `eval3`
- 10.6 Der SML-Typ `exn` („exception“)
- 10.7 Erweiterung von SMalL um SMalL-Ausnahmen — `eval4`
- 10.8 Rein funktionale Implementierung des Auswerters — `eval5`
- 10.9 Meta- und Objektsprache, Bootstrapping

10.1. Die Programmiersprache SMalL

SMalL ist eine Vereinfachung von SML mit den folgenden Merkmalen:

- Typen in SMalL
- Verzweigung in SMalL
- Globale und lokale Deklarationen in SMalL
- Rekursive Funktionen in SMalL

10.1.1. Typen in SMalL

- SMalL bietet nur zwei Typen an: ganze Zahlen und Funktionen
- SMalL lässt Funktionen höherer Ordnung zu.
- Das Typsystem von SMalL ist nicht erweiterbar.
- In SMalL können nur einstellige Funktionen definiert werden.
  Mehrstellige Funktionen können in curried Form realisiert werden.

Über den ganzen Zahlen bietet SMalL die folgenden vordefinierten Funktionen an:

- Multiplikation einer ganzen Zahl mit -1 (im Folgenden `Minus` genannt)
- Betrag (*absolute value*) einer ganzen Zahl
- Arithmitische Funktionen (für zwei ganze Zahlen):
  Subtraktion, Multiplikation, Ganzzahlige Division,
  Rest bei einer ganzzahligen Division (im Folgenden Modulo genannt)

Einige der vordefinierten Funktionen von SMalL sind also zweistellig.

10.1.2. Verzweigung in SMalL

Zur Verzweigung bietet SMalL nur das Konstrukt `if-then-else` an.

`case`–Ausdrücke und Pattern Matching sind in SMalL nicht vorhanden.

Als Bedingung einer `if-then-else`–Verzweigung ist nur ein Vergleich von ganzen Zahlen erlaubt mit:

- Vergleichsoperatoren: $=$, $<$, $>$
- Negation

Da in SMalL Vergleiche ausschließlich als Bedingung von `if-then-else`–Ausdrücken vorkommen dürfen, wird kein eigenerTyp „Boole'sche Werte" benötigt.

10.1.3. Globale und lokale Deklarationen in SMalL

Neben globalen Deklarationen lässt SMalL lokale Deklarationen zu. Das folgende Programm ist also in SMalL genauso wie in SML möglich:

```
val a = 1;
val f = fn x => let val a = 2
                in
                    2 * a * x
                end;
val b = a;
val c = f 1;
```

Im Gegensatz zu SML ermöglicht SMalL keine (globalen oder lokalen) Deklarationen, die nichtsequenziell ausgewertet werden sollen (wie es in SML mit dem Konstrukt `and` möglich ist). Damit sind keine wechselseitig rekursiven Funktionen möglich.

10.1.4. Rekursive Funktionen in SMalL

Rekursive Funktionen sind in SMalL möglich, z.B.:

```
val rec factorial = fn x => if x > 0
                            then x * factorial(x - 1)
                            else 1;
```

10.2. Die abstrakte Syntax von SMalL

- Abstrakte Syntax versus konkrete Syntax
- SML-Typdeklarationen für SMalL-Ausdrücke
- Beispiele von SMalL-Ausdrücken in konkreter und abstrakter Syntax

10.2.1. Abstrakte Syntax versus konkrete Syntax

Der im folgenden Abschnitt behandelte Auswerter für SMalL ist ein SML-Programm, das SMalL-Ausdrücke als Aufrufparameter erhält, sie auswertet und das Ergebnis dieser Auswertung liefert.

Die SMalL-Ausdrücke, die dieser Auswerter als Aufrufparameter erhält, sind im Gegensatz zu den herkömmlichen Programmen keine Zeichenfolgen, sondern SML-Ausdrücke.

Der Auswerter für SMalL wertet also SMalL-Programme aus, die in einer anderen Syntax vorliegen als die Programme, die von den Programmierern geliefert werden.

Diese andere Syntax wird „abstrakte Syntax“ genannt; die von den Programmierern verwendete Syntax wird „konkrete Syntax“ genannt.

Abstrakte Syntax versus konkrete Syntax (II)

Betrachten wir eine (auszuführende) Programmiersprache PS und eine (ausführende) Programmiersprache AS, sowie ein AS-Programm, das die Ausführung von PS-Programmen realisieren soll.

- Das AS-Programm erhält also ein beliebiges PS-Programm als Eingabe.
- Jedes PS-Programm wird vom Programmierer als Zeichenfolge geliefert und könnte im Prinzip in dem AS-Programm auch als Wert eines AS-Typs `string` repräsentiert sein.
- Diese Zeichenfolge müsste das AS-Programm jedesmal analysieren, wenn es bei der Ausführung auf Teile des PS-Programms zugreift.

  Daher ist eine andere Darstellung des PS-Programms günstiger.
- Die Überführung des PS-Programms in eine andere Darstellung geschieht üblicherweise in mehreren aufeinander aufbauenden Phasen, die je eine Aufgabe erledigen.

Abstrakte Syntax versus konkrete Syntax (III)

Die ersten zwei Phasen sind die lexikalische Analyse und die Syntaxanalyse:

- Die lexikalische Analyse transformiert die Zeichenfolge, die das PS-Programm darstellt, in eine Folge von sogenannten „Token“, die reservierte PS-Bezeichner oder im PS-Programm definierte Bezeichner oder PS-Zahlkonstanten entsprechen.
- Die Syntaxanalyse bildet aus der Folge der „Token“ AS-Ausdrücke, die das PS-Programm darstellen.

Zusammen erzeugen also die lexikalische Analyse und die Syntaxanalyse aus einem PS-Programm in konkreter Syntax ein entsprechendes PS-Programm in abstrakter Syntax.

Prinzipiell können AS und PS auch dieselbe Sprache sein (häufig ist AS die Maschinensprache des Rechners).

10.2.2. SML-Typdeklarationen für SMalL-Ausdrücke

SMalL-Ausdrücke werden durch SML-Ausdrücke vom SML-Typ `expression` repräsentiert.

Die *abstrakte Syntax* von SMalL wird durch folgende SML-Typen definiert:

```
datatype unary_op      =  Min  |  Abs

datatype binary_op     =  Add  |  Sub  |  Mult  |  Div  |  Mod

datatype comparison_op =  ==
                       |  >>
                       |  <<
                       |  Not  of comparison_op
```

SML-Typdeklarationen für SMalL-Ausdrücke (II)

```
datatype expression = IntExp of int
                    | UnOp   of unary_op * expression
                    | BinOp  of binary_op * expression * expression
                    | If     of test * expression * expression
                    | Var    of string
                    | Dcl    of string * expression
                    | FnExp  of string * expression
                    | App    of expression * expression
                    | Seq    of expression * expression

     and test       = Test   of
                         comparison_op * expression * expression
```

Man beachte hier die Verwendung des `and`–Konstrukts von SML. Es ermöglicht die Definition von Typen wie `expression` und `test`, die wechselseitig aufeinander Bezug nehmen.

10.2.3. Beispiele von SMalL-Ausdrücken in konkreter und abstrakter Syntax

```
val a = 3;

Dcl("a", IntExp 3)
```

```
val a = 3;
a;

Seq(Dcl("a", IntExp 3), Var "a")
```

```
val a = 3;
a + a;

Seq(Dcl("a", IntExp 3), BinOp(Add, Var "a", Var "a"))
```

Beispiele von SMalL-Ausdrücken . . . (II)

```
val a = 1;
val b = 2;
val c = 3;
(a + b) * c;

Seq(
    Dcl("a", IntExp 1),
    Seq(
        Dcl("b", IntExp 2),
        Seq(
            Dcl("c", IntExp 3),
            BinOp(
                  Mult,
                  BinOp(Add, Var "a", Var "b"),
                  Var "c"
                 )
            )
        )
   )
```

Beispiele von SMalL-Ausdrücken . . . (III)

```
2 <> 3

Test(Not ==, IntExp 2, IntExp 3)
```

```
if 2 <> 3 then 1 else 0;

If(Test(Not ==, IntExp 2, IntExp 3),
   IntExp 1,
   IntExp 0)
```

```
val identity = fn x => x;

Dcl("identity", FnExp("x", Var "x"))
```

Beispiele von SMalL-Ausdrücken . . . (IV)

```
val rec factorial = fn x => if x > 0
                            then x * factorial(x - 1)
                            else 1;

     Dcl( "factorial",
          FnExp( "x",
                 If( Test( >>,
                           Var "x",
                           IntExp 0  ),
                     BinOp( Mult,
                            Var "x",
                            App( Var "factorial",
                                 BinOp( Sub,
                                        Var "x",
                                        IntExp 1  )  )
                          ),
                     IntExp 1 )
                 )
        )
```

Beispiele von SMalL-Ausdrücken . . . (V)

```
(fn x => x * x) 2

App(FnExp("x", BinOp(Mult, Var "x", Var "x")), IntExp 2)
```

```
val identity = fn x => x;
identity 3;

Seq(Dcl(
        "identity",
        FnExp("x", Var("x"))
       ),
    App(
        Var "identity",
        IntExp(3)
       )
    )
```

Beispiele von SMalL-Ausdrücken . . . (VI)

```
val f = fn x => let val local = 3
                in
                  local * x
                end;
f 2;


Seq( Dcl( "f",
          FnExp( "x",
                 Seq( Dcl( "local", IntExp 3 ),
                      BinOp( Mult, Var "local", Var "x" )
                    )
               )
        ),
     App( Var "f", IntExp 2 )
   )
```

Beispiele von SMalL-Ausdrücken . . . (VII)

Offenbar ist die abstrakte Syntax von SMalL für Menschen schwerer lesbar als die konkrete Syntax. Dagegen kann ein SML-Programm die abstrakte Syntax von SMalL leichter verarbeiten als die konkrete Syntax.

Man beachte, dass die SML-Typdefinitionen unzulässige SMalL-Ausdrücke zulassen, wie etwa die Addition zweier Funktionsdeklarationen oder die Anwendung einer ganzen Zahl auf eine ganze Zahl.

Der Auswerter für SMalL muss also solche Fälle fehlerhafter Aufrufparameter abfangen können.

10.3. Ein Auswerter für SMalL: Datenstrukturen

- Werte und Umgebungen
- Darstellung von SMalL-Werten und SMalL-Umgebungen in SML
- SML-Typdeklarationen für SMalL-Werte und SMalL-Umgebungen
- Typ des Auswerters für SMalL

10.3.1. Werte und Umgebungen

Zu den atomaren Ausdrücken in SMalL (wie in anderen Programmiersprachen) gehören vom Programmierer eingeführte Namen (Variablen, Bezeichner, *identifier*).

- Der Wert eines Namens hängt von der Umgebung ab.
- Die Umgebung ist eine Liste, die vom Anfang her nach dem jeweiligen Namen durchsucht wird bis zum ersten Vorkommen des Namens.
- Eine Umgebung kann als Liste von Gleichungen bzw. Paaren `(Name, Wert)` dargestellt werden.

  Welche Werte in Umgebungen vorkommen können, hängt von den Typen der Sprache ab.

  Im Fall von SMalL gibt es demnach zwei Arten von Werten, ganze Zahlen und Funktionen.

Werte und Umgebungen (II)

Eine ganze Zahl ist ein Wert unabhängig von jeglichem Kontext. Eine Funktion dagegen ist ein Wert, der vom Kontext abhängt. Betrachten wir dazu die folgenden SMalL-Programme:

Programm `P1`:

```
val a = 1;
val f = fn x => a * x;
val a = 2;
```

Programm `P2`:

```
val a = 1;
val a = 2;
val f = fn x => a * x;
```

- `P1` definiert `f` als die Identätsfunktion auf den ganzen Zahlen.
- `P2` definiert `f` als die Verdoppelungsfunktion auf den ganzen Zahlen.
- Die Funktionsdefinitionen in den beiden Programmen sind zwar syntaktisch gleich; in der Funktionsdefinition von `P2` gilt aber die zweite Deklaration der Variablen `a`, die die erste überschattet.

Werte und Umgebungen (III)

In einem Laufzeitsystem (etwa ein Auswerter) bestimmt die Position der Funktion in der Umgebung das Ergebnis der Funktion.
Die Positionen im (statischen) Programmtext entsprechen also unmittelbar den Positionen in der (dynamischen) Umgebung.

Für `P1` und `P2` können die erzeugten Umgebungen wie folgt dargestellt werden:

- Umgebung nach den Deklarationen von Programm `P1`:

```
[ (a, 2), (f, (fn x => a * x, . )), (a, 1) ]
```

Anfang der Umgebung   Ende der Umgebung

- Umgebung nach den Deklarationen von Programm `P2`:

```
[ (f, (fn x => a * x, . )), (a, 2), (a, 1) ]
```

Werte und Umgebungen (IV)

Dem Namen `f` ist in der Umgebung ein Paar zugeordnet, das den Funktionsausdruck sowie einen Verweis auf die Umgebung enthält, die bei Anwendung der Funktion gelten soll. Dieser Verweis ist jeweils durch den Pfeil veranschaulicht.

Betrachten wir nun die Funktionsanwendung `f 3`:
Im Fall von `P1` enthält der Wert von `f` einen Verweis auf die Umgebung

```
[ (f, (fn x => a * x, . )), (a, 1) ]
```

die zur Bindung des formalen Parameters `x` an den aktuellen Parameter `3` wie folgt erweitert wird:

```
[ (x, 3), (f, (fn x => a * x, . )), (a, 1) ]
```

In dieser Umgebung wird dann der Funktionsrumpf `a * x` zum Wert `3` ausgewertet.

Werte und Umgebungen (V)

Im Fall des Programms `P2` enthält der Wert von `f` einen Verweis auf die Umgebung

```
[ (f, (fn x => a * x, . )), (a, 2), (a, 1) ]
```

die zur Bindung des formalen Parameters `x` an den aktuellen Parameter 3 wie folgt erweitert wird:

```
[ (x, 3), (f, (fn x => a * x, . )), (a, 2), (a, 1) ]
```

In dieser Umgebung wird dann der Funktionsrumpf `a * x` zum Wert 6 ausgewertet.

Auf diese Weise wird das Prinzip der *statischen Bindung* realisiert.

10.3.2. Darstellung von SMalL-Werten und -Umgebungen in SML

SMalL-Umgebung

Eine SMalL-Umgebung wird dargestellt als eine SML-Liste von Bindungen.

SMalL-Bindung: `Bdg(name, wert)`

Eine SMalL-Bindung wird als SML-Ausdruck der Gestalt `Bdg(name,wert)` dargestellt, wobei `name` vom SML-Typ `string` und `wert` die SML-Darstellung eines SMalL-Werts ist.

SMalL-Wert „ganze Zahl”: `IntVal n`

Ein SMalL-Wert vom Typ „ganze Zahl“ wird dargestellt als SML-Ausdruck der Gestalt `IntVal n`, wobei `n` vom SML-Typ `int` ist.

Die Unterscheidung zwischen `IntExp n` (Ausdruck) und `IntVal n` (Wert) ist nicht erforderlich, erlaubt aber eine bessere Nutzung der statischen Typprüfung von SML.

Darstellung von SMalL-Werten und -Umgebungen in SML (II)

SMalL-Wert „Funktion”: `FnVal(par, body, fn_env)`

Ein SMalL-Wert vom Typ „Funktion“ wird als SML-Ausdruck der Gestalt `FnVal(par, body, fn_env)` dargestellt, wobei

- `par` vom SML-Typ Zeichenfolge (`string`) ist und den Namen des (einzigen) formalen Parameters repräsentiert,
- `body` die SML-Darstellung eines SMalL-Audrucks in abstrakter Syntax ist und den Rumpf der Funktion repräsentiert,
- `fn_env` ein Verweis auf die Umgebung ist, die bei Anwendungen der Funktion gelten soll. Dieser Verweis wird durch eine SML-Referenz realisiert.

  [dieser Verweis wird später durch rein funktionale Hilfsmittel realisiert]

Darstellung von SMalL-Werten und -Umgebungen in SML (III)

Beispiele

Darstellung der Umgebung für die Deklarationen des obigen Programms `P1`:

```
          [
           Bdg("a", IntVal 2),
fn_env1-> Bdg("f", FnVal("x",BinOp(Mult,Var "a",Var "x"),fn_env1)),
           Bdg("a", IntVal 1)
          ]
```

wobei `fn_env1` ein Verweis (eine SML-Referenz) ist.
Wenn die obige Umgebung mit `U1` bezeichnet wird, dann gilt

```
!fn_env1 = tl U1
```

Darstellung von SMalL-Werten und -Umgebungen in SML (IV)

Darstellung der Umgebung für die Deklarationen des obigen Programms `P2`:

```
          [
fn_env2-> Bdg("f",FnVal("x",BinOp(Mult,Var "a",Var "x"), fn_env2)),
          Bdg("a",IntVal 2),
          Bdg("a",IntVal 1)
          ]
```

wobei `fn_env2` ein Verweis (eine SML-Referenz) ist. Wenn die obige Umgebung mit `U2` bezeichnet wird, dann gilt

```
!fn_env2 = U2
```

10.3.3. SML-Typdeklarationen für SMalL-Werte und SMalL-Umgebungen

SMalL-Werte werden durch SML-Ausdrücke vom SML-Typ `value` repräsentiert und SMalL-Umgebungen durch SML-Listen vom SML-Typ `environment`:

```
datatype value = IntVal of int
               | FnVal  of string * expression * (binding list ref)

     and binding = Bdg  of string * value

type environment = binding list
```

[`expression` wurde bereits vorher definiert zur Darstellung von SMalL-Ausdrücken]

Der Typ des Konstruktors `FnVal` könnte in der Form `string * expression * (environment ref)` geschrieben werden. SML ermöglicht aber keine wechselseitige Bezugnahme zwischen `datatype`– und `type`–Deklarationen.
Mit dem `and`–Konstrukt können nur gleichartige Deklarationen verbunden werden.

10.3.4. Typ des Auswerters für SMalL

Der Auswerter für SMalL ist eine zweistellige Funktion namens `eval`, die als Aufrufparameter ein Paar `(exp, env)` erhält, wobei

- `exp` ein SMalL-Ausdruck in abstrakter Syntax ist (vom SML-Typ `expression`),
- `env` eine SMalL-Umgebung ist, d.h. eine SML-Liste vom SML-Typ `environment`,

und als Wert ein Paar `(val, env')` liefert, wobei

- `val` ein SMalL-Wert ist, d.h. ein SML-Ausdruck vom SML-Typ `value`,
- `env'` eine SMalL-Umgebung ist, d.h. eine SML-Liste vom SML-Typ `environment`.

Enthält der SMalL-Ausdruck `exp` (nicht-lokale) Deklarationen, so sind die Umgebungen `env` und `env'` unterschiedlich: `env'` erweitert `env` um die Bindungen, die sich aus den Deklarationen in `exp` ergeben (ansonsten ist `env'` = `env`).

Der Typ des Auswerters `eval` für SMalL lautet also:

```
expression * environment -> value * environment
```

10.4. Ein Auswerter für SMalL: Programm `eval1`

Im Folgenden werden verschiedene Versionen des Auswerters `eval` vorgestellt, die durch schrittweise Verfeinerung entwickelt werden.

Um diese Versionen zu unterscheiden, werden sie nummeriert.

Dieser Abschnitt (10.4) behandelt den Auswerter `eval1`.

Außer den bereits vorgestellten Typ-Deklarationen besteht der Auswerter lediglich aus der Definition der Funktion `eval1`. Diese Definition ist als Fallunterscheidung aufgebaut mit je einem Fall pro Konstruktor des Typs `expression`.

Die in diesem Kapitel vorgestellten Auswerter sowie Testdaten können als Dateien von der WWW-Seite des Vorlesungsskriptes heruntergeladen werden.

10.4.1. Auswertung von ganzen Zahlen

```
eval1(IntExp n, env) = (IntVal n, env : environment)
```

Die Auswertung eines Zahlausdrucks ergibt den entsprechenden Zahlwert.

Die neue Umgebung ist gleich der alten.

10.4.2. Auswertung von unären Operationen über ganzen Zahlen

```
eval1(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval1(e, env)
                          in
                             case v
                               of IntVal n
                                    => (IntVal(un_op n), env)
                          end
```

- Die Hilfsfunktion `eval_op` ordnet jedem unären SMalL-Operator (`op1`) einen unären SML-Operator (`un_op`) zu.
- Der Teilausdruck `e` wird in der Umgebung `env` zum Wert `v` ausgewertet.
- Ist `v` eine ganze Zahl, muss nur noch `un_op` darauf angewandt werden.
- Zusammen mit der unveränderten Umgebung `env` ergibt dies das Gesamtergebnis.

Auswertung von unären Operationen über ganzen Zahlen (II)

Falls `e` Deklarationen enthält, können diese zwar im rekursiven Aufruf neue Bindungen erzeugen, deren Gültigkeit aber auf den jeweiligen Aufruf beschränkt ist.

Deshalb ist die vom rekursiven Aufruf zurückgelieferte Umgebung irrelevant.

Das `case`–Konstrukt prüft hier, ob `v` eine ganze Zahl (der Gestalt `Intval n`) ist, und extrahiert `n` aus `v` (was nur durch Pattern Matching möglich ist).

Hat der Wert `v` von `e` nicht die Gestalt `IntVal n`, so liegt eine Fehler vor.
$\leadsto$ Fehlerbehandlung (wird später als Erweiterung hinzugefügt!)

Dass die vordefinierten arithmetischen Operatoren der auszuführenden Programmiersprache SMalL einfach auf entsprechende Operatoren der ausführenden Programmiersprache SML zurückgeführt werden, ist der übliche Ansatz bei der Implementierung von Programmiersprachen.

10.4.3. Auswertung von binären Operationen über ganzen Zahlen

```
eval1(BinOp(op2,e1,e2), env)
                        = let fun eval_op Add  = op +
                                | eval_op Sub  = op -
                                | eval_op Mult = op *
                                | eval_op Div  = op div
                                | eval_op Mod  = op mod
                              val bin_op       = eval_op op2
                              val (v1, _)      = eval1(e1, env)
                              val (v2, _)      = eval1(e2, env)
                          in
                             case (v1,        v2       )
                               of (IntVal n1, IntVal n2)
                                   => (IntVal(bin_op(n1,n2)), env)
                          end
```

Analog zum vorigen Fall: hier nur zwei Teilausdrücke (e1, e2) statt einen!

10.4.4. Auswertung von Verzweigungen

```
eval1(If(Test(opc,e1,e2), e_then, e_else), env)
                  = let fun eval_op ==       = op =
                          | eval_op >>       = op >
                          | eval_op <<       = op <
                          | eval_op(Not t) = not o (eval_op t)
                        val comp_op          = eval_op opc
                        val (v1, _)          = eval1(e1, env)
                        val (v2, _)          = eval1(e2, env)
                    in
                       case (v1,        v2       )
                         of (IntVal n1, IntVal n2)
                              => if comp_op(n1,n2)
                                 then eval1(e_then, env)
                                 else eval1(e_else, env)
                    end
```

10.4.5. Auswertung von Variablen (Namen, Bezeichnern)

```
eval1(Var name, env) = let fun eval_var(name, env) =
                                  case env
                                    of Bdg(id,value)::env_tl
                                        => if name = id
                                           then value
                                           else eval_var(name, env_tl)
                       in
                           (eval_var(name,env), env)
                       end
```

Die Auswertung eines SMalL-Ausdrucks der Gestalt `Var name` erfolgt dadurch, dass die Umgebung `env` von ihrem Anfang her durchsucht wird, bis der erste Eintrag `Bdg(name, value)` gefunden wird.

Dann wird der Wert `value` zusammen mit der unveränderten Umgebung `env` geliefert.

10.4.6. Auswertung von Deklarationen

```
eval1(Dcl(id,e), env) = let val (v, _) = eval1(e, env)
                        in
                         case v
                          of FnVal(par, body, fn_env)
                               => let
                                   val fn_env' = Bdg(id,v)::!fn_env
                                  in
                                   fn_env := fn_env'    ;
                                   (v, Bdg(id,v)::env)
                                  end
                           | _ => (v, Bdg(id,v)::env)
                        end
```

- Die Auswertung einer Deklaration besteht darin, die Umgebung `env` (an ihrem Anfang) um eine Bindung zu erweitern.
- Diese neue Bindung ordnet dem Namen `id` den Wert `v` des Ausdrucks `e` zu.
- Das Gesamtergebnis ist also in jedem Fall das Paar `(v, Bdg(id,v)::env)`.

Auswertung von Deklarationen (II)

- Falls `v` ein Funktionswert ist, muss dafür gesorgt werden, dass die Umgebung, die bei Anwendungen der Funktion gelten soll, ebenfalls um die Bindung `Bdg(id,v)` erweitert wird (wichtig für rekursive Funktionen).

  Diese Umgebung ist in `v` als Inhalt der SML-Referenz `fn_env` zugänglich.

  Die erweiterte Umgebung `fn_env'` wird also u.a. durch Anwendung des Dereferenzierungsoperators „`!`" berechnet, und mit der Zuweisung `fn_env := fn_env'` wird der Inhalt der SML-Referenz durch die erweiterte Umgebung ersetzt.

- Die Verwendung des Sequenzierungskonstrukts „`;`"ist notwendig, damit diese Zuweisung erfolgt, bevor das Paar `(v, Bdg(id,v)::env)` als Wert geliefert wird.

  Die Zuweisung verändert den Inhalt von `fn_env`, das ja Bestandteil von `v` ist, also wird implizit auch `v` mitverändert.

  Es wäre vielleicht klarer, dem veränderten `v` einen Namen `v'` zu geben und dann `(v', Bdg(id,v')::env)` zu schreiben.

10.4.7. `val`– versus `val-rec`–Funktionsdeklarationen

Die hier vorgenommene Behandlung von SMalL–Funktionsdeklarationen entspricht den `val-rec`–Deklarationen von SML.

Zur Implementierung der SMalL–Funktionsdeklarationen entsprechend den `val`–Deklarationen von SML braucht nur die Zuweisung

```
fn_env := fn_env'
```

weggelassen zu werden (dann kann auch die Deklaration von `fn_env'` entfallen).

`val`– versus `val-rec`–Funktionsdeklarationen (II)

Der Unterschied zwischen `val-rec`– und `val`–Deklarationen in SML ist aus dem Vergleich der folgenden Beispiele gut erkennbar:

Beispiel 1:

```
- val f = fn x => x;
val f = fn : 'a -> 'a

- val f = fn x => f x;
val f = fn : 'a -> 'a

- f 1;
val it = 1 : int
```

Beispiel 2:

```
- val f = fn x => x;
val f = fn : 'a -> 'a

- val rec f = fn x => f x;
val f = fn : 'a -> 'b

- f 1;
<< terminiert nicht >>
```

Die unterschiedliche Behandlungen des Verweises `fn_env` erklärt die unterschiedlichen Abläufe in den beiden Fällen.

10.4.8. Auswertung von Funktionsausdrücken

```
eval1(FnExp(par,e),env) = (FnVal(par, e, ref env), env)
```

- Ein Ausdruck der Gestalt `FnExp(par,e)` steht für eine Funktion mit formalem Parameter `par` und Rumpf `e`.
- Der Wert dieses Ausdrucks enthält die gleichen Bestandteile und zusätzlich einen Verweis auf die Umgebung, die bei Anwendungen der Funktion gelten soll.

  Dieser Verweis, also ein SML-Referenz, wird hier mit `ref env` erzeugt.
- Steht der `FnExp`–-Ausdruck für eine anonyme Funktion, ist die Umgebung `env`, in der der `FnExp`–Ausdruck ausgewertet wird, auch die Umgebung, die bei Anwendungen der Funktion gelten soll.
- Kommt der `FnExp`–Ausdruck als zweiter Teilausdruck in einem `Dcl`–Ausdruck vor, ist die Funktion nicht anonym.

  Dann ist die Umgebung, die bei Anwendungen der Funktion gelten soll, die um die durch den `Dcl`–Ausdruck erzeugte Bindung erweiterte Umgebung `env`.

10.4.9. Auswertung von Funktionsanwendungen

```
eval1(App(e1,e2), env) = let val (v1, _) = eval1(e1, env)
                         in
                           case v1
                             of FnVal(par, body, fn_env)
                              => let
                                   val (v2,_) = eval1(e2, env)
                                   val env' = Bdg(par,v2)::!fn_env
                                   val (v, _) = eval1(body, env')
                                 in
                                   (v, env)
                                 end
                         end
```

Auswertung von Funktionsanwendungen (II)

- Zunächst wird `e1` ausgewertet, dessen Wert `v1` eine Funktion sein muss.
- Dann wird `e2` ausgewertet, dessen Wert `v2` als aktueller Parameter an die Funktion übergeben wird.
- Die Parameterübergabe erfolgt dadurch, dass die Umgebung um eine Bindung (lokale Deklaration) für den formalen Parameter erweitert wird.

  In der so erhaltenen lokalen Umgebung `env'` wird dann der Rumpf der Funktion ausgewertet.
- Der rekursive Aufruf von `eval1` liefert zwar diese (ggf. noch weiter veränderte) lokale Umgebung zurück, das Gesamtergebnis enthält aber die ursprüngliche globale Umgebung.

  $\leadsto$ alle lokalen Deklarationen sind danach wieder weg!

10.4.10. Auswertung von Sequenzen

```
eval1(Seq(e1,e2),  env) = let val (_, env1) = eval1(e1, env)
                          in
                              eval1(e2, env1)
                          end
```

- Der zweite Ausdruck der Sequenz wird in der Umgebung ausgewertet, die sich aus der Auswertung des ersten Ausdrucks der Sequenz ergibt.

  Dies ist notwendig, weil der erste Ausdruck Deklarationen enthalten kann.

  Der Wert einer Sequenz ist in SMalL wie in SML der Wert des zweiten Ausdrucks in der Sequenz.

- Dieser Fall ist der einzige, in dem die von einem rekursiven Aufruf des Auswerters zurückgelieferte Umgebung `env1` für die Weiterverarbeitung benötigt wird.

10.4.11. Abhängigkeit des Auswerters `eval1` von SML

Der Auswerter `eval1` für SMalL ist von SML abhängig, weil er in SML implementiert ist.

Der Auswerter `eval1` führt jedoch grundlegende Funktionalitäten wie die Verwaltung der Umgebung und die Funktionsanwendung nicht auf dieselben Funktionalitäten von SML zurück, sondern implementiert sie selbst.

Lediglich Hilfsmittel wie die Auswertung der Addition werden auf die entsprechenden Funktionen von SML zurückgeführt. Dies ist bei der Implementierung von Programmiersprachen üblich.

So implementiert (und dadurch spezifiziert) `eval1` in SML einen Auswertungsalgorithmus, der von SML weitgehend unabhängig ist.

10.4.12 Gesamtprogramm des Auswerters für SMalL

. . . befindet sich in der Datei `eval1.sml`.

10.4.13. Beispiele

Laden des Auswerters in SML:

```
- use "eval1.sml";
[opening eval1.sml]
eval1.sml:48.34-50.66 Warning: match nonexhaustive
          IntVal n => ...

eval1.sml:64.34-66.72 Warning: match nonexhaustive
          (IntVal n1,IntVal n2) => ...

eval1.sml:79.34-83.67 Warning: match nonexhaustive
          (IntVal n1,IntVal n2) => ...

eval1.sml:87.39-91.78 Warning: match nonexhaustive
          Bdg (id,value) :: env_tl => ...

eval1.sml:115.34-124.47 Warning: match nonexhaustive
          FnVal (par,body,fn_env) => ...
```

Die Warnungen weisen auf unvollständige Fallunterscheidungen hin:
$\longrightarrow$ nur korrekte SMalL-Ausdrücke werden behandelt!

Beispiele (II): . . . weitere Ausgaben beim Laden

```
datatype unary_op = Abs | Min
datatype binary_op = Add | Div | Mod | Mult | Sub
datatype comparison_op = << | == | >> | Not of comparison_op
datatype expression
  = App of expression * expression
  | BinOp of binary_op * expression * expression
  | Dcl of string * expression
  | FnExp of string * expression
  | If of test * expression * expression
  | IntExp of int
  | Seq of expression * expression
  | UnOp of unary_op * expression
  | Var of string
datatype test = Test of comparison_op * expression * expression
datatype value
  = FnVal of string * expression * binding list ref | IntVal of int
datatype binding = Bdg of string * value
type environment = binding list
val eval1 = fn : expression * environment -> value * environment
val it = () : unit
```

Beispiele (III)

Auswertung von SMalL-Ausdrücken

```
- val exp_01 =
              If(Test(<<, IntExp 7, IntExp 8),
                 IntExp 1,
                 IntExp 0);
val exp_01 = If (Test (<<,IntExp #,IntExp #),IntExp 1,IntExp 0)
             : expression

- val env_01 = [];
val env_01 = [] : 'a list

- val (val_01, Env_01) = eval1(exp_01, env_01);
val val_01 = IntVal 1 : value
val Env_01 = [] : environment
```

Die Datei `eval-tests.sml` enthält eine Reihe von SMalL-Ausdrücken und Umgebungen zusammen mit den Ergebnissen der Auswertung.

Beispiele (IV)

Das System SML/NJ gibt zusammengesetzte Ausdrücke i.d.R. so aus, dass ab einer gewissen Schachtelungstiefe das Zeichen # als Abkürzung für ganze Teilausdrücke steht.

Mit den beiden Zuweisungen (z.B.)

```
Compiler.Control.Print.printDepth  := 100;
Compiler.Control.Print.printLength := 100;
```

am Anfang der Datei wird SML/NJ veranlasst, diese Abkürzungen erst bei wesentlich komplexeren Ausdrücken zu verwenden als sonst.

Eine weitere Besonderheit von SML/NJ ist die Ausgabe von zyklischen Strukturen, wie sie durch die Referenzen in `FnVal`–Ausdrücken entstehen (siehe folgendes Beispiel).

Beispiele (V)

```
- val exp_11 = Seq(Dcl("identity", FnExp("x", Var "x")),
                   App(Var "identity", IntExp 3)
                   );
val exp_11 =  Seq (Dcl ("identity",FnExp ("x",Var "x")),
                   App (Var "identity",IntExp 3))
             : expression

- val env_11 = [];
val env_11 = [] : 'a list

- val (val_11, Env_11) = eval1(exp_11, env_11);
val val_11 = IntVal 3 : value
val Env_11 = [
              Bdg("identity",
                  FnVal("x",Var "x",
                        ref [Bdg("identity",FnVal("x",Var "x",%1))
                            ] as %1)
                  )
             ]
             : environment
```

Beispiele (VI)

Die Umgebung `Env_11` enthält eine Bindung für den Namen „identity“, nämlich einen `FnVal`–Ausdruck, dessen dritter Teilausdruck ja eine Referenz auf `Env_11` ist.

Diese Referenz wird ausgegeben als `ref` gefolgt von einer Wiederholung von `Env_11`, so dass die Gefahr einer nichtterminierenden Ausgabe entsteht. Tatsächlich ist die Struktur ja zyklisch.

SML/NJ bricht die Ausgabe des Zyklus aber nach der ersten Wiederholung ab und benutzt eine symbolische Notation:

`ref [...] as %1` soll bedeuten, dass die gesamte Liste mit %1 bezeichnet wird, so dass an der Stelle, wo in der Liste das Symbol %1 selbst vorkommt, wieder die gesamte Liste gemeint ist.

Diese Notation ist recht gut lesbar, aber nur eine Ausgabenotation von SML/NJ.

Beispiele (VII)

Behandlung von fehlerhaften SMalL-Ausdrücken durch den Auswerter `eval1`:

```
- val exp_24 =
        App(IntExp 1, IntExp 1);
val exp_24 = App (IntExp 1,IntExp 1) : expression

- val env_24 = [];
val env_24 = [] : 'a list

- val (val_24, Env_24) = eval1(exp_24, env_24);
uncaught exception nonexhaustive match failure
```

Der Auswerter `eval1` kann nur korrekte SMalL-Ausdrücke behandeln:

- Bei der Kompilierung der Funktion `eval1` wird festgestellt, dass nicht alle nach Typdeklaration möglichen Fälle von der Funktion `eval1` abgedeckt werden.

  Dies führt zu den Warnungen beim Laden der Datei `eval1.sml`.

- Die Auswertung von Ausdrücken kann zu Laufzeitfehlern führen. Diese werden in der Form von vordefinierten, nicht abgefangenen SML-Ausnahmen mitgeteilt.

10.4.14. Die Start-Umgebung

Zwischen vorgegebenen SMalL-Funktionen (z.B. `Min` oder `Add`) und Funktionen, die in der Sprache SMalL selbst definiert worden sind, gibt es Unterschiede.

Selbstdefinierte Funktionen können mit Hilfe des Konstruktors `App` auf den Wert eines SMalL-Ausdrucks angewandt werden, wie im obigen Beispiel:

```
App(Var "identity", IntExp 3)
```

Das unäre `Minus` lässt sich aber nicht auf diese Weise anwenden. Versuche wie

```
App(Min,       IntExp 3)
App(UnOp Min, IntExp 3)
App(Var Min,  IntExp 3)
App(Var "Min",IntExp 3)
```

ergeben entweder Typfehler oder scheitern daran, dass gar keine Bindung für den Namen „`Min`“ existiert.

Die Start-Umgebung (II)

- Die in der Sprache SMalL definierten Funktionen können an neue Namen gebunden werden, z.B. mit `Dcl("id", Var "identity")`.
- Ferner können sie als Parameter an andere Funktionen weitergereicht werden.
- Für die vorgegebenen SMalL-Funktionen bestehen diese Möglichkeiten aber nicht.
- Dieser vermeintliche „Entwurfsfehler“ ist unvermeidbar:

  Gewisse Grundfunktionen einer Programmiersprache können nicht in der Sprache selbst definiert werden und müssen daher vorgegeben werden.
- Durch die Vorgabe einer sogenannten *Start-Umgebung* können die Namen solcher Grundfunktionen in der gleichen Weise verfügbar gemacht werden wie Namen von in der Sprache selbst definierten Funktionen.

Die Start-Umgebung (III)

Für SMalL kann die Start-Umgebung folgendermaßen definiert werden:

```
val start_env = [ Bdg("Min",
                     FnVal("x", UnOp(Min,Var "x"),
                           ref nil)),
                  Bdg("Abs",
                     FnVal("x", UnOp(Abs,Var "x"),
                           ref nil)),
                  Bdg("Add",
                     FnVal("x", FnExp("y",BinOp(Add,Var "x",Var "y")),
                           ref nil)),
                  Bdg("Sub",
                     FnVal("x", FnExp("y",BinOp(Sub,Var "x",Var "y")),
                           ref nil)),
                  Bdg("Mult",
                     FnVal("x", FnExp("y",BinOp(Mult,Var "x",Var "y")),
                           ref nil)),
                  Bdg("Div",
                     FnVal("x", FnExp("y",BinOp(Div,Var "x",Var "y")),
                           ref nil)),
                  Bdg("Mod",
                     FnVal("x", FnExp("y",BinOp(Mod,Var "x",Var "y")),
                           ref nil))                                     ]
```

Die Start-Umgebung (IV)

Wertet man SMalL-Ausdrücke nicht in der leeren Umgebung aus, sondern in dieser Start-Umgebung, stehen die Namen der vorgegebenen Funktionen genau so zu Verfügung, als seien sie in SMalL selbst definiert worden:

```
- eval1( App(Var "Min", IntExp 1), start_env );
val it =
  (IntVal ~1,
   [Bdg("Min",FnVal("x",UnOp(Min,Var "x"),ref [])),
    Bdg("Abs",FnVal("x",UnOp(Abs,Var "x"),ref [])),
    Bdg("Add",FnVal("x",FnExp("y",BinOp(Add,Var "x",Var "y")),ref [])),
    Bdg("Sub",FnVal("x",FnExp("y",BinOp(Sub,Var "x",Var "y")),ref [])),
    Bdg("Mult",FnVal("x",FnExp("y",BinOp(Mult,Var "x",Var "y")),ref [])),
    Bdg("Div",FnVal("x",FnExp("y",BinOp(Div,Var "x",Var "y")),ref [])),
    Bdg("Mod",FnVal("x",FnExp("y",BinOp(Mod,Var "x",Var "y")),ref []))])
  : value * environment
```

Die Start-Umgebung (V)

Auch SML startet nicht in einer völlig leeren Umgebung, sondern gibt Bindungen für gewisse Namen durch seine Start-Umgebung vor:

```
linux% sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
- ~;
val it = fn : int -> int
```

Der Name ~ wird also vom System zur Verfügung gestellt, allerdings (in SML/NJ) nicht so, als sei die zugehörige Funktion in SML selbst definiert:

```
- ~ 1;
val it = ~1 : int                  (* OK *)
- ~1;
val it = ~1 : int                  (* OK *)
- val ~ = fn x => x;
val ~ = fn : 'a -> 'a              (* ~ ueberschatten *)
- ~ 1;
val it = 1 : int                   (* OK *)
- ~1;
val it = ~1 : int                  (* ?? *)
```

10.5. Behandlung fehlerhafter SMalL-Ausdrücke — `eval2` und `eval3`

- Schwäche des Auswerters `eval1` für SMalL
- Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2`
- Veränderter Auswerter für SMalL mit Sonderwerten
- Unzulänglichkeit des veränderten Auswerters mit Sonderwerten
- Verbesserung des Auswerters mit SML-Ausnahmen — `eval3`

10.5.1. Schwäche des Auswerters `eval1` für SMalL

Der Auswerter für SMalL des vorangehenden Abschnitts ist gegen fehlerhafte SMalL-Ausdrücke wie die folgenden nicht gewappnet:

- `App(IntExp 1, IntExp 1)`
  (der erste Parameter einer Funktionsanwendung muss eine Funktion sein)
- `UnOp(Min, FnExp("x", Var "x"))`
  (der unäre Operator `Min` ist nur auf ganzen Zahlen aber nicht auf Funktionen definiert)

Wird der Auswerter `eval1` auf einen fehlerhaften SMalL-Ausdruck angewendet, so führt dies zu einem Laufzeitfehler:

```
- eval1(UnOp(Min, FnExp("x", Var "x")), []);
uncaught exception nonexhaustive match failure
```

Diese Fehlermeldung zeigt: Kein Fall in der Deklaration der SML-Funktion `eval1` behandelt fehlerhafte Parameter, d.h., fehlerhafte SMalL-Ausdrücke.

Schwäche des Auswerters `eval1` für SMalL (II)

Beispiel: Die Auswertung von unären Operationen über ganzen Zahlen

```
eval1(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval1(e, env)
                          in
                             case v
                               of IntVal n
                                    => (IntVal(un_op n), env)
                          end
```

enthält keine Behandlung für den Fall, dass der Wert `v` des Ausdrucks `e` nicht die erwartete Gestalt `IntVal n` sondern die unerlaubte Gestalt `FnVal(...)` hat.

10.5.2. Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2`

Der Auswerter `eval1` kann dadurch verbessert werden, dass dem Typ `value` weitere Werte und den `case`–Konstrukten zur Zerlegung der SMalL-Ausdrücke weitere Fälle hinzugefügt werden ($\rightarrow$ `eval2`).

Wenn der Auswerter für SMalL verwendet wird, können zwei grundlegende Laufzeitfehler auftreten:

1. Ein SMalL-Ausdruck, d.h. ein SML-Ausdruck vom Typ `expression`, ist nicht korrekt im Sinne der Programmiersprache SMalL, wie etwa
   `App(IntExp 1, IntExp 1)` oder `UnOp(Min, FnExp("x", Var "x"))`.
2. Ein SMalL-Variable hat in der Umgebung keine Bindung, wie z.B. bei der Auswertung von `UnOp(Min, Var "a")` in der leeren Umgebung.

Mit zwei unterschiedlichen zusätzlichen Werten können die beiden Fälle unterschieden und so dem Benutzer verständliche Fehlermeldungen gegeben werden.

Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2` (II)

Die Ergänzung des SML-Typs `value` um weitere Werte kann wie folgt geschehen (die Ergänzung findet sich in der Datei `eval2.sml`):

```
datatype value = IntVal of int
               | FnVal  of string * expression * (binding list ref)
               | Error_illegal_expression
               | Error_unbound_variable
```

Die beiden letzten Zeilen sind gegenüber der ersten Version des Auswerters neu.

Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2` (III)

Der Fall zur Auswertung von Variablen wird wie folgt erweitert:

```
eval2(Var name, env) = let fun eval_var(name, env) =
                              case env
                               of Bdg(id,value)::env_tl
                                       => if name = id
                                          then value
                                          else eval_var(name, env_tl)
                                | nil => Error_unbound_variable
                       in
                           (eval_var(name,env), env)
                       end
```

Der einzige Unterschied zu `eval1` ist der zweite Fall im case–Konstrukt.

Prinzip der Verbesserung des Auswerters mit Sonderwerten — `eval2` (IV)

Der Fall zur Auswertung von unären Operationen wird wie folgt erweitert:

```
eval2(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval2(e, env)
                          in
                          case v
                            of IntVal n
                                 => (IntVal(un_op n), env)
                             | _ => (Error_illegal_expression, env)
                          end
```

Der einzige Unterschied zu `eval1` ist wieder der zweite Fall im case–Konstrukt.

In allen anderen Fällen, in denen eine Erweiterung notwendig ist, ist die Erweiterung von der gleichen Art wie in diesem Fall.

10.5.3. Veränderter Auswerter für SMalL mit Sonderwerten

Das veränderte Programm findet sich in der Datei `eval2.sml`.

Gegenüber `eval1` sind hinzugekommen:

- die beiden neuen Zeilen zur Deklaration der Sonderwerte des Datentyps `value` sowie
- jeweils eine neue Zeile pro `case`–Konstrukt, die einen Fangfall einführt, in dem einer der Sonderwerte geliefert wird.

Korrekte SMalL-Ausdrücke führen mit `eval2` zu genau den gleichen Ergebnissen wie mit `eval1`.

[siehe z.B. die Ausdrücke in der Datei `eval-tests.sml`]

10.5.4. Unzulänglichkeit des veränderten Auswerters mit Sonderwerten

In einigen Fällen fehlerhafter SMalL-Ausdrücke verhält sich `eval2` einigermaßen zufriedenstellend:

```
- eval2(App(IntExp 1, IntExp 1), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(UnOp(Min, FnExp("x", Var "x")), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(Var "a", []);
val it = (Error_unbound_variable,[]) : value * environment
```

Die Fehlermeldung beschreibt jeweils zutreffend die Fehlerursache. Nicht ganz zufriedenstellend ist, dass die Fehlermeldungen als Teilausdrücke von zusammengesetzten Ausdrücken erscheinen.

Unzulänglichkeit des veränderten Auswerters . . . (II)

Die Fehlermeldung

```
Error_illegal_expression
```

wäre sicherlich der augegebenen Fehlermeldung vorzuziehen:

```
(Error_illegal_expression,[])
```

- Dass in allen Fehlerfällen nur die lesbarere Fehlermeldung als Ergebnis von `eval2` geliefert wird, erweist sich aber als erstaunlich schwierig (wenn überhaupt möglich).
- Das Programm muss dafür grundlegend verändert werden.
- In der Praxis überwiegt der Nachteil des Veränderungsaufwands die Nachteile der etwas unzufriedenstellenden Fehlermeldungen.

Unzulänglichkeit des veränderten Auswerters . . . (III)

Aber in anderen Fällen sind die Fehlermeldungen von `eval2` einfach irreführend:

```
- eval2(UnOp(Min, Var "a"), []);
val it = (Error_illegal_expression,[]) : value * environment

- eval2(BinOp(Add, BinOp(Add, Var "a", Var "b"), Var "c"), []);
val it = (Error_illegal_expression,[]) : value * environment
```

Die Fehlerursache ist in beiden Fällen eine fehlende Variablenbindung, so dass die Fehlermeldung `Error_unbound_variable` angebracht wäre.

- Zu der irreführenden Fehlermeldung `Error_illegal_expression` kommt es, weil die Sonderwerte rekursiv weitergereicht werden.
- Wenn ein rekursiver Aufruf von `eval2` den Sonderwert `Error_unbound_variable` liefert, dann führt dieser Wert in der nächsthöheren Rekursionsstufe in den Fangfall des `case`–Konstrukts, was zum Sonderwert `Error_illegal_expression` führt.

Unzulänglichkeit des veränderten Auswerters . . . (IV)

Um die Weiterreichung der „richtigen“ Sonderwerte könnte etwa der Fall zur Auswertung von unären Operationen wie folgt verändert werden:

```
eval2(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval2(e, env)
                          in
                           case v
                             of IntVal n
                                 => (IntVal(un_op n), env)
                              | Error_unbound_variable
                                 => (v, env)
                              | Error_illegal_expression
                                 => (v, env)
                              | _ => (Error_illegal_expression, env)
                          end
```

Unzulänglichkeit des veränderten Auswerters . . . (V)

Für jeden Sonderwert würde also ein Fall im case–Konstrukt hinzukommen. In anderen Fällen würde dies schon aufwändiger werden:

```
eval2(BinOp(op2,e1,e2), env)
                    = let
                          ...                          (* wie vorher *)
                      in
                        case (v1,        v2        )
                          of (IntVal n1, IntVal n2)
                              => (IntVal(bin_op(n1,n2)), env)
                           | (Error_unbound_variable, _)
                              => (v1, env)
                           | (Error_illegal_expression, _)
                              => (v1, env)
                           | (_, Error_unbound_variable)
                              => (v2, env)
                           | (_, Error_illegal_expression)
                              => (v2, env)
                           | _ => (Error_illegal_expression, env)
                      end
```

Unzulänglichkeit des veränderten Auswerters . . . (VI)

- Die hier skizzierte Erweiterung um zwei Sonderwerte würde die Definition von `eval2` um etwa ein Drittel umfangreicher als die von `eval1`.
- Für aussagekräftigere Fehlermeldungen wären aber deutlich mehr Sonderwerte erforderlich:

  $\rightsquigarrow$ die vielen zu berücksichtigen Kombinationen würden eine inakzeptable und unübersichtliche Aufblähung des Auswerters führen

$\Rightarrow$ keine zufriedenstellende Lösung!

10.5.5. Verbesserung des Auswerters mit SML-Ausnahmen — `eval3`

SML bietet sogenannte Ausnahmen (*exceptions*) an, um Programme mit einer Behandlung vonfehlerhaften Aufrufparametern zu ergänzen.

- Bei der Verwendung von SML-Ausnahmen sind die Veränderungen des ursprünglichen Programms minimal, so dass die Struktur und die Logik dieses Programms erhalten bleiben.
- Ausnahmen sind keine Werte, sondern eine Gattung von Programmierobjekten für sich.
- Ausnahmen werden mit dem Konstrukt `raise` „erhoben“ oder „geworfen“, ähnlich wie Sonderwerte zur Fehlermeldung im vorangehenden Abschnitt verwendet wurden.

Der Auswerter `eval3` behandelt fehlerhafte SMalL-Ausdrücke mit Hilfe von SML-Ausnahmen.

Verbesserung des Auswerters mit SML-Ausnahmen . . . (II)

Die Unterschiede zwischen `eval2.sml` und `eval3.sml` sind zum einen, dass statt der zwei Sonderwerte zwei SML-Ausnahmen deklariert werden. Statt:

```
datatype value = IntVal of int
               | FnVal  of string * expression * (binding list ref)
               | Error_illegal_expression
               | Error_unbound_variable
```

lauten die Deklarationen jetzt

```
datatype value = IntVal of int
               | FnVal  of string * expression * (binding list ref)
exception illegal_expression
exception unbound_variable
```

Verbesserung des Auswerters mit SML-Ausnahmen . . . (III)

Ein weiterer Unterschied ist, dass alle Fangfälle von `eval2` der Gestalt:

```
case ...
  of ...
   | _ => (Error_illegal_expression, env)
```

für `eval3` die Gestalt haben:

```
case ...
  of ...
   | _ => raise illegal_expression
```

Entsprechend wurde

```
   | nil => Error_unbound_variable
```

zu

```
   | nil => raise unbound_variable
```

Ansonsten sind die Definitionen von `eval2` und `eval3` identisch. Die Struktur des Auswerters `eval2` bleibt also völlig erhalten.

Verbesserung des Auswerters mit SML-Ausnahmen . . . (IV)

Die Verwendung von SML-Ausnahmen verursacht also keine wesentliche Verlängerung des Programms. Aber das Verhalten des Auswerters ist jetzt wie gewünscht:

```
- eval3(App(IntExp 1, IntExp 1), []);
uncaught exception illegal_expression

- eval3(UnOp(Min, FnExp("x", Var "x")), []);
uncaught exception illegal_expression

- eval3(Var "a", []);
uncaught exception unbound_variable

- eval3(UnOp(Min, Var "a"), []);
uncaught exception unbound_variable

- eval3(BinOp(Add, BinOp(Add, Var "a", Var "b"), Var "c"), []);
uncaught exception unbound_variable
```

Verbesserung des Auswerters mit SML-Ausnahmen . . . (V)

Die Fehlermeldung (in Gestalt einer Ausnahme) beschreibt jeweils zutreffend die Fehlerursache, auch in den Fällen, in denen `eval2` eine irreführende Fehlermeldung (in Gestalt eines Sonderwerts) liefert.

Die Mitteilung „`uncaught exception`“, d.h. „nicht eingefangene Ausnahme“ bezieht sich auf die Möglichkeit, Ausnahmen zu behandeln oder „einzufangen“, von der im Auswerter `eval3` kein Gebrauch gemacht wird.

Die Programmierung von „Ausnahmebehandlern“ wird im nächsten Abschnitt erläutert.

Alle nicht fehlerhaften SMalL-Ausdrücke in der Datei `eval-tests.sml` führen mit `eval3` zu genau den gleichen Ergebnissen wie mit `eval2` und mit `eval1`.

10.6. Der SML-Typ `exn` („exception“)

Dieser Abschnitt behandelt allgemein die Verwendung von SML-Ausnahmen — unabhängig von Auswertern.

- Der vordefinierte Typ `exn`
- Ausnahmekonstruktoren
- Ausnahmen erheben (oder werfen)
- Ausnahmen als Werte
- Ausnahme versus Wert
- Ausnahmen behandeln (oder einfangen)
- Prinzip der Auswertung von `raise`– und `handle`–Ausdrücken
- Vordefinierte Ausnahmen von SML

10.6.1. Der vordefinierte Typ `exn`

- vordefinierter Typ `exn` in SML (*exception* oder *Ausnahme*)
- Werte dieses Typs heißen Ausnahmewerte (*exception values*)
- Besonderheit dieses Typs:

  der Programmierer kann dem Typ neue (Wert-)Konstruktoren hinzufügen.

  (dies ist für keinen anderen vordefinierten Typ möglich)
- Die neuen Wertkonstruktoren des vordefinierten Typs `exn` werden Ausnahmekonstruktoren (*exception constructors*) genannt.

10.6.2. Ausnahmekonstruktoren

Ein Ausnahmekonstruktor namens `A` wird wie folgt deklariert:

```
exception A;
```

Ausnahmekonstruktoren können konstant sein wie etwa in der folgenden Deklaration aus dem Auswerter `eval3.sml`

```
exception illegal_expression;
```

oder einen Parameter von irgend einem Typ `t` haben.

Ein Ausnahmekonstruktor namens `A` mit Parameter vom Typ `t` wird wie folgt deklariert:

```
exception A of t;
```

Ausnahmen können (mit `let`) auch lokal deklariert werden.

10.6.3. Ausnahmen erheben (oder werfen)

Eine Ausnahme `A` kann mit folgendem Ausdruck erhoben (oder geworfen) werden:

```
raise A
```

Beispiel der Deklaration und Erhebung einer konstanten Ausnahme:

```
- exception negative_integer;
exception negative_integer

- fun factorial x = if x < 0
                    then raise negative_integer
                    else if x = 0
                         then 1
                         else x * factorial(x - 1);
val factorial = fn : int -> int

- factorial ~4;
uncaught exception negative_integer
```

Ausnahmen erheben (oder werfen) (II)

Beispiel der Deklaration und Erhebung einer Ausnahme mit Parameter:

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x = if x < 0
              then raise negative_argument(x)
              else if x = 0
                   then 1
                   else x * fac(x - 1);
val fac = fn : int -> int

-  fac ~4;
uncaught exception negative_argument
```

Der Ausnahmeparameter ~4 wird hier nicht angezeigt, da SML/NJ (wie in anderen Fällen) die Ausgabe verkürzt und damit die volle Struktur nicht sichtbar wird.

10.6.4. Ausnahmen als Werte

Ausnahmekonstruktoren werden (fast) wie normale Wertkonstruktoren verwendet:

- Ausnahmekonstruktoren können zum Aufbau von neuen Werten und zum Pattern Matching verwendet werden.

  So können z.B. Listen von Ausnahmewerten gebildet werden.

- Ausnahmen können (mit `let`) auch lokal definiert werden, z.B.:

```
fun f x = let  exception invalid_argument
          in   if x < 0
                  then raise invalid_argument
                  else if x = 0
                          then 1
                          else x * f(x - 1)
          end;
```

Lokale Ausnahmedeklarationen können wegen gleicher Namen zu Verständnisproblemen führen. Daher sollten Ausnahmen möglichst nur global deklariert werden!

10.6.5. Ausnahme versus Wert

Wie herkömmliche Werte können Ausnahmen das Ergebnis einer Auswertung sein z.B. bei Anwendung der Fakultätsfunktion `factorial` auf eine negative ganze Zahl:

```
- factorial ~4;
uncaught exception negative_integer
```

Ausnahmen werden aber während der Auswertung nicht wie Werte von herkömmlichen Typen behandelt, sondern wie folgt:

> Liefert die Auswertung eines Teilausdruckes `T` eines zusammengesetzten Ausdrucks `B` eine Ausnahme `A` als Ergebnis, so wird diese Ausnahme `A` als Ergebnis der Auswertung des Gesamtausdrucks `B` geliefert, es sei denn, `A` wird von einem Ausnahmebehandler eingefangen.

Die Sonderstellung von Ausnahmen während der Auswertung kann am folgenden Beispiel beobachtet werden:

Ausnahme versus Wert (II)

```
- exception negative_integer;
exception negative_integer

- fun factorial x = if x < 0
                    then raise negative_integer
                    else if x = 0
                         then 1
                         else x * factorial(x - 1);
val factorial = fn : int -> int

- fun is_even x = let val x_mod_2 = x mod 2
                  in
                      case x_mod_2
                        of 0 => true
                         | _ => false
                  end;
val is_even = fn : int -> bool

- is_even(factorial ~4);
uncaught exception negative_integer
```

Ausnahme versus Wert (III)

- Aufgrund der applikativen Auswertungsreihenfolge führt die Auswertung von `is_even(factorial ~4)` zunächst zur Auswertung von `factorial ~4`.

  Die Auswertung von `factorial ~4` liefert Ausnahme `negative_integer`.

- Anschließend müsste `is_even(negative_integer)` ausgewertet werden.

  Der Ausnahmewert `negative_integer` ist aber nicht vom erwarteten Parametertyp `int`, kann also bei der Auswertung nicht wie ein „normaler" Wert behandelt werden (dies würde zu einem Typfehler zur Laufzeit führen).

- Bei einer Behandlung als „Sonderwert" würde `(negative_integer mod 2)` zu `negative_integer` ausgewertet.

  Dies würde in den Fangfall des case–Konstrukts führen und als Gesamtergebnis `false` ergeben.

Im Beispiel ergibt sich aber weder ein Typfehler noch `false` als Ergebnis. Ausnahmewerte werden also anders als andere Werte behandelt.

10.6.6. Ausnahmen behandeln (oder einfangen)

Ein Ausnahmebehandler (*exception handler*) ist eine Art Funktion, die nur Parameter vom Typ `exn` haben kann. Er hat im einfachsten Fall die Gestalt `handle A => C`.

Ein Ausdruck `B` kann mit einem Ausnahmebehandler verknüpft werden zu

```
B handle A => C
```

- Liefert die Auswertung von `B` (oder von einem Teilausdruck `T` in `B`) die Ausnahme `A`, so wird der Ausnahmebehandler `handle A => C` wirksam.

  Dann wird `C` ausgewertet und der Wert von `C` als Wert von `B` geliefert.

  Die Ausnahme `A` wird also nicht weitergereicht.
- Liefert die Auswertung von `B` etwas anderes als die Ausnahme `A`, so hat der Ausnahmebehandler `handle A => C` keine Wirkung.

Anschaulich kann man sagen, dass ein Teilausdruck eine Ausnahme in Richtung seiner umfassenden Ausdrücke wirft. Die Ausnahme wird von den umfassenden Ausdrücken einfach durchgelassen, bis sie von einem Ausnahmebehandler eingefangen wird.

Ausnahmen behandeln (oder einfangen) (II)

Das Beispiel der Fakultätsfunktion `factorial` kann wie folgt mit einem Behandler für die Ausnahme `negative_integer` ergänzt werden:

```
- exception negative_integer;
exception negative_integer

- fun factorial x = (if x < 0
                        then raise negative_integer
                        else if x = 0
                                then 1
                                else x * factorial(x - 1)
                    )
                     handle negative_integer => factorial(~x);
val factorial = fn : int -> int

- factorial ~4;
val it = 24 : int
```

Die auftretende Ausnahme `negative_integer` wird von dem Behandler eingefangen, was zur Auswertung von `factorial(~(~4))` führt.

Ausnahmen behandeln (oder einfangen) (III)

Die Ausnahmebehandlung kann auch anders erfolgen:

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x = if x < 0
              then raise negative_argument(x)
              else if x = 1
                   then 1
                   else x * fac(x - 1);
val fac = fn : int -> int

- fac ~4 handle negative_argument(y) => fac(~y);
val it = 24 : int
```

Ausnahmen behandeln (oder einfangen) (IV)

In beiden Beispielen tritt in den gleichen Fällen eine Ausnahme auf und ihre Behandlung führt auch zu den gleichen Ergebnissen:

- Im ersten Beispiel wird die Ausnahme im Rumpf der Funktion eingefangen, d.h. im Geltungsbereich des formalen Parameters `x` der Funktion `factorial`.
- Im zweiten Beispiel ist der Behandler außerhalb des Geltungsbereiches des formalen Parameters `x` der Funktion `fac` definiert.

  Der Wert `~4` des aktuellen Parameters des Aufrufes wird über die einstellige Ausnahme `negative_argument` an den Behandler weitergereicht.

Die Definition eines Behandlers besteht i.Allg. aus einem Angleichmodell:

```
handle <Muster1> => <Ausdruck1>
     | <Muster2> => <Ausdruck2>
                 .
                 .
                 .
     | <Mustern> => <Ausdruckn>
```

10.6.7. Prinzip der Auswertung von `raise`– und `handle`–Ausdrücken

Die Auswertung eines Ausdrucks

```
raise <Ausdruck>
```

führt zur Auswertung von `<Ausdruck>`, der ein Ausdruck vom Typ `exn` sein muss.

Liefert die Auswertung von `<Ausdruck>` einen Ausnahmewert `<Ausnahme>`, so ist der Wert des Ausdrucks

```
raise <Ausdruck>
```

das sogenannte Ausnahmepaket `[ <Ausnahme> ]`.

Der Begriff Ausnahmepaket dient hauptsächlich dazu, Ausnahmewerte von herkömmlichen Werten zu unterscheiden.

Achtung: Die Schreibweise `[...]` ist keine SML-Notation (und insb. keine Liste).

Prinzip der Auswertung von `raise` und `handle` (II)

- Bei der Auswertung eines zusammengesetzten Ausdrucks werden zunächst die Teilausdrücke ausgewertet.
- Ist einer der dabei ermittelten Werte ein Ausnahmepaket, wird die weitere Auswertung der Teilausdrücke abgebrochen und das Ausnahmepaket als Wert geliefert.
- So wird das Ausnahmepaket als Ergebnis der Auswertungen sämtlicher umfassender Ausdrücke weitergereicht, bis ein Behandler gefunden wird.
- Bei der Auswertung eines Ausdrucks

  ```
  <Ausdruck1> handle <Muster> => <Ausdruck2>
  ```

  wird zunächst `<Ausdruck1>` ausgewertet.

  – Ist der Wert kein Ausnahmepaket, wird dieser Wert geliefert.

  – Ist der Wert ein Ausnahmepaket `[ <Ausnahme> ]`, erfolgt Pattern Matching zwischen `<Muster>` und dem Ausnahmewert `<Ausnahme>`.

    Bei Erfolg wird `<Ausdruck2>` ausgewertet und dessen Wert geliefert, bei Misserfolg wird das Ausnahmepaket `[ <Ausnahme> ]` geliefert.

Prinzip der Auswertung von `raise` und `handle` (III)

Die Auswertung von `raise`– und `handle`–Ausdrücken kann durch das folgende Beispiel illustriert werden:

```
- exception negative_zahl;

- fun f x = if x = 0
            then true
            else if x > 0
                 then f(x - 1)
                 else raise negative_zahl;
val f = fn : int -> bool

- fun g true  = "wahr"
    | g false = "falsch";
val g = fn : bool -> string

- g(f ~3) handle negative_zahl => "Fehler";
val it = "Fehler" : string
```

Prinzip der Auswertung von `raise` und `handle` (IV)

Die Auswertung des letzten Ausdrucks kann unter Anwendung des Substitutionsmodells wie folgt erläutert werden:

```
g(f ~3) handle negative_zahl => "Fehler"

g(if ~3 = 0
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g(if false
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"
```

Prinzip der Auswertung von `raise` und `handle` (V)

```
g(      if ~3 > 0
        then f(~3 - 1)
        else raise negative_zahl ) handle negative_zahl => "Fehler"

g(      if false
        then f(~3 - 1)
        else raise negative_zahl ) handle negative_zahl => "Fehler"

g( raise negative_zahl ) handle negative_zahl => "Fehler"

g( [ negative_zahl ] ) handle negative_zahl => "Fehler"

[ negative_zahl ] handle negative_zahl => "Fehler"

"Fehler"
```

`[ negative_zahl ]` stellt ein sogenanntes Ausnahmepaket dar.

Prinzip der Auswertung von `raise` und `handle` (VI)

Ausnahmen sind ihrer Natur nach eng an die Auswertungsreihenfolge gekoppelt. Mit SML-Ausnahmen kann man herausfinden, in welcher Reihenfolge Teilausdrücke ausgewertet werden:

```
- exception a;
exception a
- exception b;
exception b

- ((raise a) + (raise b)) handle b => 2
                              | a => 1;
val it = 1 : int
```

Wäre zuerst der zweite Teilausdruck von + ausgewertet worden, so wäre das Gesamtergebnis 2 gewesen.

10.6.8. Vordefinierte Ausnahmen von SML

Einige Ausnahmen sind in SML vordefiniert, z.B. die Ausnahmen `Match` und `Bind`, die bei Fehlern während des Pattern Matching erhoben werden.

Die Standardbibliothek von SML beschreibt die vordefinierten Ausnahmen.

10.7. Erweiterung von SMalL um SMalL-Ausnahmen — `eval4`

`eval4` ist eine Erweiterung von `eval3` um SMalL-Ausnahmen.

Die SMalL-Ausnahmen unterscheiden sich von den SML-Ausnahmen dadurch, dass SMalL nur konstante Ausnahmen (d.h. Ausnahmen ohne Parameter) zulässt.

Die Erweiterung des Auswerters `eval4` um die Behandlung von Ausnahmen mit Parametern stellt keine Schwierigkeit dar.

Sie wurde unterlassen, um den erweiterten Auswerter einfach zu halten und so das Prinzip der Ausnahmebehandlung besser zur Geltung zu bringen.

10.7.1. Erweiterung der Programmiersprache SMalL — Konkrete Syntax

Die konkrete Syntax von SMalL wird um die beiden folgenden Konstrukte erweitert:

```
raise <Ausnahme>

<Ausdruck1> handle <Ausnahme> => <Ausdruck2>
```

Im Gegensatz zu SML werden SMalL-Ausnahmen, die in `raise`– oder `handle`–Ausdrücken vorkommen, nicht in `exception`–Deklarationen vereinbart.

So müssen SMalL-Programme in konkreter Syntax um etwaige `exception`–Deklarationen ergänzt werden, damit sie als SML-Programme vom SML-Laufzeitsystem verarbeitet werden können.

Dies ist die einzige Ergänzung, die nötig ist, damit aus einem SMalL-Programm in konkreter Syntax ein SML-Programm wird.

10.7.2. Erweiterung der Programmiersprache SMalL — Abstrakte Syntax

Die abstrakte Syntax von SMalL wird wie folgt erweitert.

Ein Ausdruck in konkreter Syntax der Gestalt

```
raise exn
```

wird dargestellt als:

```
Throw(exn')
```

Ein Ausdruck in konkreter Syntax der Gestalt

```
exp1 handle exn => exp2
```

wird dargestellt als:

```
Catch(exp1', exn', exp2')
```

wobei die Zeichen ' auf die Übersetzung in abstrakte Syntax hinweisen.

10.7.3. Erweiterung der SML-Typdeklarationen des Auswerters

- Der Typ `expression` wird erweitert um die zusätzlichen Wertkonstruktoren `Throw` und `Catch`. Sie definieren die abstrakte Syntax der neuen SMalL-Ausdrücke.
- Namen von SMalL-Ausnahmen haben den SML-Typ Zeichenfolge (`string`).
- Der Typ `value` erhält den zusätzlichen Wertkonstruktor `ExnVal` zur Darstellung der Ausnahmepakete.

Die Typdeklarationen sind also gegenüber denen für `eval3` wie folgt erweitert:

```
datatype expression =
                     .
                     .
                     | Throw  of string
                     | Catch  of expression * string * expression

datatype value      =
                     .
                     .
                     | ExnVal of string
```

10.7.4. Erweiterung des Auswerters

Zur Behandlung von SMalL-Ausnahmen benötigt `eval4` drei Erweiterungen:

- einen zusätzlichen Fall zur Auswertung von `Throw`–Ausdrücken;
- einen zusätzlichen Fall zur Auswertung von `Catch`–Ausdrücken;
- jeder andere Fall des Auswerters, in dessen Rumpf ein SMalL–Ausnahmepaket auftreten kann, bedarf einer Erweiterung des `case`–Konstrukts, die für das Weiterreichen der Ausnahmepakete sorgt.

Fall zur Auswertung eines `Throw`–Ausdrucks

```
eval4(Throw exn, env) = (ExnVal exn, env)
```

Der Wert eines `Throw`–Ausdrucks ist einfach das Ausnahmepaket für denselben Ausnahme–Namen `exn`. Die Umgebung bleibt unverändert.

Erweiterung des Auswerters (II)

Fall zur Auswertung eines `Catch`–Ausdrucks

```
eval4(Catch(e,exn,handler), env)
                          = let val (v, _) = eval4(e, env)
                              in
                                 case v
                                   of ExnVal exn1
                                        => if exn = exn1
                                           then eval4(handler, env)
                                           else (v, env)
                                    | _ => (v, env)
                              end
```

Zunächst wird der Teilausdruck `e` in der Umgebung `env` zum Wert `v` ausgewertet. Das Paar `(v, env)` ist das Ergebnis der gesamten Auswertung, außer wenn `v` ein Ausnahmepaket mit gleichem Ausnahme-Namen wie im `Catch`–Ausdruck ist; in diesem Fall werden alle Folgen der Auswertung von e verworfen und statt dessen der Ausnahmebehandler in der Umgebung `env` ausgewertet.

Erweiterung des Auswerters (III)

Fall zur Auswertung eines `UnOp`–Ausdrucks

```
eval4(UnOp(op1,e), env) = let fun eval_op Min = ~
                                | eval_op Abs = abs
                              val un_op       = eval_op op1
                              val (v, _)      = eval4(e, env)
                          in
                             case v
                               of IntVal n
                                    => (IntVal(un_op n), env)
                                | ExnVal _
                                    => (v, env)
                                | _ => raise illegal_expression
                          end
```

Die einzige Änderung dieses Falls gegenüber `eval3` ist der neue Fall `ExnVal _` im case–Konstrukt, damit das Ausnahmepaket unverändert weitergereicht wird.

Erweiterung des Auswerters (IV)

Fall zur Auswertung eines `BinOp`**–Ausdrucks**

```
eval4(BinOp(op2,e1,e2), env)
                    = let fun eval_op Add  = op +
                            | eval_op Sub  = op -
                            | eval_op Mult = op *
                            | eval_op Div  = op div
                            | eval_op Mod  = op mod
                          val bin_op       = eval_op op2
                          val (v1, _)      = eval4(e1, env)
                          val (v2, _)      = eval4(e2, env)
                      in
                          (* ... naechste Folie *)
```

Erweiterung des Auswerters (V)

```
          case (v1,         v2        )
            of (IntVal n1, IntVal n2)
                 => (IntVal(bin_op(n1,n2)), env)
             | (ExnVal _,   _         )
                 => (v1, env)
             | (_,         ExnVal _ )
                 => (v2, env)
             | _ => raise illegal_expression
      end
```

Auch hier sind lediglich die neuen Fälle `ExnVal _` für das unveränderte Weiterreichen der Ausnahmepakete hinzugekommen.

Alle anderen betroffenen Fälle von `eval4` werden in gleicher Weise erweitert, damit das Ausnahmepaket unverändert weitergereicht wird.

10.7.5. Beispiele

Alle SMalL-Ausdrücke in der Datei `eval-tests.sml` führen mit `eval4` zu genau den gleichen Ergebnissen wie mit `eval3`.

Die Datei `eval4-tests.sml` enthält zusätzliche Beispiele mit den neuen Konstruktoren `Throw` und `Catch`, die von `eval1`, `eval2`, `eval3` nicht ausgewertet werden können.

10.7.6. Weitergabe von Sonderwerten und Ausnahmebehandlung im Vergleich

Der Auswerter `eval4` ist in seinem Prinzip sehr ähnlich zu der vorher skizzierten und dann kritisierten Erweiterung von `eval2`.

Während die Erweiterung von `eval2` SML-Sonderwerte weiterreichen muss, muss `eval4` SMalL-Ausnahmepakete weiterreichen.

Dies geschieht auf fast die gleiche Weise, wie das folgende Beispiel zeigt.

Weitergabe von Sonderwerten und Ausnahmen . . . (II)

Fall `Unop`, Erweiterung von `eval2`:

```
case v
  of IntVal n
       => (IntVal(un_op n), env)
   | Error_unbound_variable
       => (v, env)
   | Error_illegal_expression
       => (v, env)
   | _ => (Error_illegal_expression, env)
```

Fall `Unop`, `eval4`:

```
case v
  of IntVal n
       => (IntVal(un_op n), env)
   | ExnVal _
       => (v, env)
   | _ => raise illegal_expression
```

Weitergabe von Sonderwerten und Ausnahmen . . . (III)

Die Ähnlichkeit der Vorgehensweise sollte nicht überraschen, weil in beiden Versionen dasselbe Problem gelöst werden soll.

- Die Programmstücke für `eval4` zur Behandlung von Ausnahmen sind kompakter.

  Das Muster `ExnVal _` reicht zur Erfassung jedes beliebigen Ausnahmepakets, egal wie viele weitere Ausnahme–Namen hinzukommen.

- In der Erweiterung von `eval2` wächst die Anzahl der zusätzlichen Fälle mit der Anzahl der Sonderwerte stark an.

- Die Erweiterung des Datentyps `value` um nullstellige (Wert-)Konstruktoren (hier: `Error_illegal_expression` und `Error_unbound_variable`) für `eval2` ist aber eigentlich nicht nötig.

  Ein einstelliger Wertkonstruktor `ErrorVal of string` würde den gleichen Zweck erfüllen, womit auch nur noch ein Muster in jedem case–Ausdruck benötigt würde.

Mit `eval4` sollte hier auch nur demonstriert werden, wie die typische Ausnahmebehandlung in Programmiersprachen funktioniert.

10.8. Rein funktionale Implementierung des Auswerters — `eval5`

- Verweise als Funktionen
- Ein rein funktionaler Auswerter für SMalL

10.8.1. Verweise als Funktionen

- Eine Umgebung kann eine Bindung für eine Funktion enthalten.
- Diese Bindung enthält als Bestandteil diejenige Umgebung, die mit der Bindung selbst beginnt, die wiederum die Umgebung enthält, die die Bindung enthält, usw.
- Offenbar liegt hier ein Zyklus vor.

Betrachten wir ein wesentlich einfacheres Beispiel für einen solchen Zyklus:

- Gegeben sei ein Typ `T` mit einem zweistelligen Konstruktor `K`, und wir betrachten ein Gebilde `A` der Gestalt `K(1, .)`, wobei das zweite Argument von `K` das Gebilde `A` selbst sein soll.
- Versucht man, `A` in der gewohnten Form niederzuschreiben, stellt man fest, dass das Niederschreiben nicht terminiert.

  Deshalb ist `A` auch kein Ausdruck und wird hier nur als „Gebilde“ bezeichnet.

Verweise als Funktionen (II)

Um `A` niederschreiben zu können, benötigt man ein zusätzliches Mittel für die Darstellung, zum Beispiel einen Pfeil wie in den Diagrammen von Abschnitt 10.3.1:

```
K( 1, . )
```

Was durch diesen Pfeil illustriert wird, nennt man abstrakt einen *Verweis*. Der Verweis ist nicht dasselbe wie `A`, sondern liefert `A`, wenn er „verfolgt“ wird.

Da das zweite Argument von `K` nicht direkt `A` ist, sondern etwas, das erst verfolgt werden muss, um `A` zu erhalten, ist eine Indirektion eingeführt, die den Zyklus unterbricht und eine endliche Darstellung ermöglicht.

Diese Abstraktion kann unter anderem durch SML-Referenzen implementiert werden (siehe folgendes Beispiel).

Verweise als Funktionen (III)

```
- datatype T = K of int * (T ref) | K0;
datatype T = K of int * T ref | K0

- fun zyklusK n = let val verweis = ref K0
                      val A = K(n, verweis)
                  in
                      verweis := A ;      (* Zyklus erzeugen *)
                      A
                  end;
val zyklusK = fn : int -> T

- val A as K(n,verweis) = zyklusK 1;
val A        = K (1,ref (K (#,#)))  : T
val n        = 1                    : int
val verweis  = ref (K (1,%0)) as %0 : T ref

- val A' as K(n',verweis') = !verweis; (* Verweis verfolgen *)
val A'       = K (1,ref (K (#,#)))  : T
val n'       = 1                    : int
val verweis' = ref (K (1,%1)) as %1 : T ref
```

Verweise als Funktionen (IV)

In dieser Implementierung ist

- ein „Verweis“ eine SML-Referenz,
- der Zyklus entsteht mit Hilfe des Zuweisungsoperators „`:=`“, und
- „verfolgen“ bedeutet Anwenden des Dereferenzierungsoperators „`!`“.

Eine andere Realisierung beruht auf der Beobachtung, dass eine nullstellige Funktion nicht dasselbe ist wie ihr Wert. Sie muss erst auf `()` angewandt werden, um diesen Wert zu liefern.

In dieser folgenden, rein funktionalen Implementierung ist daher

- ein „Verweis“ eine nullstellige Funktion,
- der Zyklus entsteht durch Rekursion (`fun` bedeutet ja `val rec`), und
- „verfolgen“ bedeutet Anwenden der nullstelligen Funktion auf `()`.

Es werden also keine Konstrukte mit Nebeneffekten (wie die Zuweisung) benötigt!

Verweise als Funktionen (V)

```
- datatype T  = K of int * (unit -> T) | K0;
datatype T = K of int * (unit -> T) | K0

- fun zyklusK n =let
                  fun verweis() = K(n,verweis)     (* Zyklus .. *)
                  val A = verweis()                (*  erzeugen *)
                 in
                     A
                 end;
val zyklusK = fn : int -> T

- val A as K(n,verweis) = zyklusK 1;
val A        = K(1,fn) : T
val n        = 1       : int
val verweis  = fn      : unit -> T

- val A' as K(n',verweis') = verweis();      (* Verweis folgen *)
val A'       = K(1,fn) : T
val n'       = 1       : int
val verweis' = fn      : unit -> T
```

10.8.2. Ein rein funktionaler Auswerter für SMalL

Der Auswerter `eval5` implementiert im Gegensatz zu `eval4` die Verweise für die Zyklen zwischen Umgebungen und Bindungen durch nullstellige Funktionen.

Insgesamt unterscheiden sich die beiden Versionen nur an vier Stellen:

1. In der `datatype`–Deklaration für den Typ `value` ändert sich

```
| FnVal  of string * expression * (binding list ref)
```

   zu

```
| FnVal  of string * expression * (unit -> binding list)
```

2. Im Fall für die Auswertung von Funktionsausdrücken ändert sich

```
(FnVal(par, e, ref env), env)
```

   zu

```
(FnVal(par, e, fn()=>env), env)
```

Die Änderungen 1. und 2. ergeben sich daraus, dass ein Verweis jetzt keine SML-Referenz sondern eine nullstellige Funktion ist.

Ein rein funktionaler Auswerter für SMalL (II)

3. Im Fall für die Auswertung von Funktionsanwendungen ändert sich

   ```
   val env'   = Bdg(par,v2)
                 :: !fn_env
   ```

   zu

   ```
   val env'   = Bdg(par,v2)
                 :: fn_env()
   ```

Die Änderung 3. ergibt sich daraus, wie ein Verweis verfolgt wird.

Ein rein funktionaler Auswerter für SMalL (III)

4. Im Fall für die Auswertung von Deklarationen ändert sich

```
case v of FnVal(par, body, fn_env)
            => let
                   val fn_env' = Bdg(id,v) :: !fn_env
               in
                   fn_env := fn_env'   ;
                   (v, Bdg(id,v)::env)
                 end
```

zu

```
case v of FnVal(par, body, fn_env)
            => let
                     fun fn_env'() = Bdg(id,v'()) :: fn_env()
                 and v'() = FnVal(par,body,fn_env')
               in
                  (v'(), Bdg(id,v'())::env)
               end
```

Ein rein funktionaler Auswerter für SMalL (IV)

- Der Zyklus entsteht in `eval5` durch die wechselseitige Rekursion zwischen `fn_env'` und `v'`.
- Da SML wechselseitige Rekursion nur zwischen Funktionen erlaubt, aber nicht zwischen einer Funktion und einer Konstanten, muss `v'` ebenfalls als (nullstellige!) Funktion definiert werden.

  Wo vorher `v` stand, steht also hinterher `v'()`.

`eval5` liefert in jedem Fall die gleichen Ergebnisse wie die Version `eval4`:

- Die dritten Argumente von `FnVal` werden vom SML-System anders ausgegeben.
- Wo vorher `ref [...]` stand, steht jetzt `fn`.

Diese Änderung der Repräsentation ist unabhängig von allen vorherigen Modifikationen des Auswerters. Man kann so jede der bisherigen Versionen des Auswerters in eine rein funktionale Implementierung umschreiben.

10.9. Meta- und Objektsprache, Bootstrapping

(mit aus der Linguistik entliehenen Begriffen)

- In der Linguistik nennt man *Metasprache* eine (natürliche) Sprache in der Aspekte einer anderen Sprache (der sogenannten *Objektsprache*) beschrieben werden.
- Die Metasprache ist die Implementierungs- oder ausführende Sprache (hier SML).
  Die Objektsprache ist die implementierte oder auszuführende Sprache (hier SMalL).
- In der Informatik (wie auch in der Linguistik) kommt es häufig vor, dass Meta- und Objektsprache dieselbe Sprache sind.

Meta- und Objektsprache, Bootstrapping (II)

Verschiedene Sprachebenen bei der Implementierung einer Programmiersprache:

- Man kann in einer (meist Maschinen-)Sprache den Kern (wie etwa SMalL) einer Programmiersprache (wie SML) implementieren.
- Dann können in dieser einfachen Kernsprache sämtliche Erweiterungen dieser Kernsprache implementiert werden,

  z.B. zusammengesetzte Typen, Pattern Matching oder die statische Typprüfung.
- In der Regel geschehen solche Erweiterungen stufenweise, so dass mehrere Zwischensprachen zwischen der Kern- und Zielsprache entstehen.
- Dieser Implementierungsansatz heißt *Bootstrapping*.
- Vorteil des Bootstrapping:

  Nur die Implementierung der Kernsprache muss für jeden Computertyp und jedes Betriebssystem geleistet werden muss.

  Dadurch erleichtert Bootstrapping die Portierung von Programmiersprachen.

11. Bildung von Abstraktionsbarrieren mit abstrakten Typen und Modulen

- 11.1 Vorzüge des Verbergens
- 11.2 Abstrakte Typen in SML
- 11.3 Beispiel: Abstrakte Typen zur Implementierung der Datenstruktur „Menge“
- 11.4 Module in SML
- 11.5 Hinweis auf die Standardbibliothek von SML

11.1. Vorzüge des Verbergens

Die „zentrale“ Funktionalität eines Programms lässt sich oft in relativ wenigen und übersichtlichen Prozeduren bzw. Funktionen fassen.

Darüber hinaus gehende Anforderungen führen sehr oft zu einer erheblichen Verlängerung und Verkomplizierung des Programmtexts, wie z.B.

- Fehlerbehandlung mit verständlichen Fehlermeldungen
- komfortable Ein- und Ausgabe

Möglichkeiten, um größere Programme einfach und übersichtlich zu gestalten:

- Aufteilung in Teilprogramme (Prozeduren, Funktionen), die je eine (Teil-)Aufgabe implementieren (vgl. Kapitel 4)
- Abstraktion durch Prozeduren höherer Ordnung (vgl. Kapitel 7)
- Definition und Verwendung von abstrakten Datentypen

Vorzüge des Verbergens (II)

- Ein *abstrakter Typ* ergänzt eine Typdefinition (`datatype`-Deklaration in SML) um Namen und Prozeduren, deren Implementierung aber außerhalb der Definition des abstrakten Typs unsichtbar ist.
- Ein *Modul* ist ein Teilprogramm, das lokale Definitionen beinhalten kann.

  Für ein Modul wird jeweils festgelegt, welche Definitionen des Moduls außerhalb des Moduls, welche nur innerhalb des Moduls verwendet werden können.
- Sowohl bei abstrakten Typen als auch bei Modulen kann erreicht werden, dass nur die zur Verfügung gestellten Operationen verwendet werden.
- Das Prinzip, nur die für die Verarbeitung notwendigen Namen und Operationen zur Verfügung zu stellen, aber die interne Repräsentation von Werten zu verbergen, ist unter dem Namen *information hiding* bekannt.

  Es ist eines der wichtigsten Hilfsmittel bei der Softwareentwicklung.

11.2. Abstrakte Typen in SML

- Motivationsbeispiel: Das Farbmodell von HTML
- Ein SML-Typ zur Definition des Farbmodells von HTML
- Ein abstrakter Typ zur Definition der Farben von HTML
- Implementierungstyp und Schnittstelle eines abstrakten Typs
- Vorteile des Verbergens mit abstrakten Typen

11.2.1. Motivationsbeispiel: Das Farbmodell von HTML

Die Markup-Sprache (oder Auszeichnungssprache) HTML sowie die Style-Sheet-Sprache (oder Formatierungssprache) CSS1 repräsentieren Farben nach RGB–Modell.

Hierin wird jede Farbe durch Anteile an den drei Grundfarben Rot, Grün und Blau definiert, wobei diese Anteile natürliche Zahlen von 0 bis 255 sind.

Das RGB-Tripel (0, 0, 0) entspricht der Farbe Schwarz, das RGB-Tripel (255, 255, 255) entspricht der Farbe Weiß.

In der Spezifikation von HTML 4.0 sind insgesamt 16 Farbennamen (vor-)definiert.

Dieses Farbmodell wird im Folgenden durch eine `datatype`–Deklaration in SML definiert, deren Nachteile dann durch einen abstrakten Typ überwunden werden können.

11.2.2. Ein SML-Typ zur Definition des Farbmodells von HTML

Mit einer `datatype`–Deklaration kann in SML ein Typ `color` definiert werden:

```
- datatype color = rgb of int * int * int;
datatype color = rgb of int * int * int
```

Mit dieser Typdeklaration können HTML-Farbnamen wie folgt deklariert werden, z.B.:

```
val Black   = rgb(  0,   0,   0);
val Gray    = rgb(128, 128, 128);
val Red     = rgb(255,   0,   0);
val Green   = rgb(  0, 128,   0);
val Lime    = rgb(  0, 255,   0);
val Blue    = rgb(  0,   0, 255);
val White   = rgb(255, 255, 255);
```

Man beachte, dass die grüne Farbe mit maximalem Grünanteil `Lime` heißt.
`Green` bezeichnet die grüne Farbe mit halbem Grünanteil.

Ein SML-Typ zur Definition des Farbmodells von HTML (II)

Eine Funktion zum Mischen zweier Farben kann wie folgt definiert werden:

```
- exception negative_part;
- fun mix(color1, part1, color2, part2) =
        if part1 < 0 orelse part2 < 0
        then raise negative_part
        else let val parts = part1 + part2;
                 val rgb(r1, g1, b1) = color1;
                 val rgb(r2, g2, b2) = color2;
                 val r = (part1*r1 + part2*r2) div parts;
                 val g = (part1*g1 + part2*g2) div parts;
                 val b = (part1*b1 + part2*b2) div parts
             in
                 rgb(r, g, b)
             end;
val mix = fn : color * int * color * int -> color
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = rgb (85,84,84) : color
- mix(rgb(1,1,1), 1, rgb(128,128,128), 1);
val it = rgb (64,64,64) : color
```

Ein SML-Typ zur Definition des Farbmodells von HTML (III)

Die vorangehende `datatype`–Deklaration hat einige Nachteile:

- Der Typkonstruktor `rgb` und die Tripel-Gestalt einer Farbe sind sichtbar.

  So kann die Funktion `mix` nicht nur auf Farbennamen sondern auch auf `rgb`–Tripel angewendet werden.

- Es wird nicht verhindert, dass ungültige `rgb`–Tripel mit negativen Komponenten oder mit Komponenten größer als 255 vom Programmierer verwendet werden:

```
- val a = rgb(~1,0,999);
val a = rgb (~1,0,999) : color
- mix(rgb(~12,~12,1200),5, rgb(~20,~20,2000),3);
val it = rgb (~15,~15,1500) : color
```

- Deklarationen von Farbennamen wie `Black`, `Red`, `Blue` und `Lime` und Funktionen wie `mix` können auch weit auseinander liegend vorkommen.

  Es fehlt eine syntaktische Einheit, die alle Deklarationen zusammenfasst, die mit RBG-Farben zu tun haben.

11.2.3. Ein abstrakter Typ zur Definition der Farben von HTML

Ein abstrakter Typ für die Farben kann in SML wie folgt deklariert werden:

```
abstype color = rgb of int * int * int
with  val Black   = rgb(  0,   0,   0);
      val Silver  = rgb(192, 192, 192);
      val Gray    = rgb(128, 128, 128);
      val White   = rgb(255, 255, 255);
      val Maroon  = rgb(128,   0,   0);
      val Red     = rgb(255,   0,   0);
      val Purple  = rgb(128,   0, 128);
      val Fuchsia = rgb(255,   0, 255);
      val Green   = rgb(  0, 128,   0);
      val Lime    = rgb(  0, 255,   0);
      val Olive   = rgb(128, 128,   0);
      val Yellow  = rgb(255, 255,   0);
      val Navy    = rgb(  0,   0, 128);
      val Blue    = rgb(  0,   0, 255);
      val Teal    = rgb(  0, 128, 128);
      val Aqua    = rgb(  0, 255, 255);
```

Ein abstrakter Typ zur Definition der Farben von HTML (II)

```
    exception negative_part;
    fun mix(color1, part1, color2, part2) =
           if part1 < 0 orelse part2 < 0
           then raise negative_part
           else let val parts = part1 + part2;
                    val rgb(r1, g1, b1) = color1;
                    val rgb(r2, g2, b2) = color2;
                    val r = (part1*r1 + part2*r2) div parts;
                    val g = (part1*g1 + part2*g2) div parts;
                    val b = (part1*b1 + part2*b2) div parts
                in
                    rgb(r, g, b)
                end;
    fun display(rgb(r,g,b)) = print(Int.toString r  ^  " "  ^
                                    Int.toString g  ^  " "  ^
                                    Int.toString b  ^  "\n");
end;
```

Ein abstrakter Typ zur Definition der Farben von HTML (III)

Dies ist eine syntaktische Einheit, die die Deklarationen der Farbennamen, der Ausnahme `negative_part` und der Funktionen `mix` und `display` einschließt.

So wird ein abgeschlossenes Teilprogramm gebildet, das alle Deklarationen umfasst, die zu dem neuen Typ `color` gehören und zu dessen Definition beitragen.

Die Mitteilung des SML-Systems als Reaktion auf die obige Deklaration ist

```
type color
val Black = - : color
val Silver = - : color
val Gray = - : color
    [ ... ]                      (* hier weggelassen *)
val Teal = - : color
val Aqua = - : color
exception negative_part
val mix = fn : color * int * color * int -> color
val display = fn : color -> unit
```

Daran fällt auf, dass die innere Gestalt einer Farbe, d.h. die (Wert-) Konstruktoren des neuen Typs `color`, in dieser Mitteilung nirgends erwähnt wird.

Ein abstrakter Typ zur Definition der Farben von HTML (IV)

Tatsächlich ist diese innere Gestalt der Werte des Typs außerhalb der `abstype`–Deklaration nicht mehr verwendbar:

```
- mix(rgb(1, 1, 1), 1, Gray, 1);
Error: unbound variable or constructor: rgb
```

Dagegen sind die Namen der Farben und Funktionen der `abstype`–Deklaration außerhalb der `abstype`–Deklaration verwendbar — sonst wäre ihre Deklaration auch nutzlos. Nur die innere Gestalt ihrer Werte wird verborgen:

```
- Red;
val it = - : color
- mix(Red, 1, Red, 1);
val it = - : color
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = - : color
- display(mix(Red, 1, mix(Lime, 1, Blue, 1), 2));
85 84 84
val it = () : unit
```

Ein abstrakter Typ zur Definition der Farben von HTML (V)

- Ein Programm, das den Typ `color` verwendet, kann auf die dort definierten Farben zugreifen und neue Farben mit der zur Verfügung gestellten Funktion `mix` erzeugen.
- Farben können auf keine andere Weise gebildet werden, insbesondere auch nicht mit dem Konstruktor `rgb`.
- Damit sind ungültige `rgb`–Tripel (mit negativen Werten oder Werten größer als 255 als Komponenten) ausgeschlossen.

11.2.4. Implementierungstyp und Schnittstelle eines abstrakten Typs

- Die Definition eines abstrakten Typs besteht aus einer Typdefinition und Wertdefinitionen, die auch Funktionsdefinitionen sein können.
- Der Typ, der in einer `abstype`–Deklaration definiert wird, wird *Implementierungstyp* des abstrakten Typs genannt.
- Die Deklarationen in einer `abstype`–Deklaration, einschließlich der Deklarationen von Funktionen, bilden die sogenannte *Schnittstelle* des abstrakten Typs.
- Ein Programm, das einen abstrakten Typ verwendet, d.h., sich darauf bezieht, wird manchmal „Klient“ dieses abstrakten Typs genannt.

11.2.5. Vorteile des Verbergens mit abstrakten Typen

Unabhängigkeit der Klienten von der Implementierung eines Typs

- Wird ein abstrakter Typ in einem Programm verwendet, so verlangt eine Veränderung seiner Definition keine Veränderung seiner Klienten, so lange die Schnittstelle des abstrakten Typs unverändert bleibt.

  Beispiel:
  Eine Verfeinerung des Farbmodells, so dass nicht nur 256 sondern 1024 Farbwerte erlaubt sind, ist bei unveränderter Schnittstelle möglich.

- Da in dem Programm, das den abstrakten Typ verwendet, keine Implementierungsdetails des abstrakten Typs vorkommen können, ist das Programm unabhängig von etwaigen Implementierungsänderungen des abstrakten Typs.

- Dies erleichtert eine stufenweise Entwicklung und Wartung von komplexen Programmen.

Vorteile des Verbergens mit abstrakten Typen (II)

Einkapselung zur sicheren Wertverarbeitung

- Werte eines abstrakten Datentyps können nur unter Verwendung der zur Verfügung gestellten Namen und Funktionen erzeugt und manipuliert werden werden.
- Diese *Einkapselung* verhindert eine Verarbeitung der Werte eines abstrakten Typs, die bei der Implementierung des Typs nicht beabsichtigt wurde.
- In der Praxis entstehen häufig Fehler, wenn Werte eines Typs in einer Weise verarbeitet werden, die bei der Implementierung des Typs nicht beabsichtigt wurde.

Die Einkapselungstechnik stellt einen wesentlichen Beitrag zur Erstellung korrekter und sicherer Software dar.

11.3. Beispiel: Abstrakte Typen zur Implementierung der Datenstruktur „Menge“

Im Folgenden betrachten wir endliche Mengen von ganzen Zahlen, die extensional definiert sind.

Der Übersichtlichkeit halber werden nur die folgenden Konzepte definiert:

- der Name `empty_set` zur Definition der leeren Menge;
- das Prädikat `is_empty` zur Überprüfung, ob eine Menge leer ist;
- das Prädikat `is_element` zur Überprüfung der Elementbeziehung;
- die Funktion `insert`, um ein Element in eine Menge einzufügen;
- die Funktion `remove`, um ein Element aus einer Menge zu entfernen;
- die Funktion `union`, um zwei Mengen zu vereinigen;
- die Funktion `display`, um die Elemente einer Menge zu drucken.

Beispiel: Abstrakte Typen für die Datenstruktur „Menge“ (II)

2 unterschiedliche abstrakte Typen mit derselben Schnittstelle:

- Der erste abstrakte Typ stellt Mengen als unsortierte Listen ohne Wiederholungen von Elementen dar.
- Der zweite abstrakte Typ verfeinert diese erste Darstellung mit der Berücksichtigung der kleinsten und größten Mengenelemente zur Beschleunigung der Funktionen `is_element`, `insert` und `remove`.

Ein durchaus realistisches Szenario ist, dass im Rahmen eines größeren Programmierprojekts zunächst die einfachere erste Implementierung realisiert wird.

Zur Effizienzsteigerung wird dann später die zweite Version implementiert.

Wenn die Mengen als abstrakte Typen realisiert sind, ist die Umstellung auf die zweite Implementierung möglich, ohne den Rest des Programms zu verändern.

11.3.1. Erster abstrakter Typ zur Implementierung der Menge als Liste

Mengen von ganzen Zahlen als unsortierte Listen ohne Wiederholung von Elementen:

```
- abstype set = Set of int list
    with
        val empty_set                 = Set nil;
        fun is_empty x                = (x = Set nil);
        fun is_element(_, Set nil)    = false
          | is_element(e, Set(h::t)) = (e = h) orelse
                                        is_element(e, Set t);
        fun insert(e, Set nil)        = Set(e::nil)
          | insert(e, Set(h::t))      =
                        if e = h
                        then Set(h::t)
                        else let val Set L = insert(e, Set t)
                             in
                                Set(h::L)
                             end;
```

Erster abstrakter Typ zur Implementierung der Menge . . . (II)

```
        fun remove(e, Set nil)   = Set nil
          | remove(e, Set(h::t)) =
                       if e = h
                       then Set t
                       else let val Set L = remove(e, Set t)
                            in
                               Set(h::L)
                            end;
        fun union(Set nil,   M)     = M
          | union(Set(h::t), Set L) =
                       if is_element(h, Set L)
                       then union(Set t, Set L)
                       else let val Set L1 = union(Set t, Set L)
                            in
                               Set(h::L1)
                            end;
        fun display(Set nil)   = print "\n"
          | display(Set(h::t)) = ( print(Int.toString(h) ^ " ") ;
                                   display(Set t) );
   end;
```

Erster abstrakter Typ zur Implementierung der Menge . . . (III)

```
type set
val empty_set = - : set
val is_empty = fn : set -> bool
val is_element = fn : int * set -> bool
val insert = fn : int * set -> set
val remove = fn : int * set -> set
val union = fn : set * set -> set
val display = fn : set -> unit

- is_empty(remove(1, insert(1, empty_set)));
val it = true : bool
- display(remove(2, insert(3, insert(2, insert(1, empty_set)))));
1 3
val it = () : unit
- is_element(1,  insert(1, empty_set));
val it = true : bool
- val set1 = insert(3, insert(2, insert(1, empty_set)));
val set1 = - : set
```

11.3.2. Implementierung der Menge als Liste: 2. Version

Eine Menge von ganzen Zahlen wird nun als Tripel `(k, g, L)` dargestellt, wobei

- `k` das kleinste Element der Menge ist,
- `g` das größte Element der Menge ist,
- `L` die unsortierte Liste (ohne Wiederholung) der Mengenelemente ist.

```
- abstype set = Set of int * int * int list
              | EmptySet
    with val empty_set                              = EmptySet;
         fun is_empty x                             = (x = EmptySet);
         fun is_element(_, EmptySet)                = false
           | is_element(e, Set(k, g, h::t)) =
                  let fun member(_, nil)  = false
                        | member(m, h::t) = if m = h
                                            then true
                                            else member(m, t)
                  in if e < k orelse e > g
                     then false
                     else member(e, h::t)
                  end;
```

Implementierung der Menge als Liste: 2. Version (II)

```
fun insert(e, EmptySet)                = Set(e, e, e::nil)
  | insert(e, Set(k, g, L))            =
        if e < k
        then Set(e, g, e::L)
        else if e > g
             then Set(k, e, e::L)
             else let fun add(e, nil)  = e::nil
                        | add(e, h::t) = if e = h
                                         then h::t
                                         else h::add(e,t)
                  in
                     Set(k, g, add(e, L))
                  end;
```

Implementierung der Menge als Liste: 2. Version (III)

```
fun remove(e, EmptySet)              = EmptySet
  | remove(e, Set(k, g, L))          =
        if e < k orelse e > g
        then Set(k, g, L)
        else let
               fun remove(e, nil)   = nil
                 | remove(e, h::t) = if e = h
                                     then t
                                     else h::remove(e,t);
               fun min(h::nil)      = h
                 | min(h::t)        = Int.min(h, min t);
               fun max(h::nil)      = h
                 | max(h::t)        = Int.max(h, max t);
               val L'               = remove(e, L)
             in
               if L' = nil
               then EmptySet
               else Set(min L', max L', L')
             end;
```

Implementierung der Menge als Liste: 2. Version (IV)

```
fun union(s, EmptySet)        = s
  | union(EmptySet, s)        = s
  | union(Set(k1, g1, L1), Set(k2, g2, L2)) =
       let fun member(_, nil)  = false
             | member(m, h::t) = if m = h
                                 then true
                                 else member(m, t);
           fun merge(nil, L)  = L
             | merge(h::t, L) = if member(h, L)
                                then merge(t, L)
                                else h::merge(t, L)
       in
         Set(Int.min(k1,k2),Int.max(g1,g2),merge(L1,L2))
       end;
```

Implementierung der Menge als Liste: 2. Version (V)

```
      fun display(EmptySet)          = print "\n"
        | display(Set(_, _, h::t)) =
              let fun display_list(nil)  = print "\n"
                    | display_list(h::t) =
                        ( print(Int.toString(h) ^ " ");
                          display_list(t)
                        );
              in
                 display_list(h::t)
              end
     end;
type set
val is_empty = fn : set -> bool
val is_element = fn : int * set -> bool
val insert = fn : int * set -> set
val remove = fn : int * set -> set
val union = fn : set * set -> set
val display = fn : set -> unit
```

Implementierung der Menge als Liste: 2. Version (VI)

```
- is_empty(remove(1, insert(1, empty_set)));
val it = true : bool

- display(remove(2, insert(3, insert(2, insert(1, empty_set)))));
3 1
val it = () : unit

- is_element(1,  insert(1, empty_set));
val it = true : bool

- val set1 = insert(3, insert(2, insert(1, empty_set)));
val set1 = - : set
```

Der einzige Unterschied gegenüber der ersten Version liegt in der Reihenfolge der Mengenelemente in der Liste, die die Menge darstellt.

Dieser Unterschied kommt daher, dass die Operationen `insert` und `remove` in unterschiedlicher Weise implementiert sind.

11.4. Module in SML

Module ermöglichen, größere Programme hierarchisch zu strukturieren.
Die Modulbegriffe von SML heißen:

- Struktur,
- Signatur,
- Funktor.

In einer SML-*Struktur* können Werte (Funktionen sind in SML auch Werte), Typen und Ausnahmen definiert werden.

Eine SML-*Signatur* teilt mit, was eine Struktur implementiert, ohne die Implementierung preiszugeben. Eine Signatur ist also dem Typ einer Funktion ähnlich, der in abgekürzter Form (abstrakt) wiedergibt, was die Funktion leistet, ohne die Implementierung der Funktion preiszugeben.

Ein SML-*Funktor* ist ein parametrisiertes Modul. Ein SML-Funktor kann auch als eine Art Funktion angesehen werden, womit Strukturen auf Strukturen abbildet werden.

Module in SML (II)

Die folgende Ähnlichkeit gilt also:

- Strukturen sind Werten ähnlich.
- Signaturen sind Typen ähnlich.
- Funktoren sind Funktionen ähnlich.

Im Folgenden werden SML-Strukturen, SML-Signaturen und SML-Funktoren näher erläutert.

11.4.1. SML-Strukturen

Deklarationen können in einer SML-Struktur zusammengefasst werden.

Beispiel: eine Struktur zur Definition eines Typs „komplexe Zahlen“:

```
structure Complex =
 struct
  type t                             = real * real;
  val zero                           = (0.0, 0.0) : t;
  fun sum       ((x1,y1):t, (x2,y2):t) = (x1 + x2, y1 + y2) : t;
  fun difference((x1,y1):t, (x2,y2):t) = (x1 - x2, y1 - y2) : t;
  fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2 - y1 * y2,
                                          x1 * y2 + x2 * y1) : t;
  fun reciprocal((x,y) : t)            = let val r = x * x + y * y
                                         in
                                             (x/r, ~y/r) : t
                                         end;
  fun quotient  (z1 : t,    z2 : t)    = product(z1, reciprocal z2)
 end;
```

SML-Strukturen (II)

In einer Struktur deklarierte Namen sind außerhalb der Struktur nicht sichtbar; z.B.:

```
- reciprocal(1.0, 0.0);
Error: unbound variable or constructor: reciprocal
```

Ein Name `N`, der in einer Struktur `S` deklariert ist, kann außerhalb von `S` als `S.N` verwendet werden; z.B.:

```
- Complex.reciprocal(1.0, 0.0);
val it = (1.0,0.0) : Complex.t
```

Dient eine Struktur `S` zur Definition eines Typs `t`, so wird in der Struktur dieser Typ `t`, außerhalb dieser Struktur `S.t` genannt:

```
- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- Complex.product(i, i);
val it = (~1.0,0.0) : Complex.t
```

SML-Strukturen (III)

Zur Definition des Typs „komplexe Zahlen“ könnte ein abstrakter Typ der vorangehenden Struktur vorgezogen werden. Ein abstrakter Typ würde die Implementierung der Werte verbergen und so unbeabsichtigte Verwendungen verhindern.

Die Mitteilung, die SML liefert, wenn die Struktur `Complex` deklariert wird, ist die Signatur dieser Struktur, die aus den Mitteilungen besteht, die gegeben würden, wenn die Deklarationen der Struktur einzeln definiert würden:

```
structure Complex :
  sig
    type t = real * real
    val difference : t * t -> t
    val product : t * t -> t
    val quotient : t * t -> t
    val reciprocal : t -> t
    val sum : t * t -> t
    val zero : t
  end
```

11.4.2. SML-Signaturen

Eine SML-Signatur ist eine abstrakte Beschreibung der Deklarationen einer oder mehrerer SML-Strukturen. Eine Signatur kann vom SML-System aus einer Strukturdeklaration ermittelt oder vom Programmierer selbst deklariert werden.

Beispiel: eine Signatur für Strukturen, die einen Typ `t` sowie die grundlegenden arithmetischen Operationen über `t` implementieren:

```
- signature ARITHMETIC =
   sig
    type t
    val zero       : t
    val sum        : t * t -> t
    val difference : t * t -> t
    val product    : t * t -> t
    val reciprocal : t -> t
    val quotient   : t * t -> t
  end;
```

Die Ausdrücke einer Signatur, die zwischen den reservierten Wörtern `sig` und `end` vorkommen, heißen *(Signatur-) Spezifikationen*.

SML-Signaturen (II)

Die Signatur `ARITHMETIC` kann in sogenannten *Signatur-Constraints* verwendet werden, wenn eine Struktur definiert wird, die alle in der Signatur spezifizierten Komponenten deklariert:

```
- structure Rational : ARITHMETIC =
   struct
    type t                                  = int * int;
    val zero                                = (0, 1) : t;
    fun sum       ((x1,y1):t, (x2,y2):t) = (x1*y2+x2*y1, y1*y2) :t;
    fun difference((x1,y1):t, (x2,y2):t) = (x1*y2-x2*y1, y1*y2) :t;
    fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2, y1 * y2) : t;
    fun reciprocal((x,y) : t)               = (y,x) : t
    fun quotient  (z1 : t,    z2 : t)  = product(z1, reciprocal z2)
   end;
structure Rational : ARITHMETIC
```

SML-Signaturen (III)

```
- structure Complex : ARITHMETIC =
   struct
    type t                                  = real * real;
    val zero                                = (0.0, 0.0) : t;
    fun sum       ((x1,y1):t, (x2,y2):t) = (x1 + x2, y1 + y2) : t;
    fun difference((x1,y1):t, (x2,y2):t) = (x1 - x2, y1 - y2) : t;
    fun product   ((x1,y1):t, (x2,y2):t) = (x1 * x2 - y1 * y2,
                                             x1 * y2 + x2 * y1) : t;
    fun reciprocal((x,y) : t)           = let val r = x * x + y * y
                                          in
                                              (x/r, ~y/r) : t
                                          end;
    fun quotient  (z1 : t,    z2 : t)   = product(z1, reciprocal z2)
   end;
structure Complex : ARITHMETIC
```

SML-Signaturen (IV)

- Ähnlich wie Typ-Constraints die statische Typprüfung ermöglichen, ermöglichen Signaturen das statische Zusammenfügen von Deklarationen aus unterschiedlichen Teilprogrammen, d.h. das Zusammenfügen von Deklarationen aus Teilprogrammen, ohne deren Implementierungen zu berücksichtigen.
- Die Typabkürzungen `Rational.t` und `Complex.t` bezeichnen unterschiedliche Typen, nämlich `int * int` und `real * real`, obwohl beide Strukturen dieselbe Signatur `ARITHMETIC` haben.
- Es ist üblich, dass die Namen von Strukturen mit einem Großbuchstaben beginnen und die Namen von Signaturen ganz aus Großbuchstaben bestehen.

  Dies ist eine Konvention, die aber nicht von SML erzwungen wird.

11.4.3. Spezifikation versus Deklaration in SML-Signaturen

Eine Signatur besteht aus Spezifikationen, nicht aus Deklarationen. Zum Beispiel ist

```
type t
```

eine Spezifikation.

Sie teilt mit, dass in der Signatur der Name `t` einen Typ bezeichnet. Diese Spezifikation teilt aber nicht mit, wie der Typ `t` definiert ist und welche Wertkonstruktoren der Typ `t` hat.

Um den Unterschied zwischen Spezifikationen und Deklarationen zu unterstreichen, darf das reservierte Wort `fun` in einer Signatur nicht vorkommen, sondern nur das reservierte Wort `val` wie etwa in:

```
val sum  : t * t -> t
```

11.4.4. `eqtype`–Spezifikationen in SML-Signaturen

In einer SML-Signatur kann die Spezifikation eines Typs `t` mit dem reservierten Wort `eqtype` statt `type` eingeführt werden, wenn die Gleichheit über `t` definiert sein muss.

Funktionstypen (also mit dem Typkonstruktor `->`) können natürlich nicht mit `eqtype` deklariert werden, da es keine Gleichheit über Funktionstypen gibt.

11.4.5. `datatype`–Spezifikationen in SML-Signaturen

In einer SML-Signatur kann die Spezifikation eines Typs `t` auch in Form einer `datatype`–Deklaration erfolgen.

In diesem Fall ist nicht nur spezifiziert, dass `t` ein Typ ist, sondern auch, welche Wertkonstruktoren der Typ `t` hat.

Ein Beispiel dafür kommt in der Signatur `LIST` vor (am Ende dieses Kapitels).

11.4.6. Angleich einer Struktur an eine Signatur — Struktursichten

Eine Struktur `Struk` kann an eine Signatur `Sig` angeglichen werden, wenn alle Komponenten, die in `Sig` spezifiziert werden, in `Struk` deklariert sind.

- Es ist aber möglich, eine Struktur an eine Signatur anzugleichen, die weniger Komponenten spezifiziert, als die Struktur deklariert.

  So können eingeschränkte Sichten (*views*) auf eine Struktur definiert werden.

- Die folgende Signatur spezifiziert z.B. nur einen Teil der Namen, die in der Struktur `Complex` deklariert sind:

```
- signature RESTRICTED_ARITHMETIC =
    sig
      type t
      val zero       : t
      val sum        : t * t -> t
      val difference : t * t -> t
    end;
```

Angleich einer Struktur an eine Signatur — Struktursichten (II)

Mit der Signatur `RESTRICTED_ARITHMETIC` kann eine Einschränkung der Struktur `Complex` definiert werden, die nur die in `RESTRICTED_ARITHMETIC` vorkommenden Namen zur Verfügung stellt.

```
- structure RestrictedComplex :  RESTRICTED_ARITHMETIC = Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC

- val i = (0.0, 1.0) : RestrictedComplex.t;
val i = (0.0,1.0) : Complex.t
- RestrictedComplex.sum(RestrictedComplex.zero, i);
val it = (0.0,1.0) : Complex.t
- RestrictedComplex.product(RestrictedComplex.zero, i);
Error: unbound variable or constructor: product in path
       RestrictedComplex.product
- Complex.product(RestrictedComplex.zero, i);
val it = (0.0,0.0) : Complex.t
```

In der Deklaration ist `: RESTRICTED_ARITHMETIC` ein sogenanntes *Signatur-Constraint*. Mit Signatur-Constraints kann man *information hiding* erreichen.

11.4.7. Parametrisierte Module in SML: SML-Funktoren

Die Strukturen `Rational` und `Complex` können um eine Funktion `square` erweitert werden, die eine rationale bzw. komplexe Zahl quadriert:

```
fun square z = product(z, z)
```

- Ergänzen wir `Rational` und `Complex` um diese Deklaration, so sind die Funktionen `Rational.square` und `Complex.square` doch unterschiedlich definiert, weil sie sich auf unterschiedliche Definitionen der Funktion `product` beziehen.
- `Rational.square` und `Complex.square` können identisch deklariert werden, auch wenn sie unterschiedlich definiert sind.
- Jeder Zahlentyp, der über ein Produkt verfügt, ermöglicht dieselbe Definition des Quadrats im Bezug auf dieses Produkt.
- Die Erweiterung beider Strukturen `Rational` und `Complex` um eine Funktion `square` wäre nachteilhaft:

  Die Identität der `square`–Deklarationen würde verborgen und so etwaige Veränderungen dieser Funktionen erschwert werden.

Parametrisierte Module in SML: SML-Funktoren (II)

Ein SML-Funktor ist eine SML-Struktur, die andere SML-Strukturen als Parameter erhält, also eine sogenannte *parametrisierte Struktur*.

So kann z.B. die Funktion `square` für alle Strukturen mit der Signatur `ARITHMETIC` wie folgt definiert werden:

```
- functor Extended(S : ARITHMETIC) =
      struct
         fun square z = S.product(z, z)
      end;
functor Extended : <sig>
```

Der Funktor `Extended` stellt erst dann eine benutzbare Struktur dar, wenn der Name `S` an eine Struktur mit Signatur `ARITHMETIC` (also z.B. `Rational` oder `Complex`) gebunden wird.

Parametrisierte Module in SML: SML-Funktoren (III)

Dies geschieht etwa wie folgt:

```
- structure ExtComplex = Extended(Complex);
structure ExtComplex : sig val square : S.t -> S.t end

- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- ExtComplex.square(i);
val it = (~1.0,0.0) : Complex.t

- structure ExtRational = Extended(Rational);
structure ExtRational : sig val square : S.t -> S.t end

- val minusone = (~1, 1) : Rational.t;
val minusone = (~1,1) : Rational.t

- ExtRational.square(minusone);
val it = (1,1) : Rational.t
```

Parametrisierte Module in SML: SML-Funktoren (IV)

Die Quadratfunktionen in `ExtComplex` und `ExtRational` sind verschiedene Funktionen, auch wenn ihre Deklaration im Funktor `Extended` syntaktisch identisch ist:

```
- ExtComplex.square(0.0, 1.0);
val it = (~1.0,0.0) : Complex.t

- ExtRational.square(0, 1);
val it = (0,1) : Rational.t
```

- Die (statische) Typprüfung verwendet für einen Funktor ausschließlich die Information, die die Signatur liefert. Damit kann zur Laufzeit ein Funktor auf unterschiedliche Strukturen angewendet werden.
- Ein typkorrekter Funktor kann auf jede Struktur angewendet werden, die mit der Signatur des Parameters des Funktors angeglichen werden kann.

Man könnte auch die Signatur `ARITHMETIC` ohne die Funktion `quotient` definieren und diese Funktion durch einen Funktor auf `product` und `reciprocal` zurückführen.

11.4.8. Generative und nichtgenerative Strukturdeklarationen

Zwei Formen von Strukturdeklarationen sind also möglich:

1. *generative Strukturdeklaration*, z.B.:

   ```
   structure RestrictedComplex  =
    struct type t                            = real * real;
           val zero                          = (0.0, 0.0) : t;
           fun sum        ((x1,y1),(x2,y2)) = (x1 + x2, y1 + y2) : t;
           fun difference((x1,y1),(x2,y2)) = (x1 - x2, y1 - y2) : t;
    end;
   ```

   Die Bezeichnung „generativ“ unterstreicht, dass die Strukturdeklaration ihre Komponenten explizit deklariert.

2. *nichtgenerativen Strukturdeklaration*, z.B.:
   (unter Verwendung bereits deklarierter Strukturen und/oder Signaturen)

   ```
   structure RestrictedComplex :  RESTRICTED_ARITHMETIC = Complex;
   ```

   In einer solchen Strukturdeklaration werden die Komponenten der Struktur nicht explizit, sondern implizit in Bezug auf bestehende Deklarationen definiert.

11.4.9. Weiteres über Module in SML

Geschachtelte Strukturen sind möglich, z.B. in:

```
- structure Nb = struct
                    structure ComplexNb  = Complex;
                    structure RationalNb = Rational
                 end;
structure Nb :
  sig
    structure ComplexNb : <sig>
    structure RationalNb : <sig>
  end

- val i = (0.0, 1.0) : Nb.ComplexNb.t;
val i = (0.0,1.0) : Complex.t
- Nb.ComplexNb.product(i, Nb.ComplexNb.zero);
val it = (0.0,0.0) : Complex.t
- val minusone = (~1, 1) : Nb.RationalNb.t;
val minusone = (~1,1) : Rational.t
- Nb.RationalNb.sum(minusone, Nb.RationalNb.zero);
val it = (~1,1) : Rational.t
```

Weiteres über Module in SML (II)

Lange Namen

Namen wie `Complex.t`, `Complex.product` und `Nb.RationalNb.sum` werden *lange Namen* (oder „lange Bezeichner“) genannt.

Da geschachtelte Strukturen möglich sind, können lange Namen aus mehr als zwei (nichtlangen) Namen bestehen, wie etwa `Nb.RationalNb.sum`.

Teilen (*sharing*) von Deklarationen in Strukturen

In einer Struktur– oder Signaturdeklaration kann verlangt werden, dass Namen aus unterschiedichen Strukturen oder Signaturen „geteilt“ werden, d.h. als identisch gelten (siehe dazu die Literatur über SML).

11.5. Hinweis auf die Standardbibliothek von SML

Die Standardbibliothek von SML besteht aus Modulen.

In ihr kommen Signaturdeklarationen und Spezifikationen vor. Ein Beispiel stellen die Spezifikationen der Struktur `List` dar.

Kennt man den Namen einer Struktur wie `List`, kann man im SML-System die zugehörige Signatur herausfinden, indem man einen neuen (irrelevanten) Strukturnamen deklariert:

```
- structure Struct = List;
structure Struct : LIST
```

Die Mitteilung des SML-Systems enthält also die Information, dass die Struktur `List` die Signatur `LIST` hat.

Hinweis auf die Standardbibliothek von SML (II)

Deklariert man dafür wiederum einen neuen (irrelevanten) Signaturnamen, erfährt man aus der Mitteilung des SML-Systems die Schnittstelle der Struktur:

```
- signature SIG = LIST;
signature LIST =
  sig
    datatype 'a list = :: of 'a * 'a list | nil
    exception Empty
    val null : 'a list -> bool
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val last : 'a list -> 'a
    val getItem : 'a list -> ('a * 'a list) option
    val nth : 'a list * int -> 'a
    val take : 'a list * int -> 'a list
    val drop : 'a list * int -> 'a list
    val length : 'a list -> int
    val rev : 'a list -> 'a list
    val @ : 'a list * 'a list -> 'a list
    val concat : 'a list list -> 'a list
```

Hinweis auf die Standardbibliothek von SML (III)

```
  val revAppend : 'a list * 'a list -> 'a list
  val app : ('a -> unit) -> 'a list -> unit
  val map : ('a -> 'b) -> 'a list -> 'b list
  val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
  val find : ('a -> bool) -> 'a list -> 'a option
  val filter : ('a -> bool) -> 'a list -> 'a list
  val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val exists : ('a -> bool) -> 'a list -> bool
  val all : ('a -> bool) -> 'a list -> bool
  val tabulate : int * (int -> 'a) -> 'a list
end
```

Mit einer gewissen Erfahrung in der Listenverarbeitung reicht diese Information oft schon aus, um die geeignete Funktion für das gerade bearbeitete Problem zu finden.

12. Imperative Programmierung in SML

- 12.1 SML-Referenzen
- 12.2 Iteration in SML
- 12.3 SML-Felder
- 12.4 Beispiel: Sortieren eines Feldes durch direktes Einfügen
  (*straight insertion sort*)
- 12.5 Ein- und Ausgaben in SML

12.1. SML-Referenzen

- Die Zustandvariablen in SML heißen *Referenzen*.

- Die SML-Referenzen sind *Zustandsvariablen mit expliziter Dereferenzierung*.

12.1.1. Deklaration einer Referenz — Referenzierungsoperator in SML

Eine Referenz wird in SML unter Verwendung der Funktion `ref` deklariert:

```
- val kto = ref 100;
val kto = ref 100 : int ref
```

Es gibt in SML keine Möglichkeit eine Referenz zu deklarieren, ohne dabei der deklarierten Referenz einen Initialwert zuzuweisen.

Der Referenzierungsoperator in SML ist `ref`:

```
- val kto = ref 100;
val kto = ref 100 : int ref

- val v = ref kto;
val v = ref (ref 100) : int ref ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

12.1.2. Typ einer Referenz

Für jeden Monotyp `t` kann eine Zustandsvariable vereinbart werden, deren Inhalt ein Wert von Typ `t` ist.

Die folgenden Deklarationen definieren die Referenzen `r` und `w` als Zustandsvariablen, deren Werte Funktionen vom Typ `int -> int` sind:

```
- val r = ref (fn x => 2 * x);
val r = ref fn : (int -> int) ref

- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref
```

Es ist in SML nicht möglich, eine Zustandsvariable zu vereinbaren, deren Inhalt ein Wert eines Polytyps ist:

```
- val v = ref [];
stdIn:8.1-8.15 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val v = ref [] : ?.X1 list ref
```

Typ einer Referenz (II)

Legt ein Typ-Constraint einen Typ für [] fest, so kann eine Zustandsvariable mit initialem Wert [] vereinbart werden, wie das folgende Beispiel zeigt:

```
- val v = ref ([] : string list);
val v = ref [] : string list ref
```

Referenzen, deren Inhalte Referenzen sind, sind möglich:

```
-  val w = ref (fn x:int => 0);
val w = ref fn : (int -> int) ref

- val z = ref w;
val z = ref (ref fn) : (int -> int) ref ref

- !(!z)(9);
val it = 0 : int
```

12.1.3. Dereferenzierungsoperator in SML

Der Dereferenzierungsoperator in SML ist „!”:

```
- val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- !w(!v);
val it = 0 : int
```

Der Dereferenzierungsoperator verhält sich, als ob er wie folgt (unter Verwendung von Pattern Matching) definiert wäre:

```
fun ! (ref x) = x
```

12.1.4. Sequenzierungsoperator in SML

Der Wert einer Sequenz von Ausdrücken ($\mathtt{A}_1$; $\mathtt{A}_2$; ...; $\mathtt{A}_n$) ist der Wert des letzten Ausdrucks der Sequenz, also von $\mathtt{A}_n$:

```
- 1; 2; 3;
val it = 1 : int
val it = 2 : int
val it = 3 : int

- val x = (1; 2; 3);
val x = 3 : int
```

Man beachte den Unterschied zum folgenden Beispiel:

```
- val x = 1; 2; 3;
val x = 1 : int
val it = 2 : int
val it = 3 : int
```

Im ersten Beispiel ist die Sequenz der definierende Ausdruck der Wertdeklaration. Im zweiten Beispiel ist die Wertdeklaration der erste Ausdruck der Sequenz.

12.1.5. Zuweisungsoperator in SML

Der Wert einer Zuweisung in SML ist () (*unity*):

```
- val r = ref 1;
val r = ref 1 : int ref
- r := !r + 2;
val it = () : unit
- !r;
val it = 3 : int
```

Da in SML Referenzen Zustandsvariablen mit expliziter Dereferenzierung sind, kann das vorangehende Beispiel nicht wie folgt fortgesetzt werden:

```
- r := r + 1;
Error: operator and operand don't agree [literal]
  operator domain: int ref * int ref
  operand:         int ref * int
  in expression:
    r + 1
Error: overloaded variable not defined at type
  symbol: +
  type: int ref
```

Zuweisungsoperator in SML (II)

Der Inhalt der Referenz `r` kann aber wie folgt um 1 erhöht werden:

```
- r := !r + 1;
val it = () : unit

 - !r;
 val it = 4 : int
```

Da in SML der Wert `()` — vom Typ Typ `unit` — als Wert einer Zuweisung geliefert wird, hat der Zuweisungsoperator „`:=`“ den folgenden Polytyp:

```
'a ref * 'a -> unit.
```

12.1.6. Druckverhalten von SML bei Referenzen

SML liefert Referenzen, druckt sie aber nicht aus. Anstelle einer Referenz druckt SML das Symbol `ref` gefolgt von ihrem Inhalt:

```
- val v = ref 5;
val v = ref 5 : int ref

- v;
val it = ref 5 : int ref

-  val w = ref (fn(x:int) => 0);
val w = ref fn : (int -> int) ref

- w;
val it = ref fn : (int -> int) ref
```

Dieses Druckverhalten von SML sollte nicht dazu verleiten, die Referenzen in SML für Zustandsvariablen ohne explizite Dereferenzierung zu halten!

12.1.7. Gleichheit zwischen Referenzen

Die Gleichheit für Referenzen in SML ist „=“.

```
- val v1 = ref 12;
val v1 = ref 12 : int ref

- val v2 = v1;
val v2 = ref 12 : int ref

- v1 = v2;
val it = true : bool
```

Sind zwei Referenzen gleich, so sind ihre Inhalte ebenfalls gleich:

```
- !v1 = !v2;
val it = true : bool
```

Zwei unterschiedliche Referenzen mit demselben Inhalt sind aber nicht gleich:

```
- ref 12 = ref 12;
val it = false : bool
```

Gleichheit zwischen Referenzen (II)

Dieses Verhalten kann durch wie folgt verdeutlicht werden:

```
- val v1 = ref 12;
val v1 = ref 12 : int ref

- val v2 = ref 12;
val v2 = ref 12 : int ref

- v1 = v2;
val it = false : bool
```

Das Ergebnis der Auswertung des Ausdruckes `ref 12 = ref 12` zeigt, dass `ref` nicht referenztransparent, also nicht rein funktional ist.

12.1.8. Vordefinierte SML-Prozeduren über Referenzen

- Die Dereferenzierung, die Zuweisung und die Gleichheit sind die einzigen vordefinierten Prozeduren in SML über Referenzen.

- Im Gegensatz zu anderen Programmiersprachen ermöglicht SML nicht, zu einer Referenz, d.h. einer Speicheradresse, eine Zahl zu addieren, um so auf eine darauffolgende Speicherzelle zuzugreifen.

  Diese Möglichkeit, die bei Maschinensprache sinnvoll ist, ist bei modernen höheren Programmiersprachen „verpönt“, weil sie zu schwer verständlichen und daher häufig fehlerhaften Programmen führt.

12.2. Iteration in SML

Zur Iteration bietet SML die While-Schleife an; z.B.:

```
- val zaehler = ref 4;
- while ! zaehler > 0 do
      ( print(Int.toString(!zaehler) ^ " ") ;
        zaehler := !zaehler - 1  ) ;
4 3 2 1 val it = () : unit
```

Ein `while`–Ausdruck in SML hat die folgende Gestalt:

```
while A1 do A2
```

- Der Ausdruck `A1` (vom Typ `bool`) ist die „Bedingung“ des `while`–Ausdrucks.
- Der Ausdruck `A2` (mit beliebigem Typ) wird *Rumpf* des `while`–Ausdrucks genannt.
- Der Typ eines `while`–Ausdrucks ist `unit`.

Man beachte die Verwendung der Klammern und der Sequenz im Rumpf des `while`–Ausdrucks des vorangehenden Beispiels.

Iteration in SML (II)

Die Fakultätsfunktion kann in SML statt mittels Endrekursion auch iterativ implementiert werden:

```
- fun fak n =
      let val zaehler = ref n;
          val akk     = ref 1;
      in
        while !zaehler <> 0 do
            (akk := !akk * !zaehler; zaehler := !zaehler - 1);
        ! akk
      end;
val fak = fn : int -> int

- fak 4;
val it = 24 : int
```

Der Ausdruck `! akk` unmittelbar nach dem `while`–Ausdruck dient lediglich dazu, den berechneten Wert als Wert der Anwendung der Prozedur `fak` zu liefern.
Dieser Ausdruck `! akk` ist nicht Teil des Rumpfes des `while`–Ausdrucks.

12.3. SML-Felder

- Ein Feld (*array*) ist einem Vektor ähnlich.
- Im Gegensatz zu den Komponenten eines Vektors sind jedoch die Komponenten eines Feldes veränderbar.
- Felder nehmen in der imperativen Programmierung eine ähnliche Stellung wie Listen in der rein funktionalen Programmierung ein.
- Die Struktur `Array` der Standardbibliothek von SML bietet Prozeduren an, mit denen Felder deklariert und verwendet werden können.

12.3.1. Deklaration eines Feldes

Felder können wie folgt deklariert werden:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array
```

Bei der Deklaration eines Feldes wird also ein (gleicher) Initialwert allen Komponenten des Feldes zugewiesen.

Felder können Komponenten von unterschiedlichen Typen haben, jedoch wie bei Listen müssen alle Komponenten eines Feldes denselben Typ haben:

```
- val int_feld = Array.array(5, 12);
val int_feld = [|12,12,12,12,12|] : int array

- val int_list_feld = Array.array(2, [1,2,3]);
val int_list_feld = [|[1,2,3],[1,2,3]|] : int list array
```

12.3.2. Zugriff auf die Komponenten eines Feldes

Auf die Komponenten eines Feldes kann direkt zugegriffen werden:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.sub(f, 0);
val it = "a" : string

- Array.sub(f, 3);
uncaught exception subscript out of bounds
raised at: stdIn:47.1-47.10
```

Die erste Komponente eines Feldes mit 3 Komponenten ist die Komponente mit Nummer 0, die letzte die Komponente mit Nummer 2.

12.3.3. Veränderung der Komponenten eines Feldes

Die Komponenten eines Feldes können verändert werden:

```
- val f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.update(f, 1, "b");
val it = () : unit
- Array.sub(f, 1);
val it = "b" : string
- Array.update(f, 2, "c");
val it = () : unit

- val zaehler = ref 0;
val zaehler = ref 0 : int ref
- while !zaehler < 3 do
      ( print(Array.sub(f, !zaehler) ^ " ");
        zaehler := !zaehler + 1 );
a b c val it = () : unit
```

12.3.4. Länge eines Feldes

Die Funktion `Array.length` liefert die Länge eines Feldes:

```
- val  f = Array.array(3, "a");
val f = [|"a","a","a"|] : string array

- Array.length f;
val it = 3 : int
```

12.3.5. Umwandlung einer Liste in ein Feld

Es ist möglich aus einer Liste ein Feld zu erzeugen:

```
- Array.fromList [1, 2, 3];
val it = [|1,2,3|] : int array
```

12.3.6. Umwandlung eines Feldes in eine Liste

Aus einem Feld kann auch eine Liste erzeugt werden:

```
- fun array_to_list(a : 'a array) =
      let val length     = Array.length a;
          val counter_rf = ref length;
          val list_rf    = ref [] : 'a list ref
      in
        while ! counter_rf > 0 do
           ( counter_rf := ! counter_rf - 1
           ; list_rf := Array.sub(a, ! counter_rf) :: ! list_rf
           );
        ! list_rf
      end;
val array_to_list = fn : 'a array -> 'a list

- array_to_list(f);
val it = ["a","b","c"] : string list
```

12.3.7. Gleichheit für Felder

Die Gleichheit zwischen Feldern ist referenzbezogen. Folglich liefert die Gleichung

```
- Array.fromList [1, 2, 3] = Array.fromList [1, 2, 3];
val it = false : bool
```

genauso wie die Gleichung

```
- ref 12 = ref 12;
val it = false : bool
```

den Wert `false`.

12.4. Beispiel: Sortieren eines Feldes durch direktes Einfügen (straight insertion sort)

- totale Ordnung
- Sortieren
- internes Sortieren durch direktes Einfügen (*straight insertion sort*)
- Komplexität des internen Sortierens durch direktes Einfügen

12.4.1. Totale Ordnung

Eine Ordnung $<$ über eine Menge M ist eine binäre Relation über M, d.h. eine Teilmenge des Kartesischen Produkts $M \times M$, die die folgenden Eigenschaften erfüllt:

- $<$ ist *reflexiv*:
  für alle $m \in M$ gilt $x < x$
- $<$ ist *antisymmetrisch*:
  für alle $m_1, m_2 \in M$, falls $m_1 < m_2$ und $m_2 < m_1$, dann $m_1 = m_2$
- $<$ ist *transitiv*:
  für alle $m_1, m_2, m_3 \in M$, falls $m_1 < m_2$ und $m_2 < m_3$, dann $m_1 < m_3$

Eine Ordnung $<$ über einer Menge M heißt *total*, wenn für alle $m_1 \in M$ und $m_2 \in M$, $m_1 < m_2$ oder $m_2 < m_1$.

12.4.2. Sortieren

- Unter dem Sortieren von Elementen aus einer Menge M mit (totaler) Ordnung $<$ versteht man das Anordnen in auf- oder absteigender Reihenfolge bezüglich $<$.
- Sortieren dient i.Allg. dazu, die Suche nach einem Element in der gegebenen Menge zu vereinfachen.
- Man unterscheidet zwischen dem Sortieren von Felder (*internes Sortieren*) und das Sortierten im Dateien (*externes Sortieren*).
- Internes Sortieren wird verwendet, um Datensätze im Hauptspeicher zu sortieren. Dabei setzt man oft voraus, dass kein oder nur sehr wenig weiterer Speicherplatz als das zu sortierende Feld zur Verfügung steht.
- Externes Sortieren geht auf die Zeit zurück, wo große Datenmengen auf Bändern gespeichert wurden, worauf ausschließlich sequenziell, also ohne sogenannte direkte Zugriffe wie auf Felder, geschrieben wurde.

  Um den Inhalt eines Bandes zu sortieren, verwenden externe Sortierverfahren mindestens ein weiteres Band, auf dem Zwischenergebnisse aufgenommen werden.

Sortieren (II)

- Datensätze bestehen i.d.R. aus einem sogenannten *Schlüssel* sowie weiteren Werten.
- Der *Schlüssel* ist ein ggf. künstlich hinzugefügter Teil des Datensatzes, über dessen Typ eine (totale) Ordnung definiert ist.

Der Einfachheit halber betrachten wir im Folgenden Schlüssel, die ganze Zahlen sind, die totale Ordnung $\leq$ über den ganzen Zahlen, und Datensätze, die nur aus einem Schlüssel bestehen.

12.4.3. Internes Sortieren durch direktes Einfügen (straight insertion sort)

Diese Sortiermethode wird oft von Kartenspieler verwendet. Die Datensätze (etwa Karten) werden in eine Zielsequenz `a(1), ..., a(i-1)` und eine Quellsequenz `a(i), ..., a(n)` aufgeteilt:

$$\underbrace{\mathtt{a}(1)\ \ldots\ \mathtt{a}(\mathtt{i}-1)}_{\text{Zielsequenz}}\quad \underbrace{\mathtt{a}(\mathtt{i})\ \ldots\ \mathtt{a}(\mathtt{n})}_{\text{Quellsequenz}}$$

Anfangs enthält die Zielsequenz nur `a(1)`, die Quellsequenz `a(2)` bis `a(n)`:

$$\underbrace{\mathtt{a}(1)}_{\text{Zielsequenz}}\quad \underbrace{\mathtt{a}(2)\ \ldots\ \mathtt{a}(\mathtt{n})}_{\text{Quellsequenz}}$$

Wir nehmen an, dass nach aufsteigenden Schlüsseln sortiert werden soll.

Internes Sortieren durch direktes Einfügen (II)

Beginnend mit $\mathtt{i} = 2$ wird der erste Datensatz `a(i)` der Quellsequenz in die Zielsequenz eingefügt, dass sich eine um einen Datensatz längere sortierte Zielsequenz ergibt.

Dafür müssen alle Elemente der Zielsequenz, die echt größer als `a(i)` sind, um eine Stelle nach rechts verschoben werden:

$$\texttt{vorher:}\quad \underbrace{\overset{\mathtt{a}(\mathtt{1})}{2}\quad 5\quad 12\quad 15\quad 22\quad 35}_{\text{Zielsequenz}}\quad \underbrace{\overset{\mathtt{a}(\mathtt{i})}{14}\quad 20\quad \overset{\mathtt{a}(\mathtt{n})}{30}}_{\text{Quellsequenz}}$$

$$\text{nachher}:\quad \underbrace{\overset{\mathtt{a}(\mathtt{1})}{2}\quad 5\quad 12\quad 14\quad 15\quad 22\quad 35}_{\text{Zielsequenz}}\quad \underbrace{\overset{\mathtt{a}(\mathtt{i}+\mathtt{1})}{20}\quad \overset{\mathtt{a}(\mathtt{n})}{30}}_{\text{Quellsequenz}}$$

Internes Sortieren durch direktes Einfügen (III)

Für jeden Wert von `i` zwischen `2` und `n` wird `a(i)` wie folgt in die Zielsequenz eingefügt:

- Angefangen mit $\mathtt{k} = \mathtt{i} - 1$ bis $\mathtt{k} = 1$ wird `a(i)` mit `a(k)` verglichen.

  Gilt `a(k) > a(i)`, so werden die Werte von `a(k)` und `a(i)` vertauscht.

- Um das Vertauschen zu ermöglichen, muss einer der Werte von `a(k)` und `a(i)` in einer Zustandsvariable `z` aufgenommen werden.

  Da `a(i)` in der Regel mit mehreren Feldkomponenten `a(k)` verglichen wird, empfehlt es sich, den Wert von `a(i)` „zwischenzuspeichern“.

Internes Sortieren durch direktes Einfügen (IV)

Es ergibt sich ein Algorithmus, der in einem nicht weiter formalisierten *Pseudo-Code* (mit Zustandsvariablen mit expliziter Dereferenzierung) ausgedruckt ist:

```
i := 2;
while ! i <= n do
   z := ! a(!i);
   k := ! i;
   while ! k >= 1 and ! z < ! a(!k-1) do
          a(!k) := ! a(!k-1);
          k := ! k - 1
   end-while;
   a(!k) := ! z;
   i := ! i + 1
end-while
```

Die Abbruchbedingung `! k >= 1 and ! z < ! a(!k-1)` der inneren `while`–Schleife kann in `! a(!i) < ! a(!k-1)` vereinfacht werden, wenn anstelle der Zustandsvariable `z` eine Feldkomponente `a(0)` verwendet wird. Ist der Wert von `a(0)` derselbe wie von `a(!i)`, so ist die neue Bedingung verletzt, wenn $\texttt{k} = 1$.

Internes Sortieren durch direktes Einfügen (V)

So ergibt sich der folgende (verbesserte) Algorithmus:

```
i := 2;
while !i <= n do
   a(0) := ! a(!i);
   k := ! i;
   while ! a(!i) < ! a(!k-1) do
          a(!k) := ! a(!k-1);
          k := ! k - 1
   end-while;
   a(!k) := ! a(0);
   i := ! i + 1
end-while
```

Dieser Algorithmus kann wie folgt in SML implementiert werden, wobei die zu sortierende Sequenz als Liste ein- und ausgegeben wird. Als Initialwerte der Feldkomponente `a(i)` und des Zählers `k` wird willkürlich 0 gewählt:

Internes Sortieren durch direktes Einfügen (VI)

```
- fun direktes_einfuegen(sequenz : int list) : int list =
         let val a = Array.fromList(0 :: sequenz);
             val i = ref 2;
             val k = ref 0;
             val n = Array.length a - 1;
         in
           while ! i <= n do
              ( Array.update(a, 0, Array.sub(a, ! i));
                Array.update(a, 0, Array.sub(a, 0));
                k := ! i;
                while Array.sub(a, 0) < Array.sub(a, ! k - 1) do
                   ( Array.update(a, ! k,  Array.sub(a, ! k - 1));
                     k := ! k - 1  );
                Array.update(a, ! k, Array.sub(a, 0));
                i := ! i + 1  );
           tl(array_to_list a)
         end;
val direktes_einfuegen = fn : int list -> int list
```

12.4.4. Komplexität des internen Sortierens durch direktes Einfügen

- Während des `i`-ten Durchlaufs der äußeren While-Schleife werden mindestens `1` und höchstens $\mathtt{i}-1$ Schlüssel verglichen.

  Sind alle Permutationen von `n` Schlüsseln in der sortierenden Sequenz gleich wahrscheinlich, so werden im Durchschnitt während des `i`-ten Durchlaufs $\mathtt{V} = \mathtt{i}/2$ Schlüssel verglichen.

- Die Anzahl der Feldkomponenten, die während des `i`-ten Durchlaufs einen neuen Wert erhalten, beträgt $\mathtt{V}+2$, weil am Anfang des Durchlaufs `a(0)` den Wert von `a(i)` erhält.

Gesamtzahlen der Schlüsselvergleiche und Aktualisierungen von Feldkomponenten:

	Mindestzahl	Durchschnitt	Höchstzahl
Schlüsselvergleiche	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2$
Aktualisierungen	$3(n-1)$	$(n^2+9n-10)/4$	$(n^2+3n-3)/2$

Komplexität des internen Sortierens durch direktes Einfügen (II)

- Die Mindestzahlen kommen vor, wenn die zu sortierende Sequenz bereits sortiert ist, die Höchstzahlen, wenn die zu sortierende Sequenz in umgekehrter Reihenfolge angeordnet ist.
- Wird als Größe des Problems die Länge der zu sortierenden Sequenz angesehen, und als Zeiteinheit ein Vergleich oder eine Aktualisierung, so ist der durchschnittliche Zeitbedarf des Sortierens durch direktes Einfügen $O(n^2)$.

Das Sortieren durch direktes Einfügen ist kein guter Sortieralgorithmus.

12.5. Ein- und Ausgaben in SML

- Datenströme (streams)
- Ausgabestrom
- Eingabestrom
- Standard–Ein- und -Ausgabetextströme `TextIO.stdIn` und `TextIO.stdOut`
- Die vordefinierte Prozedur `print`
- Beispiel: Inhalt einer Datei einlesen und an die Standardausgabe weiterleiten

12.5.1. Datenströme (*streams*)

- Die Ein- und Ausgabe wird in SML (wie in anderen Programmiersprachen) mit dem Paradigma des „Datenstroms“ (*stream*) beschrieben.
- Ein Datenstrom stellt in SML eine Folge von Datensätzen vom Typ Zeichen, Zeichenfolgen oder auch Binärzahl dar, deren Länge unbegrenzt ist.
- Die Datensätze eines Datenstroms werden nacheinander zwischen dem „Datenerzeuger“ (*producer*) und dem „Datenverbraucher“ (*consumer*) verschickt.
- Man unterscheidet zwischen „Eingabedatenströme“ (*input streams*) und „Ausgabedatenströme“ (*output streams*).
- Der *Datenerzeuger eines Eingabedatenstroms* kann bei einer interaktiven SML-Sitzung die Tastatur des Terminals sein.

  Es kann aber auch eine Datei sein, deren Inhalt sequenziell eingelesen wird.

  Der *Datenverbraucher eines Eingabedatenstroms* ist das Programm, in dem dieser Eingabedatenstrom deklariert ist.

Datenströme (*streams*) (II)

- Der *Datenerzeuger eines Ausgabedatenstroms* ist das Programm, in dem dieser Ausgabedatenstroms deklariert ist.

  Der *Datenverbraucher eines Ausgabedatenstroms* kann bei einer interaktiven SML-Sitzung der Bildschirm sein.

  Der Datenverbraucher kann auch ein Drucker oder eine Datei sein, in die die Datensätze des Ausgabedatenstroms geschrieben werden.

- Ein Datenstrom wird in SML durch eine Deklaration erzeugt.

  Dabei erhält ein Datenstrom einen Namen (Bezeichner).

  Die Deklaration eines Eingabedatenstroms spezifiziert zudem einen Datenerzeuger, die Deklaration eines Ausgabedatenstroms einen Datenverbraucher.

Datenströme (*streams*) (III)

- Mit der Deklaration eines Eingabe- oder Ausgabedatenstroms wird auf Ressourcen des zugrundeliegenden Betriebssystems zugegriffen.

  Man sagt, dass der Datenstrom „geöffnet“ wird.
- Damit sparsam mit Betriebssystemressourcen umgegangen werden kann, können (und sollen!) Datenströme, die nicht mehr benötigt werden, geschlossen werden.
- In SML stellen Datenströme Typen dar, die über vordefinierte Prozeduren zum Erzeugen und Schließen sowie zum Steuern der Datenübertragung eines Datenstroms verfügen.
- Die vordefinierten Datenstromprozeduren sind selbstverständlich nicht referenztransparent, weil der Aufruf zu unterschiedlichen Zeitpunkten desselben Ausdrucks zur Steuerung eines Datenstroms unterschiedliche Folgen haben mag.

  Die Datenstromprozeduren sind folglich nicht rein funktional, also im strengen Sinne keine Funktionen.

Datenströme (*streams*) (IV)

- SML bietet Datenströme von den zwei folgenden Arten an:
  - Datenströme von Zeichen (SML-Typ `char`) oder Zeichenfolgen (SML-Typ `string`) — auch „Textströme“ genannt;
  - Datenströme von Binärdaten dargestellt als Worte einer Länge von 8 Bits ($= 1$ Byte) — auch „Binärströme“ genannt.
- Die Datenstromprozeduren für beide Datenstromarten sind in der SML-Standardbibliothek definiert.
- Das Modul (oder Struktur) `TextIO` der SML-Standardbibliothek enthält die Prozeduren zur Verwendung von Textströmen.

  Das Modul `BinIO` der SML-Standardbibliothek enthält die Prozeduren zur Verwendung von Binärströmen.

Im Folgenden werden einige Prozeduren zur Verwendung von Textströmen eingeführt.

12.5.2. Ausgabestrom

Ein Ausgabetextstrom namens `strom1` mit einer Datei `datei1` als Datenverbraucher kann wie folgt deklariert (oder geöffnet) werden:

```
- val strom1 = TextIO.openOut "datei1";
val strom1 = - : TextIO.outstream
```

Die Mitteilung zeigt, dass der Typ eines Ausgabetextstroms `TextIO.outstream` ist.

- Existiert die Datei `datei1` noch nicht, so wird sie mit dieser Deklaration erzeugt.
- Ist die Datei `datei1` aber schon vorhanden, wenn die vorangehende Deklaration ausgewertet wird, geht ihr bisheriger Inhalt verloren.

Der Ausgabetextstrom `strom1` kann wie folgt verwendet werden, um in die Datei `datei1` die Zeichenfolge „Erster Schreibtest“ zu schreiben:

```
- TextIO.output(strom1, "Erster Schreibtest");
val it = () : unit
```

Ausgabestrom (II)

- Ausgabeströme übergeben ihre Daten an ihren Datenverbraucher nicht immer sofort.
- Verzögerungen können auftreten, die durch die Verwaltung der auf dem Prozessor gleichzeitig laufenden Prozesse durch das Betriebssystem bedingt sind.

  Die Prozedur `TextIO.flushOut` bewirkt, dass alle vom Ausgabestrom noch nicht ausgegebenen Datensätze an den Verbraucher des Ausgabestroms weitergeleitet werden:

  ```
  - TextIO.flushOut strom1;
  val it = () : unit
  ```

Der Ausgabetextstrom `strom1` kann wie folgt geschlossen werden:

```
- TextIO.closeOut strom1;
val it = () : unit
```

12.5.3. Eingabestrom

Sei `datei2` eine Datei, deren Inhalt die folgende Zeichenfolge ist:

```
"Erster Lesetext\n"
```

Eine Eingabetextstrom namens `strom2` (vom Typ `TextIO.instream`) mit einer Datei `datei2` als Datenerzeuger kann wie folgt deklariert (oder geöffnet) werden:

```
- val strom2 = TextIO.openIn "datei2";
val strom2 = - : TextIO.instream
```

Der Eingabetextstrom `strom2` wird wie folgt verwendet, um das erste Zeichen des Stromes (also aus der Datei `datei2`) zu erhalten:

```
- val zeichen = TextIO.input1(strom2);
val zeichen = SOME #"E" : TextIO.elem option
```

Eingabestrom (II)

Das Ergebnis ist ein Wert des zusammengesetzten Typs `TextIO.elem option`:

- Der Polytyp `option` besitzt zwei Wertkonstruktoren: `SOME` und `NONE`.

  `option` erlaubt jeden Typ `'a` zur Unterscheidung zwischen zulässigen und unzulässigen Werten zu erweitern.

- `elem` ist der abstrakte Typ der Datensätze eines Eingabe- oder Ausgabetextstroms.

Eingabestrom (III)

Eine beliebige Anzahl n (hier: $n = 6$) von Datensätzen aus dem Eingabetextstrom `strom2` kann wie folgt erhalten werden:

```
- val zeichenfolge = TextIO.inputN(strom2, 6);
val zeichenfolge = "rster " : TextIO.vector
```

Da das erste Zeichen „`E`“ bereits „konsumiert“ wurde, werden die folgenden 6 Zeichen geliefert werden. Die gelieferte Zeichenfolge hat den Typ `TextIO.vector`, der nicht mit den herkömmlichen SML-Vektoren verwechselt werden darf.

Die übrigen Datensätze des Eingabetextstroms `strom2` können erhalten werden durch:

```
- val rest = TextIO.input(strom2);
val rest = "Lesetest\n" : TextIO.vector
```

Der Eingabetextstrom `strom2` wird am Ende wie folgt geschlossen:

```
- TextIO.closeIn strom2;
val it = () : unit
```

12.5.4. Standard–Ein- und -Ausgabetextströme `TextIO.stdIn` und `TextIO.stdOut`

Die sogenannten Standard-Ein- und -Ausgaben sind in SML Textströme. Sie heißen `TextIO.stdIn` und `TextIO.stdOut`.

- So kann die Zeichenfolge `"abcd\n"` am Bildschirm ausgegeben werden:

```
- val bildschirm = TextIO.stdOut;
val strom3 = - : TextIO.outstream

- TextIO.output(bildschirm, "abcd\n");
abcd
val it = () : unit
```

- Zeichenfolgen können wie folgt von der Tastatur eingelesen werden:

```
- val tastatur = TextIO.stdIn;
val tastatur = - : TextIO.instream

- TextIO.input(tastatur);
123456\n
val it = "123456\n" : TextIO.vector
```

12.5.5. Die vordefinierte Prozedur `print`

SML bietet auch die vordefinierte Prpozedur `print`, die wir uns schon mehrmals benutzt haben, um Zeichenfolgen an die Standardausgabe zu leiten:

```
- print "abc\n";
abc
val it = () : unit
```

Obwohl `print` im Modul (Struktur) `TextIO` definiert ist, ist diese Prozedur nicht nur durch ihren langen Namen `TextIO.print` aufrufbar, sondern auch unter dem Namen `print`:

```
- TextIO.print "abc\n";
abc
val it = () : unit

- print "abc\n";
abc
val it = () : unit
```

12.5.6. Beispiel: Inhalt einer Datei einlesen und an die Standardausgabe weiterleiten

Die folgende Prozedur liest eine Datei namens `datei` ein und leitet ihren Inhalt an die Standardausgabe `TextIO.stdOut`, d.h. an den Bildschirm, weiter:

```
- fun cat datei =
      let val datei = TextIO.openIn datei;
          val zeichenfolge = ref "";
      in
        while ( zeichenfolge := TextIO.inputN(datei, 1);
                ! zeichenfolge <> ""
              )
        do
           TextIO.output(TextIO.stdOut, ! zeichenfolge);
        TextIO.closeIn datei
      end;
val cat = fn : string -> unit
```

13. Formale Beschreibung der Syntax und Semantik von Programmiersprachen

Für jede Programmiersprache müssen festgelegt werden:

1. die Syntax der Programmiersprache
   (welche Programme sind zur Auswertung zulässig)
2. die Semantik der Programmiersprache
   (wie sind zulässige Programme auszuwerten)

- 13.1 Formale Beschreibung der Syntax einer Programmiersprache
- 13.2 Formale Beschreibung der Semantik einer Programmiersprache:
  Ziele und Ansätze
- 13.3 Einführung in die denotationelle Semantik funktionaler Programmiersprachen

13.1. Formale Beschreibung der Syntax einer Programmiersprache

- Syntax versus Semantik
- Ziele von formalen Beschreibungen der Syntax von Programmiersprachen
- Lexikalische Analyse und Symbole versus Syntaxanalyse und Programme
- EBNF-Grammatiken zur formalen Syntaxbeschreibungen
- Kontextfreie Grammatiken zur formalen Beschreibung der Syntax von Programmen
- Syntaxdiagramme

13.1.1. Syntax versus Semantik

Nicht jeder Ausdruck ist ein zulässiger SML-Ausdruck; z.B.

```
- val 2mal x => 2 * x;
Error: syntax error: deleting DARROW INT
```

1. Zur Deklaration einer Funktion darf das reservierte Wort `val` nur verwendet werden, wenn ein `fn`–Ausdruck folgt.

   Andernfalls muss das reservierte Wort `fun` verwendet werden.
2. Der Name `2mal` ist in SML nicht zulässig, weil er mit einer Zahl anfängt.
3. Das Symbol => darf nur innerhalb von `fn`– oder `case`–Ausdrücken vorkommen.

Alle diese Mängel betreffen den Aufbau des Ausdrucks, nicht seine vermeintliche Bedeutung, die leicht zu erraten ist.

Syntax versus Semantik (II)

- Inkorrekt gebildete Ausdrücke können festgestellt werden, bevor sie ausgewertet werden.

  $\rightsquigarrow$ statische Prüfung möglich (also zur Übersetzungszeit)

- Die Syntax einer Programmiersprache bestimmt, ob Ausdrücke dieser Programmiersprache korrekt aufgebaut sind.

  Syntaxfehler können immer (in endlicher Zeit) festgestellt werden.

  Syntaktische Korrektheit ist eine *entscheidbare* Eigenschaft.

Der folgende Ausdruck ist syntaktisch korrekt gebildet:

```
- fun fak1 x = if x > 0
                then x * fak1 x - 1
                else 1;
val fak1 = fn : int -> int
```

Syntax versus Semantik (III)

Dies ist jedoch keine korrekte Deklaration der Fakultätsfunktion: Aufgrund der von SML festgelegten Präzedenzen steht er für die folgende Deklaration:

```
- fun fak1 x = if x > 0
                then x * (fak1 x) - 1
                else 1;
val fak1 = fn : int -> int
```

Die Anwendung der Funktion `fak1` auf eine echt positive ganze Zahl `x` führt zum rekursiven Aufruf von `fak1 x`. Folglich terminiert die Auswertung von `fak1 x` nicht.

Der Mangel in der Deklaration der Funktion `fak1` ist nicht syntaktischer, sondern *semantischer* Natur.

Syntax versus Semantik (IV)

Die Terminierung eines Programms ist eine sogenannte *semi-entscheidbare* Eigenschaft, d.h.:

1. Ein Algorithmus ist möglich, der die Terminierung eines Programms überprüft und nach endlicher Zeit die Terminierung meldet, falls das Programm terminiert.
2. Kein Algorithmus ist möglich, der die Terminierung eines Programms überprüft und nach endlicher Zeit meldet, ob das Programm terminiert oder nicht terminiert.

Für jeden Algorithmus, der die Terminierung von Programmen überprüft, wird es also immer Programme geben, die nicht terminieren und dessen Nichtterminierung der Algorithmus nicht feststellt.

Die Mängel in den Deklarationen von `2mal` und `fak1` unterscheiden sich prinzipiell:

1. Die syntaktische Korrektheit eines Ausdrucks (oder Programms) ist entscheidbar.
2. Viele wichtige semantische Eigenschaften von Programmen, wie u.a. die Terminierung, sind hingegen semi–entscheidbar.

13.1.2. Ziele von formalen Beschreibungen der Syntax von Programmiersprachen

Formale Beschreibungen der Syntax von Programmiersprachen dienen

- Programmierern als Gebrauchsanweisungen;
- als Grundlage für die Implementierung der Programmiersprache;
- zur automatischen Ableitung eines Algorithmus aus der (formalen) Syntaxbeschreibung, der die Korrektheit von Ausdrücken gemäß dieser Syntax überprüft.

Automatisch heißt hier, dass eine Prozedur höherer Ordnung eine Syntaxbeschreibung `SB` als Aufrufparameter erhält und daraus einen „Syntaxprüfer" gemäß `SB`, d.h. eine Prozedur `syntax_prüfer(SB)`, liefert.

Angewandt auf einen Ausdruck `expr` überprüft das Programm `syntax_prüfer(SB)`, ob `expr` der Syntaxbeschreibung `SB` entspricht oder nicht.

13.1.3. Lexikalische Analyse und Symbole versus Syntaxanalyse und Programme

Die Syntaxprüfung eines Programms (gemäß einer gegebenen Syntaxbeschreibung) besteht aus den folgenden zwei Phasen:

- Die *lexikalische Analyse*, d.h. die Überprüfung der Syntax der reservierten Namen (in SML wie etwa `fun`, `val`, `case`, `*`, `div`) und vom Programmierer frei ausgewählten Namen (wie etwa `2mal`, `fak1`).

  Namen werden in diesem Kontext auch *Symbole* oder *Lexeme* (Englisch *token*) genannt.

- Die *syntaktische Analyse* (oder *Syntaxanalyse*), d.h. die Überprüfung der korrekten Anordnung der Symbole (oder Lexeme) in einem Programm.

  Während der Syntaxanalyse wird auch aus dem Quellprogramm ein Programm in abstrakter Syntax erzeugt.

Ein Programm zur lexikalischen Analyse wird Symbolentschlüssler (auch *Lexer*, *Scanner*) genannt. Ein Programm zur Syntaxanalyse wird Zerteiler (oder *Parser*) genannt.

13.1.4. EBNF-Grammatiken zur formalen Syntaxbeschreibungen

Die *alphabetischen Namen* von SML fangen mit einem (kleinen oder großen) Buchstaben an, der gefolgt wird von endlich vielen (auch null) Buchstaben (`a` . . . `z` `A` . . . `Z`), Ziffern (`0` `1` `2` . . . `9`), Underscore (`_`) oder Hochkommata (single quote: `'`).

Die Definition der alphabetischen Namen von SML kann wie folgt mit vier sogenannten EBNF-Regeln formalisiert werden: [EBNF = *Erweiterte Backus-Naur-Form*]

Startsymbol:		AlphaName
AlphaName	::=	Anfangszeichen Folgezeichen * .
Anfangszeichen	::=	Buchstabe .
Folgezeichen	::=	Buchstabe | "0" | "1" | "2" | . . . | "9" | "_" | "'" .
Buchstabe	::=	"a" | "b" | "c" | . . . | "x" | "y" | "z"
		| "A" | "B" | "C" | . . . | "X" | "Y" | "Z" .

Dabei ist AlphaName das Startsymbol, und die EBNF-Regeln haben paarweise unterschiedliche linke Seiten (vor dem Zeichen ::=). Das Zeichen | im rechten Teil einer EBNF-Regel trennt Alternativen.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (II)

- Die erste Regel besagt, dass ein AlphaName aus genau einem Buchstabe gefolgt von beliebig vielen (eventuell null) Folgezeichen besteht.
- Die zweite Regel definiert ein Folgezeichen als einen Buchstaben oder 0 oder 1 . . . oder 9.
- Die dritte Regel definiert einen Buchstaben in ähnlicher Weise.

Die folgenden Symbole gehören zur Spezifikationssprache EBNF („::=“ wird manchmal als *becomes* gelesen; „*“ wird Kleene'scher Stern genannt):

AlphaName   Anfangszeichen   Folgezeichen   Buchstabe

::=   *   .   ” |

Es sind Metasymbole der Metasprache, mit der eine Objektsprache (hier die der SML-Namen) spezifiziert wird.

Das Zeichen „. . . “ hingegen ist eine Abkürzung, die nicht zur Spezifikationssprache EBNF gehört.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (III)

- Das Metasymbol „*“ bedeutet eine n-fache ($n \in \mathbb{N}$, eventuell $n = 0$) Wiederholung des vorstehenden Symbols.
- Das Metasymbol „.“ markiert das Ende einer EBNF-Regel.
- Das Zeichen „" “ wird verwendet, um Symbole der Objektsprache einzuführen (z.B. "4").
- Das Symbol „$|$“ bezeichnet eine Alternative.

Weitere Metasymbole:

- „$+$“ bedeutet eine n-fache ($n \in \mathbb{N}$ und $n \geq 1$) Wiederholung des vorstehenden Symbols.
- [A] drückt eine Option aus: [A] bedeutet kein oder ein Vorkommen von A.
- Die Klammern „)“ und „(“ sind weitere Metasymbole, womit „$|$“, „*“ und „$+$“ auf zusammengesetzte Ausdrücke angewandt werden können.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (IV)

Beispiel:

Ternärzahl ::= "0" | ("1" | "2") ("0" | "1" | "2") * .

Diese EBNF-Regel definiert Zahlen zur Basis 3 ohne führende Null. Die nächste Regel definiert Zahlen zur Basis 4, eventuell mit führender Null oder Vorzeichen:

Quatärzahl ::= ["+" | "-"] ("0" | "1" | "2" | "3") * .

Metasymbole wie

AlphaName Anfangszeichen Folgebuchstabe

im ersten Beispiel sind *Nichtterminalsymbole*. Wie sie heißen, ist belanglos, sie können beliebig umbenannt werden (allerdings konsistent in allen Regeln).

Die Symbole, die zwischen zwei " vorkommen, heißen *Terminalsymbole*.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (V)

Aus den folgenden EBNF-Regeln

Startsymbol:	Binaerzahl
Binaerzahl	::= Binaerziffer Binaerziffer * .
Binaerziffer	::= "0" | "1".

werden unendlich viele (Objekt-)Symbolen (oder Worte) einer Objektsprache unter anderem die folgenden hergeleitet:

0 1 01 10 001 010 100 101 ...

Eine Menge von EBNF-Regeln, die paarweise unterschiedliche linke Teile haben, zusammen mit der Angabe eines Startsymbols wird EBNF-Grammatik genannt. (Die Angabe des Startsymbols darf nicht vergessen werden!)

Die Herleitung eines (Objekt-)Symbols aus einer EBNF-Grammatik kann durch einen Algorithmus beschrieben werden.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (VI)

Herleitungsalgorithmus für EBNF-Grammatiken:

1. Wähle eine Alternative im rechten Teil der Regel für das Startsymbol.
2. So lange ein Ausdruck der Gestalt (S) mit S Terminal- oder Nichterminal-Symbol vorkommt, wähle willkürlich einen solchen Ausdruck (S) und ersetze ihn durch S.
3. So lange ein Nichterminalsymbol oder ein Ausdruck der Gestalt expr *, expr + oder [expr] vorkommt, wähle (willkürlich) einen der folgenden Schritte 3.1, 3.2, 3.3 oder 3.4 und fahre mit diesem Schritt fort:
   - 3.1 Fall Nichterminalsymbol:
     - 3.1.1 Wähle (willkürlich) ein Nichtterminalsymbol NT.
     - 3.1.2 Wähle (willkürlich) eine Alternative A in der rechten Seite der EBNF-Regel mit linker Seite NT.
     - 3.1.3 Ersetze das Nichtterminalsymbol NT durch die Alternative A.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (VII)

Herleitungsalgorithmus (Forts.):

3.2 Fall Ausdruck A der Gestalt expr * :
   Wähle (willkürlich) einen der folgenden Schritte:
   3.2.1 Streiche A = expr * .
   3.2.2 Ersetze A = expr * durch expr expr * .

3.3 Fall Ausdruck A der Gestalt expr + :
   Wähle (willkürlich) einen der folgenden Schritte:
   3.3.1 Ersetze A = expr + durch expr .
   3.3.2 Ersetze A = expr + durch expr expr + .

3.4 Fall Ausdruck A der Gestalt [expr]: Wähle (willkürlich) einen der folgenden Schritte:
   3.4.1 Streiche A = [expr] .
   3.4.2 Ersetze A = [expr] durch expr .

4. Fahre bei Schritt 2 fort.

5. Liefere das Ergebnis.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (VIII)

Beispiel einer Anwendung des Herleitungsalgorithmus:

EBNF-Grammatik:	Startsymbol:	Binaerzahl
	Binaerzahl	::= Binaerziffer Binaerziffer * .
	Binaerziffer	::= "0" | "1".

Im Folgenden wird immer der am weitesten rechts stehende Ausdruck entsprechend des Algorithmus bearbeitet (beginnend mit dem Startsymbol):

Binaerzahl				
Binaerziffer	Binaerziffer *			(Schritt 1)
Binaerziffer	Binaerziffer	Binaerziffer *		(Schritt 3.2.2)
Binaerziffer	Binaerziffer	Binaerziffer	Binaerziffer *	(Schritt 3.2.2)
Binaerziffer	Binaerziffer	Binaerziffer		(Schritt 3.2.1)
Binaerziffer	Binaerziffer	1		(Schritt 3.1.2)
Binaerziffer	0	1		(Schritt 3.1.2)
0	0	1		(Schritt 3.1.2)

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (IX)

Ein anderer Ablauf des Herleitungsalgorithmus, der ebenfalls die Binärzahl 001 liefert, ist wie folgt:

Binaerzahl				
Binaerziffer	Binaerziffer *			(Schritt 1)
0	Binaerziffer *			(Schritt 3.1.2)
0	Binaerziffer	Binaerziffer *		(Schritt 3.2.2)
0	0	Binaerziffer *		(Schritt 3.1.2)
0	0	Binaerziffer	Binaerziffer *	(Schritt 3.2.2)
0	0	Binaerziffer		(Schritt 3.2.1)
0	0	1		(Schritt 3.1.2)

Der Herleitungsalgorithmus liefert den Grund für die Bezeichnungen Terminal- und Nichtterminalsymbole: Während eines Ablaufs des Herleitungsalgorithmus bleiben Terminalsymbole erhalten, die Nichtterminalsymbole werden nach und nach ersetzt.

EBNF-Grammatiken zur formalen Syntaxbeschreibungen (X)

Im Schritt 2 können mehrere Ausdrücke (S) zur Wahl stehen. Der Herleitungsalgorithmus legt nicht fest, welcher gewählt werden soll. Man sagt, dass der Herleitungsalgorithmus *nichtdeterministisch* ist.

Man kann sich leicht davon überzeugen, dass die Wahl im Schritt 2 die Reihenfolge der Ersetzungen beeinflusst, jedoch nicht das am Ende des Ausführung des Algorithmus hergeleitete (Objekt-)Symbol.

Wie der Schritt 2 enthält der Schritt 3.4 eine nichtdeterministische Wahl, nämlich die Wahl einer Alternative. Im Gegensatz zur Wahl im Schritt 2 beeinflußt die Wahl im Schritt 3.4 das hergeleitete (Objekt-)Symbol.

Um beide Arten von Nichtdeterminismus zu unterscheiden, spricht man von *don't care*-Nichtdeterminismus (Schritt 2) und *don't know*-Nichtdeterminismus (Schritt 3.4).

13.1.5. Kontextfreie Grammatiken zur formalen Beschreibung der Syntax von Programmen

Die `local`–Ausdrücke von SML haben die folgende Gestalt:

```
local
   "Deklarationen"
in
   "Deklarationen"
end
```

wobei `"Deklarationen"` jeweils eine oder mehrere SML-Deklarationen bezeichnet.

`local`–Ausdrücke stellen eine Art dreielementige Klammerung dar: `local` ist wie eine öffnende Klammer, `in` ist ein Zwischenteil und `end` ist eine schließende Klammer.

Kontextfreie Grammatiken zur Syntax-Beschreibung . . . (II)

Die Gestalt der `local`–Ausdrücke kann wie folgt mit einer EBNF-Grammatik spezifiziert werden:

$$
\begin{array}{lll}
\text{Localausdruck} & ::= & "\texttt{local}" \\
 & & (\ \text{Deklaration}\ [\ ";"\ ]\ )+ \\
 & & "\texttt{in}" \\
 & & (\ \text{Deklaration}\ [\ ";"\ ]\ )+ \\
 & & "\texttt{end}".
\end{array}
$$

(Der Einfachheit halber wird in dieser EBNF-Regel die Möglichkeit der Reihung von SML-Deklarationen mit `and` zur nichtsequenziellen Auswertung nicht berücksichtigt.)

Weitere Regeln sind zur Spezifikation des Nichtterminalsymbols Deklaration nötig.

Im obigen EBNF-Regel kommen die Metasymbole $+$,), (,] und [vor. Die Klammern) und (dienen dazu, den Geltungsbereich des Operators $+$ anzugeben.

Kontextfreie Grammatiken zur Syntax-Beschreibung . . . (III)

Die folgende EBNF-Grammatik definiert arithmetische Ausdrücke über Binärzahlen:

Startsymbol:		ArithmAusd
ArithmAusd	::=	["+" | "−"] Term | Term ("+" | "−") Term .
Term	::=	Faktor | Faktor ("*" | "/") Faktor .
Faktor	::=	Binaerzahl | "(" ArithmAusd ")" .
Binaerzahl	::=	Binaerziffer Binaerziffer * .
Binaerziffer	::=	"0" | "1" .

Die folgenden Ausdrücke können damit hergeleitet werden:

```
10
+ 01
(10 * 111)
- (10 * 111)
+ ( + 01 + - (10 * 111) )
```

Kontextfreie Grammatiken zur Syntax-Beschreibung . . . (IV)

Die vorangehende EBNF-Grammatik weist EBNF-Regeln mit wesentlich komplizierteren rechten Seiten auf als die EBNF-Grammatik zur Definition der Syntax der alphanumerischen Namen von SML.

Eine EBNF-Grammatik heißt *regulär*, wenn jede Alternative einer rechten Seite jeder Regel

- ein Terminalsymbol oder
- ein Terminalsymbol gefolgt von einem Nichtterminalsymbol gefolgt von * ist.

Die EBNF-Grammatik für die Syntax der alphanumerischen Namen ist regulär.
Die obige EBNF-Grammatik für arithmetische Ausdrücke ist aber nicht regulär.

Reguläre Grammatiken ermöglichen keine Form der Klammerung, wie sie in `local`–Ausdrücke oder in den arithmetischen Ausdrücke vorkommt.

EBNF-Grammatiken sind in jedem Fall *kontextfreie* Grammatiken.

Kontextfreie Grammatiken zur Syntax-Beschreibung . . . (V)

Kontextfreie Grammatiken sind zu einfach, um alle Aspekte von natürlichen Sprache zu formalisieren:

Beispiel: eine kontextfreie Grammatik mit Startsymbol Satz

$$\begin{array}{lll} \text{Satz} & ::= & \text{Subjekt Verb .} \\ \text{Subjekt} & ::= & \text{"Sie"} \mid \text{"Er" .} \\ \text{Verb} & ::= & \text{"sieht"} \mid \text{"liest" .} \end{array}$$

Erlaubt korrekte deutsche Sätze wie etwa „Sie sieht“ und „Er liest“ herzuleiten.

Beispiel:

$$\begin{array}{lll} \text{Satz} & ::= & \text{Subjekt Verb .} \\ \text{Subjekt} & ::= & \text{"Er"} \mid \text{"Wir" .} \\ \text{Verb} & ::= & \text{"sehen"} \mid \text{"liest" .} \end{array}$$

Damit können inkorrekte Sätze wie „Er sehen“ und „Wir liest“ hergeleitet werden.

Kontextfreie Grammatiken zur Syntax-Beschreibung . . . (VI)

Die Berücksichtigung eines Kontexts ist mit kontextfreien Grammatiken nicht möglich.

Beispiel für nicht-kontextfreie Eigenschaften der Syntax von Programmiersprachen:

Die Verwendung eines Namens oder einer Ausnahme setzt voraus, dass dieser Name oder diese Ausnahme deklariert wurde, sonst:

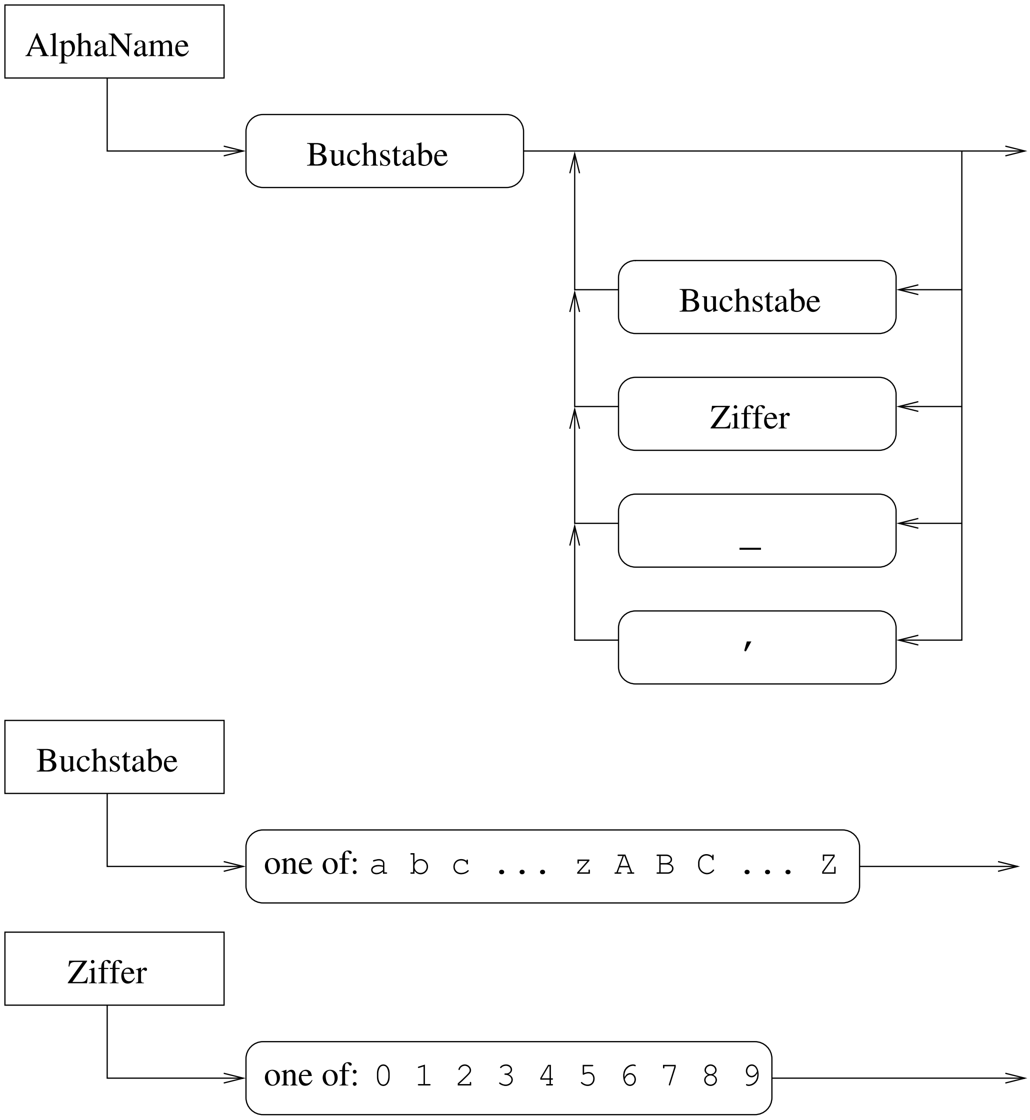
```
- 2 + drei;
Error: unbound variable or constructor: drei

- raise ausnahme;
Error: unbound variable or constructor: ausnahme
```

Um die Kontextabhängigkeiten einer Programmiersprache zu spezifizieren, wird eine kontextfreie Grammatik für diese Programmiersprache mit Zusatzbedingungen ergänzt. Auch diese Zusatzbedingungen werden statisch überprüft.

13.1.6. Syntaxdiagramme

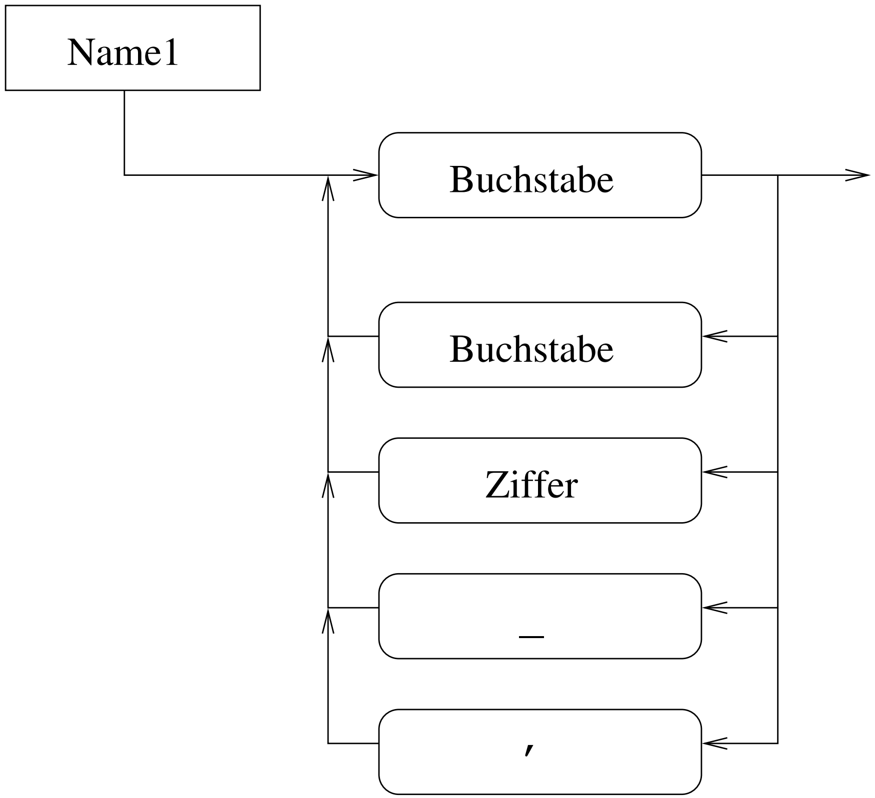
Anstelle von EBNF-Regel werden auch sogenannte Syntaxdiagramme verwendet. Die folgenden Diagramme geben z.B. die Definition der Syntax von alphanumerischen Namen von SML wieder:

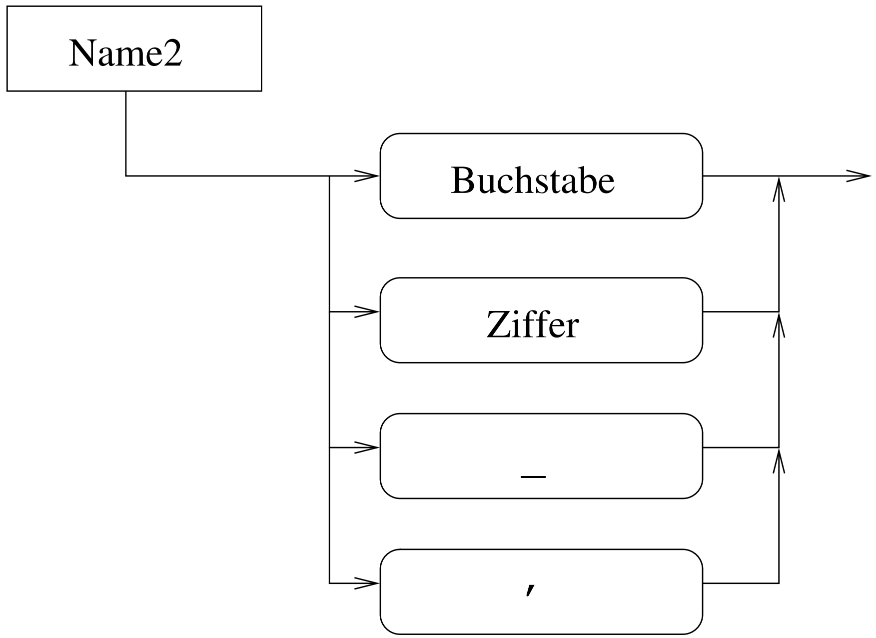


Man vergleiche diese Diagramme mit der EBNF-Grammatik zur Definition der Syntax von alphanumerischen Namen von SML.

Syntaxdiagramme (II)

Varianten der Definition von Namen mit unterschiedlichen Eigenschaften:





13.2. Formale Beschreibungen der Semantik einer Programmiersprache: Ziele und Ansätze

Eine formale Beschreibung der Semantik einer Programmiersprache wird dazu verwendet, um präzis festzulegen,

- was Programme in dieser Programmiersprache berechnen oder
- wie Programme in dieser Programmiersprache ausgeführt werden sollen.

Es kann z.B. sinnvoll sein zu untersuchen, ob ein Programm die Fibonacci-Zahlen berechnet, ohne sich dabei dafür zu interessieren, ob das Programm einen iterativen oder baumrekursiven Prozess implementiert.

Ein Ansatz zur Festlegung, was Programme in einer gegebenen Programmiersprache S berechnen, ist der Ansatz der *denotationellen Semantik*.

Formale Beschreibungen der Semantik . . . (II)

Wird der Rechenvorgang während der Ausführung eines Programms in einer gegebenen Programmiersprache beschrieben, so spricht man von einer *operationalen Semantik* dieser Programmiersprache. Operationale Semantiken von Programmiersprachen können in unterschiedlichen Formalismen definiert werden.

Zur Beschreibung sowohl des „was“ wie des „wie“ einer Berechnung mit imperativen Programmen werden sogenannten *axiomatische Semantiken* bevorzugt. Dabei handelt es sich um logische Kalküle.

Zur Beschreibung der Semantik einer rein funktionalen Programmiersprache eignet sich der Ansatz der *denotationellen Semantik* gut. Er beschreibt, was Programme berechnen, jedoch nicht wie berechnet wird. Wie rein funktionale Programme ausgewertet werden, lässt sich mit einem ziemlich einfachen Auswertungsalgorithmus beschreiben.

So liefert eine denotationelle Semantik für eine rein funktionale Programmiersprache eine nützliche, abstrakte Ergänzung des Auswertungsalgorithmus.

13.3. Einführung in die denotationelle Semantik funktionaler Programmiersprachen

- Mathematische Funktionen zur Repräsentation von Programmfunktionen
- Programmfunktionen versus mathematische Funktionen
- Werte von (mathematischen) Funktionen zur Repräsentation von Programmausdrücken ohne (Programm-)Wert
- Syntaktische und semantische (Wert-)Domäne, Semantikfunktionen
- Strikte und nicht-strikte (mathematische) Funktionen
- Approximationsordnung
- Denotation einer Funktionsdeklaration
- Semantische Domäne
- Denotationelle Semantik einer rein funktionalen Programmiersprache
- Fixpunktsemantik rekursiver Funktionen

13.3.1. Mathematische Funktionen zur Repräsentation von Programmfunktionen

Eine denotationelle Semantik für eine Programmiersprache S soll bestimmen, welche mathematische Funktion eine Funktionsdeklaration in der Programmiersprache S entspricht, d.h., welche mathematische Funktion die Denotation („Inhalt“, Bedeutung) der Funktionsdeklaration ist.

Betrachten wir wieder einmal die Deklaration von `fak1`:

```
- fun fak1 x = if x > 0
               then x * (fak1 x) - 1
               else 1;
val fak1 = fn : int -> int
```

`fak1` entspricht einer (math.) Funktion, die jede negative ganze Zahl auf 1 abbildet. Es liegt nahe, als Denotation von `fak1` eine partielle (math.) Funktion zu berücksichtigen, die auf den echt positiven ganzen Zahlen nicht definiert ist.

Mathematische Funktionen zur Repräsentation . . . (II)

Auch die folgende SML-Funktion `p2` stellt kein großes Verständnisproblem dar:

```
- fun p2(x, y) = y;
val p2 = fn : 'a * 'b -> 'b
```

`p2` implementiert offenbar die zweite Projektion für Paare, d.h. die folgende (math.) Funktion:

$$\begin{array}{rccc} proj_2 : & A \times B & \rightarrow & B \\ & (x, y) & \mapsto & y \end{array}$$

Welcher mathematischen Funktion entspricht aber das folgende Programm?

```
- val p2fak1 = fn (x, y) => p2(fak1 x, y);
```

Mathematische Funktionen zur Repräsentation . . . (III)

Beruht die Programmiersprache wie SML auf der Auswertung in applikativer Reihenfolge, so implementiert `p2fak1` die folgende partielle mathematische Funktion:

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$f(x,y) = \begin{cases} y & \text{falls } x \leq 0 \\ \mathit{undefiniert} & \text{sonst} \end{cases}$$

Für eine Programmiersprache, die auf einer Auswertung in normaler Reihenfolge oder auf einer verzögerten Auswertung beruht, implementiert aber `p2fak1` die mathematische Funktion $proj_2$.

Der Ansatz „denotationelle Semantik“ ermöglicht, Deklarationen wie den obigen eine Denotation für jede Auswertungsart zu geben.

13.3.2. Programmfunktionen versus mathematische Funktionen

Untersucht man die denotationelle Semantik einer Programmiersprache, so müssen die folgenden Begriffe auseinander gehalten werden:

- *Funktionsdeklaration:* der Ausdruck (aus einem Programm), dem eine Bedeutung (Denotation) gegeben werden soll;
- *Programmfunktion:* unser intuitives, oft unpräzises, manchmal fehlerhaftes Verständnis der Funktionsdeklaration — also die Intuition einer mathematischen Funktion;
- *mathematische Funktion:* ein formal definiertes Objekt.

Das Ziel einer denotationellen Semantik einer funktionalen Sprache S ist die Feststellung einer math. Funktion für jede Funktionsdeklaration in der Sprache S.

13.3.3. Werte (mathematischer) Funktionen zur Repräsentation von Programmausdrücken ohne (Programm-)Werte

Betrachten wir nochmals die Deklaration von `fak1`.
Ist `n` eine echt positive ganze Zahl, so terminiert die Auswertung von `fak1 n` nicht.
Die (intuitive) Programmfunktion f, die der Funktionsdeklaration `fak1` entspricht, ist also über die echt positiven ganzen Zahlen nicht definiert.
Anders ausgedrückt: $f(\mathtt{n})$ hat keinen Wert, wenn $\mathtt{n} \in \mathbb{Z}$ und $\mathtt{n} > 0$.

Programmausdrücke ohne Werte können in zwei Weise denotiert — d.h., mit einer Bedeutung versehen — werden:

1. Das math. Gegenstück eines Programmausdrucks ohne Wert ist undefiniert.
2. Das mathematische Gegenstück eines Programmausdrucks ohne Wert ist ein Sonderwert $\perp$ (gesprochen *bottom*).

Werte von (math.) Funktionen zur Repräsentation . . . (II)

- Der 1. Ansatz hat den Nachteil, Berechnungen im math. Modell zu erschweren.
- Mit dem 2. Ansatz kann man die Undefiniertheit eines Programmausdrucks feststellen, wenn der Programmausdruck den Sonderwert als Denotation erhält.

  Dieser 2. Ansatz wird deswegen gewählt.

Das Beispiel der Deklaration von `p2fak1` zeigt diesen Vorteil:

- Sei $\perp$ die Denotation von `fak1 4` ($\perp$ ist der Sonderwert für „undefiniert“).
- Ist $proj_2$ die Denotation von `p2`, so ist $proj_2(\perp, 4)$ die Denotation von `p2fak1 4`.
- Je nach Auswertungsart definiert man:
  - $proj_2(\perp, 4) = \perp$ (Auswertung in applikativer Reihenfolge) oder
  - $proj_2(\perp, 4) = 4$ (verzögerte Auswertung).

13.3.4. Syntaktische und semantische (Wert-)Domäne, Semantikfunktionen

Jedem Monotyp der Programmiersprache wird eine Menge, *semantische (Wert-)-Domäne* oder kurz *Domäne* genannt, als Denotation zugewiesen, die ein bottom-Element zur Repräsentation von undefinierten Programmwerten enthält.

Falls nötig, werden die bottom-Elemente unterschiedlicher Domänen dadurch unterschieden, dass das bottom-Element einer Domäne D als $\perp_D$ bezeichnet wird.

- Die Denotation des Typs `int` ist z.B. $\mathbb{Z} \cup \{\perp\}$.
- Die Denotation des Typs `real` ist z.B. $\mathbb{G} \cup \{\perp\}$, wobei $\mathbb{G}$ die Menge der Gleitkommazahlen ist ($\mathbb{G} \subset \mathbb{Q}$).
- Die Denotation des Typs `bool` kann z.B. die Menge $\{t, f, \perp\}$, $\{1, 0, \perp\}$ oder sogar $\{true, false, \perp\}$ sein.

  Wird die letzte dieser drei Mengen ausgewählt, so muss zwischen `true` als Ausdruck der Programmiersprache und dessen Denotation $true$ unterschieden werden.

Syntaktische und semantische Domänen, . . . (II)

Wir werden sehen, dass über die Denotation eines Typs eine Ordnung mit bestimmten Eigenschaften benötigt wird.

Die Programmausdrücke, die denselben Typ besitzen, bilden eine sogenannte *syntaktische (Wert-)Domäne*.

Eine Abbildung (d.h. math. Funktion), die die Elemente einer syntaktischen Domäne (die Ausdrücke eines bestimmten Typs `t`) auf ihre Denotation abbildet (also auf Elemente der semantischen Domäne von `t`), heißt *Semantikfunktion*.

Die Definition einer denotationellen Semantik für eine funktionale Programmiersprache besteht in der Festlegung von syntaktischen und semantischen Domänen sowie von Semantikfunktionen.

13.3.5. Strikte und nicht-strikte (mathematische) Funktionen

Beruht eine funktionale Programmiersprache auf einer Auswertung in applikativer Reihenfolge, so terminiert die Auswertung der Funktionsanwendung

```
funk(arg1, ..., argi, ..., argn)
```

nicht, wenn die Auswertung des `i`-ten Arguments `argi` nicht terminiert.

Sind die Denotationen der Ausdrücke `funk`, `arg1`, $\ldots$, `argi`, $\ldots$, `argn` die (math.) Objekte f (eine n-stellige Funktion), $d_1, \ldots, d_i, \ldots d_n$, so muss also gelten:

$$d_i = \perp : \quad f(d_1, \ldots, \perp, \ldots, d_n) = \perp$$

Beruht aber die Programmiersprache auf der verzögerten Auswertung, so kann es sein, dass

$$f(d_1, \ldots, \perp, \ldots, d_n) \neq \perp$$

Strikte und nicht-strikte (mathematische) Funktionen (II)

Definition (strikte und nicht-strikte (math.) Funktionen)
Seien $D_1, \ldots, D_n$ und D Mengen, die $\perp$ als Element haben, f eine n-stellige (math.) Funktion von $D_1 \times \ldots \times D_n$ in D und $i \in \{1, \ldots, n\}$.

- f heißt *strikt im i-ten Argument*, falls für alle $d_1 \in D_1, \ldots, d_{i-1} \in D_{i-1}, d_{i+1} \in D_{i+1}, \ldots, d_n \in D_n$ gilt:

$$f(d_1, \ldots, d_{i-1}, \perp, d_{i+1}, \ldots, d_n) = \perp$$

- f heißt *strikt*, wenn f in allen seinen Argumenten strikt ist.
- f heißt *nicht-strikt*, wenn f in mindestens einem seiner Argumente nicht strikt ist.

Die Denotation einer Funktionsdeklaration in einer Programmiersprache, die auf der Auswertung in applikativer Reihenfolge beruht, muss also eine strikte Funktion sein.

Strikte und nicht-strikte (mathematische) Funktionen (III)

Die Denotation einer Funktionsdeklaration in einer Programmiersprache, die auf der verzögerten Auswertung beruht, kann aber muss nicht eine strikte Funktion sein. So ist z.B. die Denotation des Ausdrucks

```
fun id x = x;
```

in einer Programmiersprache mit verzögerter Auswertung die strikte (math.) Funktion ist (strikt, weil $i(\perp) = \perp$):

$$\begin{array}{rccc} i: & A & \rightarrow & A \\ & x & \mapsto & x \end{array}$$

Nicht selten werden die Begriffe Denotation und Programmfunktion im informellen Sprachgebrauch verwechselt.
Die Verwendung der Begriffe „strikte“ und „nicht-strikte Funktionen“ ist nur bezüglich math. Funktionen (mit dem Sonderelement $\perp$ in Ursprungs- und Bildmenge) richtig, nicht für Programmfunktionen.

13.3.6. Approximationsordnung

```
fun fak2 x = if x = 0 then 1
                      else x * fak2 (x - 1);
```

Ist `n` eine positive ganze Zahl, so liefert der Aufruf `fak2 n` die Fakultät von `n`.
Die (math.) Funktion

$$
\begin{aligned}
f0: \quad & \mathbb{Z} \cup \{\perp\} \to \mathbb{Z} \cup \{\perp\} \\
& n \mapsto \begin{cases} n! & \text{falls } n \geq 0 \\ \perp & \text{falls } n < 0 \\ \perp & \text{falls } n = \perp \end{cases}
\end{aligned}
$$

ist eine mögliche Denotation für `fak2`, weil sie die folgenden Bedingungen erfüllt:

1. $f0$ bildet $\mathbb{Z} \cup \{\perp\}$ in $\mathbb{Z} \cup \{\perp\}$ ab;
2. $f0(\perp) = \perp$;
3. Wenn $f0(n) \neq \perp$, dann $f0(n) = n!$.

Approximationsordnung (II)

Der dritte Fall der Definition von $f0(n)$ ist notwendig, damit die Anwendung von `fak2` auf Programmausdrücke ohne Werte abgedeckt ist ($f0$ also total auf $\mathbb{Z} \cup \{\perp\}$) und $f0$ strikt ist, was die Auswertungsreihenfolge von SML verlangt.

Die folgende (math.) Funktion $f24$ ist aber auch eine mögliche Denotation für `fak2`:

$$\begin{aligned} f24: \quad & \mathbb{Z} \cup \{\perp\} \to \mathbb{Z} \cup \{\perp\} \\ & n \mapsto \begin{cases} n! & \text{falls } n \geq 24 \\ \perp & \text{falls } n < 24 \\ \perp & \text{falls } n = \perp \end{cases} \end{aligned}$$

Die Grenze von 24 statt 0 ist sicherlich etwas überraschend. $f24$ erfüllt aber genauso wie $f0$ die Bedingungen für eine Denotation der SML-Funktionsdeklaration von `fak2`:

1. $f24$ bildet $\mathbb{Z} \cup \{\perp\}$ in $\mathbb{Z} \cup \{\perp\}$ ab;
2. $f24(\perp) = \perp$;
3. Wenn $f24(n) \neq \perp$, dann $f24(n) = n!$.

Approximationsordnung (III)

Offenbar gibt es für jede positive ganze Zahl n eine (math.) Funktion fn, die als Denotation von `fak2` genauso gut (oder genauso schlecht) wie $f24$ in Frage kommt.

Auch die folgende (math.) konstante Funktion f_{inf}, die jedes Element von $\mathbb{Z} \cup \{\perp\}$ auf $\perp$ abbildet, kommt als Denotion von `fak2` in Frage:

$$\begin{array}{llll} f_{inf}: & \mathbb{Z} \cup \{\perp\} & \rightarrow & \mathbb{Z} \cup \{\perp\} \\ & n & \mapsto & \perp \end{array}$$

Wie jede Funktion fn erfüllt f_{inf} die Bedingungen 1., 2. und 3.:

1. f_{inf} bildet $\mathbb{Z} \cup \{\perp\}$ in $\mathbb{Z} \cup \{\perp\}$ ab;
2. $f_{inf}(\perp) = \perp$;
3. Wenn $f_{inf}(n) \neq \perp$, dann $f_{inf}(n) = n!$.

Warum soll die Denotation von `fak2` $f0$ und nicht $f24$ oder f_{inf} sein?
Der Begriff *Approximationsordnung* liefert dafür eine Begründung.

Approximationsordnung (IV)

Wir erinnern zunächst daran, wie eine Ordnung definiert ist:

- Eine (partielle) Ordnung über einer Menge M ist eine reflexive, transitive und antisymmetrische binäre Relation $\leq$ über M.
- Gilt $m1 \leq m2$ oder $m2 \leq m1$ für alle $m1, m2 \in M$, so ist $\leq$ eine *totale* Ordnung über M.

Ist D eine semantische Domäne, also eine Menge, die $\perp$ enthält, so wird die folgende Ordnung $\sqsubseteq$ über D definiert:

Definition

1. $\perp \sqsubseteq d$ für alle $d \in D$, d.h. $\perp$ ist Minimum der Ordnung $\sqsubseteq$;
2. $d \sqsubseteq d$ für alle $d \in D$.

Approximationsordnung (V)

Über $\mathbb{Z} \cup \{\perp\}$, der Denotation des SML-Typs `int`, ist die Approximationsordnung $\sqsubseteq$ wie folgt definiert:

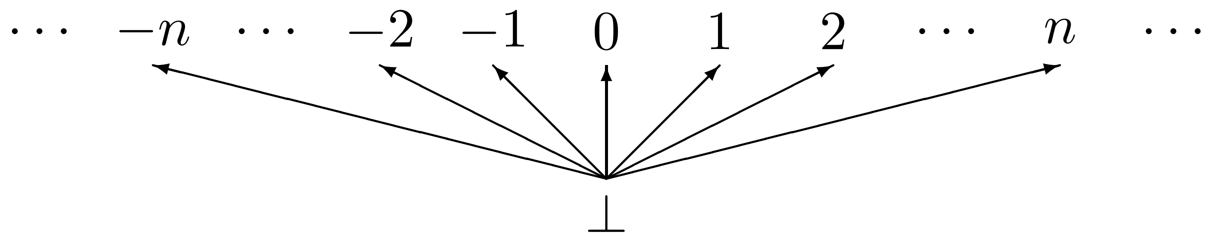
Definition

1. $\perp \sqsubseteq n$ für alle $n \in \mathbb{Z}$;
2. $n \sqsubseteq n$ für alle $n \in \mathbb{Z}$;
3. $\perp \sqsubseteq \perp$.

Man beachte, dass zwischen ganzen Zahlen n und m die Beziehung $n \sqsubseteq m$ nur dann gilt, wenn $n = m$.

Approximationsordnung (VI)

Diese Approximationsordnung über $\mathbb{Z} \cup \{\perp\}$ wird wie folgt graphisch dargestellt:



[gerichtete Kante von a nach b bedeutet $a \sqsubseteq b$; $a \sqsubseteq a$ wird nicht dargestellt]

Wird $\perp$ als „undefiniert" ausgelegt, so drückt $\sqsubseteq$ einen *Definiertheitsgrad* aus:
$\perp$ ist weniger definiert als jeder andere Wert, jeder Wert ist genau so viel definiert wie er selbst.

Die (math.) Funktion $f24$ ist weniger definiert als die (math.) Funktion $f0$, weil sie weniger Elemente der Urspungsmenge als $f0$ auf definierte Werte abbildet
— anders ausgedrückt: weil sie mehr Elemente der Urspungsmenge auf $\perp$ abbildet.

Approximationsordnung (VII)

Die folgende Definition formalisiert diesen Begriff „Definiertheitsgrad“.

Definition (Semantische Domäne)

- Eine semantische (Wert-)Domäne ist eine Menge D mit $\perp \in D$ (oder $\perp_D \in D$), über die eine Ordnung $\sqsubseteq$ (oder $\sqsubseteq_D$) definiert ist, für die gilt:

  1. $\perp \sqsubseteq d$ für alle $d \in D$;
  2. $d \sqsubseteq d$ für alle $d \in D$.

- Sind $D_1, \ldots, D_n$ semantische Domänen, so ist eine Ordnung $\sqsubseteq$ über das kartesische Produkt $D_1 \times \ldots \times D_n$ wie folgt definiert:

$$(d_1, \ldots, d_n) \sqsubseteq (e_1, \ldots, e_n) \quad :\Longleftrightarrow \quad d_1 \sqsubseteq e_1 \text{ und } \ldots \text{ und } d_n \sqsubseteq e_n$$

Approximationsordnung (VIII)

Definition (Approximationsordnung für Vektoren und Funktionen)

Sind D_1, ..., D_n und D semantische Domänen so ist eine Ordnung $\sqsubseteq$ über der Menge der (totalen) Funktionen von $D_1 \times \ldots \times D_n$ in D wie folgt definiert:

$$f \sqsubseteq g \quad :\Longleftrightarrow$$

$$\text{für alle } (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \text{ gilt: } f(d_1, \ldots, d_n) \sqsubseteq g(d_1, \ldots, d_n)$$

Diese Ordnung über der Menge der (totalen) Funktionen von $D_1 \times \ldots \times D_n$ in D heißt *Approximationsordnung*.

Man beachte, dass die beiden Definitionen zusammen auch den Fall abdecken, wenn die Bildmenge der Funktionen ein Kartesisches Produkt ist.

Im Allgemeinen liegt über (math.) Funktionen keine totale Ordnung vor.

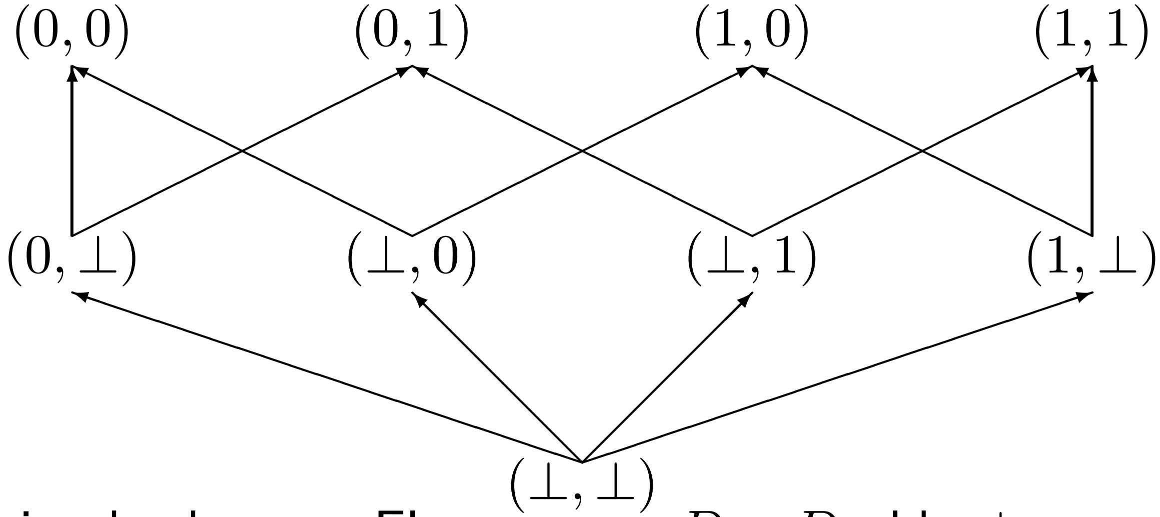
Approximationsordnung (IX)

Betrachten wir die semantische Domäne $D = \{0, 1, \perp\}$ mit der partiellen Ordnung:

$$\perp \sqsubseteq 0 \qquad 0 \sqsubseteq 0 \qquad \perp \sqsubseteq 1 \qquad 1 \sqsubseteq 1$$

Für diese Ordnung sind 0 und 1 unvergleichbar: Es gilt weder $0 \sqsubseteq 1$ noch $1 \sqsubseteq 0$.

Darstellung der auf $D \times D$ induzierte Ordnung:



Das Paar $(\perp, \perp)$ ist das bottom-Element von $D \times D$, d.h. $\perp_{D \times D} = (\perp_D, \perp_D)$.
Die oben eingeführte Ordnung über $D \times D$ ist nicht total:
die Vektoren $(0, \perp)$ und $(1, 1)$ sind z.B. für diese Ordnung nicht vergleichbar.

13.3.7. Denotation einer Funktionsdeklaration

Die Approximationsordnung ermöglicht, zwischen $f0$, $f24$, fn und f_{inf} als Denotationskandidaten für die SML-Funktionsdeklaration `fak2` zu unterscheiden:
Bezüglich der Approximationsordnung ist $f0$ maximal: es gilt:

$$f_{inf} \sqsubseteq fn \sqsubseteq fm \sqsubseteq f0 \text{ für alle } n, m \in \mathbb{N} \text{ mit } n \geq m.$$

Eine der möglichen Denotationen einer Funktionsdeklaration, die bezüglich der Approximationsordnung maximal ist (also am wenigsten undefiniert ist), soll als Denotation ausgewählt werden.

Dies ist sicherlich ein vernünftiges Kriterium:
Die formale Bedeutung einer Funktionsdeklaration soll eine (math.) Funktion sein, die so weit definiert ist, wie die Funktionsdeklaration es überhaupt ermöglicht.

Denotation einer Funktionsdeklaration (II)

Es stellen sich jedoch die folgenden Fragen:

- Existiert immer eine maximale (math.) Funktion unter den Denotationskandidaten einer Funktionsdeklaration?
- Falls es immer mindestens eine maximale (math.) Funktion unter den Denotationskandidaten gibt, ist sie eindeutig (d.h. gibt es nur eine einzige maximale Funktion)?

Beide Fragen können mit ja beantwortet werden, so dass der vorangehende Ansatz zur Denotation einer Funktionsdeklaration einen Sinn ergibt.

Für nichtrekursive Funktionsdeklarationen ist dies nicht schwer zu zeigen.
Der Beweis für rekursive Funktionsdeklarationen ist aber keineswegs unmittelbar (siehe Ende des Kapitels).

13.3.8. Semantische Domäne

Der vorangehende Abschnitt hat als semantische Domäne für einen Vektor ein kartesisches Produkt von Domänen betrachtet. Je nach dem, ob die Auswertung der Programmiersprache, deren Semantik spezifiziert wird, eine Auswertung in applikativer Reihenfolge oder eine verzögerte Auswertung ist, werden die semantischen Domänen für zusammengesetzte Typen unterschiedlich definiert.

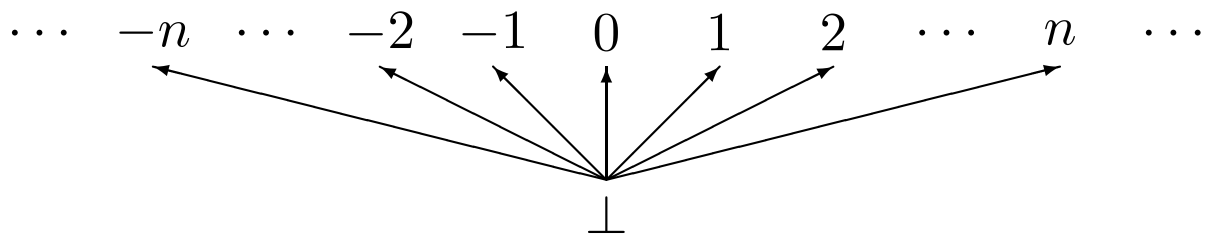
Flache Domäne

Sei $\mathbb{Z} \cup \{\perp\}$ die Domäne, die die Denotation des Typs `int` ist.

Die Ordnung $\sqsubseteq$ über $\mathbb{Z} \cup \{\perp\}$ ist wie folgt definiert:

$$\perp \sqsubseteq n \text{ für alle } n \in \mathbb{Z}$$

Graphische Darstellung dieser Ordnung, die nicht mit $\leq$ verwechseln werden darf:



Eine solche Ordnung, die nur Vergleiche mit $\perp$ bietet, wird *flache Ordnung* genannt.

Über den semantischen Domänen, die Denotation nicht-zusammengesetzter Monotypen (wie etwa `int`, `real`, `bool`) sind, haben wir immer flache Ordnungen.

Vollständigkeit der semantischen Domäne

Eine $\sqsubseteq$–Kette in einer semantischen Domäne D ist eine (abzählbar) unendliche Folge

$$d_0, d_1, d_2, \ldots, d_n, \ldots$$

von Elementen von D, so dass:

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$$

d.h. für alle $i \in \mathbb{N}$ gilt: $d_i \sqsubseteq d_{i+1}$. Eine obere Schranke (*upper bound*) einer $\sqsubseteq$–Kette ist ein Wert s, so dass für jedes $i \in \mathbb{N}$ gilt: $d_i \sqsubseteq s$.

Eine obere Schranke k der Kette heißt kleinste obere Schranke (*least upper bound*), wenn $k \sqsubseteq s$ für jede obere Schranke s der Kette gilt.

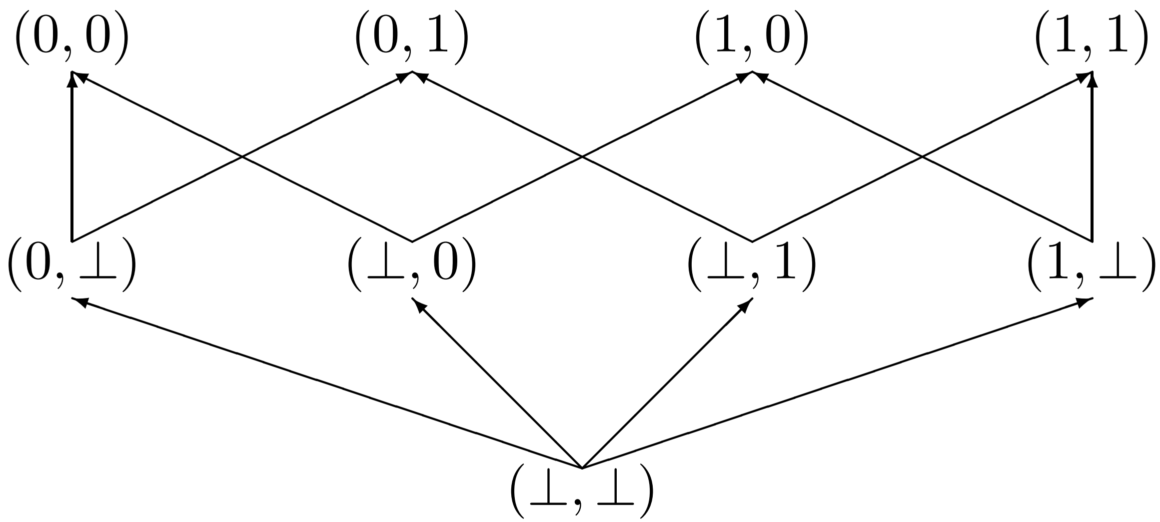
Beispiel: Die $\leq$–Kette 0, 1, 2, . . . , n, . . . hat keine obere Schranke in $\mathbb{N}$.

In der flachen Domäne $D = \mathbb{Z} \cup \{\perp\}$ hat jede Kette in D eine kleinste obere Schranke in D, weil jede Kette nur zwei unterschiedliche Elemente enthält.
(ab einem gewissen Index n gilt immer $d_n = d_{n+1} = \ldots = d_{n+m} = \ldots$)

Vollständigkeit der semantischen Domäne (II)

- Es wird von einer semantischen Domäne D verlangt, dass jede $\sqsubseteq$–Kette, informell *Approximationskette* genannt, in D eine kleinste obere Schranke in D besitzt.
- Diese *Vollständigkeitsbedingung* für semantische Domänen stellt sicher, dass jede *Approximationskette* ein Limes innerhalb der semantischen Domäne hat.

Betrachten wir die Denotation $D = \{0, 1, \perp\}$ des Typs `bool` und die Denotation $D \times D$ des Typs `bool * bool`:



Offenbar hat jede $\sqsubseteq$–Kette in $D \times D$ eine kleinste obere Schranke in D.

Vollständigkeit der semantischen Domäne (III)

Für endliche semantische Domänen stellt die Vollständigkeit keine strenge Bedingung dar. Zur Denotation von Funktionsdeklarationen werden aber unendliche semantische Domänen verwendet. Eine mögliche Denotation für die Deklaration

```
fun pluseins n = n + 1;
```

kann aber z.B. die folgende unendliche semantische Domäne sein:

$$D = \{(n, n+1) | n \in \mathbb{Z}\} \cup \{(\perp, \perp)\}$$

Eine Kette über D kann wie folgt definiert werden:

$$\text{Für } m \in \mathbb{N}: \quad f_m: \quad D \to D \qquad n \mapsto \begin{cases} \perp & \text{falls } n \in \mathbb{Z} \text{ und } n \geq m \\ n+1 & \text{falls } n \in \mathbb{Z} \text{ und } n < m \\ \perp & \text{falls } n = \perp \end{cases}$$

Vollständigkeit der semantischen Domäne (IV)

Offenbar gilt: $f_m \sqsubseteq f_{m+1}$ für alle $m \in \mathbb{N}$.

Man kann zeigen, dass die folgende Funktion die kleinste obere Schranke der Kette $(f_m)_{m \in \mathbb{N}}$ ist:

$$\begin{array}{llll} f_\infty : & D & \to & D \\ & n & \mapsto & n+1 \quad \text{falls } n \in \mathbb{Z} \end{array}$$

Diese Funktion f_∞ ist ein Element der Menge der Funktionen von D in D.

Man kann zeigen, dass diese Funktionsmenge vollständig ist.

Die Domäneprodukte $\times$ und $\otimes$

Sind $D1$ und $D2$ semantische Domänen, so bezeichnet $D1 \times D2$ die semantische Domäne mit bottom-Element $(\perp, \perp)$ und mit der Approximationordnung, die von den Ordnungen von $D1$ und $D2$ induziert wird.

Die Produktdomäne $D1 \times D2$ ist sinnvoll, wenn eine Programmiersprache mit verzögerter Auswertung betrachtet wird. Vektoren der Gestalt $(\perp, d2)$ mit $d2 \neq \perp$ und $(d1, \perp)$ mit $d1 \neq \perp$ ermöglichen, Funktionsdeklarationen wie

```
fun p2 (x : int, y : int) = y;
```

als Denotation eine nicht-strikten Funktion wie

$$\begin{array}{rccl} proj_2 : & \mathbb{Z} \cup \{\perp\} \times \mathbb{Z} \cup \{\perp\} & \rightarrow & \mathbb{Z} \cup \{\perp\} \\ & (x, y) & \mapsto & y \end{array}$$

zu geben, für die gilt: $proj_2(\perp, y) = y$.

Die Domäneprodukte $\times$ und $\otimes$ (II)

Bei Auswertung in applikativer Reihenfolge werden als mögliche Denotationen nur strikte Funktionen berücksichtigt, so dass Vektoren der Gestalt $(\perp, d2)$ mit $d2 \neq \perp$ und $(d1, \perp)$ mit $d1 \neq \perp$ unerwünscht sind.

Sie werden in der folgenden Definition einer Produktdomäne ausgeschlossen:

$$D1 \otimes D2 = \{\ (d1, d2) \mid d1 \in D1, d1 \neq \perp, d2 \in D2, d2 \neq \perp\ \} \ \cup \ \{\ (\perp, \perp)\ \}$$

Das bottom-Element von $D1 \otimes D2$ ist $(\perp, \perp)$ (d.h. $(\perp_{D1}, \perp_{D2})$) und die Approximationsordnung von $D1 \otimes D2$ wird wie zuvor definiert.

Die Domänevereinigungen $\oplus$ und $+$

- In der *verquickenden* (oder *verschmelzenden*) Vereinigung $D1 \oplus D2$ werden
  - die bottom-Elemente von $D1$ und $D2$ gleichgesetzt und
  - die anderen Elemente auseinandergehalten.

  Haben $D1$ und $D2$ einen Schnitt, der nicht nur $\perp$ enthält, so müssen also die Elemente von $D1$ (oder $D2$) umbenannt werden.

- Die Ordnung über $D1 \oplus D2$ ist die Vereinigung der Ordnungen über $D1$ und $D2$.

  Außer $\perp$ kann kein Element von $D1$ mit einem Element von $D2$ in $D1 \oplus D2$ verglichen werden.

- Die *unterscheidende* Vereinigung $D1 + D2$ hält jedes Element von $D1$ (einschl. $\perp_{D1}$) und von $D2$ (einschl. $\perp_{D2}$) auseinander.

- Die Ordnung über $D1 + D2$ ist die Vereinigung der Ordnungen von $D1$ und $D2$.

- Damit $D1+D2$ über ein Minimum verfügt, wird $D \cup D2$ ein Element $\perp$ hinzugefügt, das weder in $D1$ noch in $D2$ vorkommt. Für dieses (neue) $\perp$-Element gilt:

  $\forall d1 \in D1 : \perp \sqsubseteq d1,\ \forall d2 \in D2 : \perp \sqsubseteq d2$; und somit: $\perp \sqsubseteq \perp_{D1}, \perp \sqsubseteq \perp_{D2}$.

Die Domänevereinigungen $\oplus$ und $+$ (II)

Die Domänevereinigungen werden zur Denotation von Typen mit Varianten verwendet wie etwa:

```
datatype t = k1 of t1 | ... | kn of tn;
```

Sind $D_1, \ldots, D_n$ die Denotationen (d.h. semantischen Domänen) der Typen `t1`, . . . , `tn`, so ist bei einer Programmiersprache mit Auswertung in applikativer Reihenfolge die verquickende Vereinigung $D_1 \oplus \ldots \oplus D_n$ die Denotation von `t`.

Man kann sich leicht davon überzeugen, dass $\oplus$ assoziativ ist, so dass der Ausdruck $D_1 \oplus \ldots \oplus D_n$ einen Sinn ergibt.

Angehobene Domäne

Ist D eine semantische Domäne, so erhält man die angehobene (*lifted*) semantische Domäne $D_\perp$ dadurch, dass D um ein (neues) $\perp$ Element ergänzt wird, für dass gilt:

$$\begin{array}{ll} \perp \neq d & \text{für alle } d \in D; \text{ insb. } \perp \neq \perp_D \\ \perp \sqsubseteq d & \text{für alle } d \in D; \text{ insb. } \perp \sqsubseteq \perp_D \end{array}$$

Man beachte, dass für alle Domänen $D1$ und $D2$ gilt:

$$\begin{array}{rcl} D1 + D2 & = & D1_\perp \oplus D2_\perp \\ (D1 \times D2)_\perp & = & D1_\perp \otimes D2_\perp \end{array}$$

Die Domänenanhebung ermöglicht eine andere Denotation eines Typs mit Varianten.

Angehobene Domäne (II)

Betrachten wir dazu wieder die `datatype`–Deklaration:

```
datatype t = k1 of t1 | ... | kn of tn;
```

Der Wertkonstruktor `k1` hat den Typ `t1 -> t`. Ist D_1 die Denotation des Typs `t1`, so kann die semantische Domäne der Werte des Typs `t` der Gestalt `k1(.)` die Denotation $(\{k1\} \times D_1)_\perp$ erhalten. So ist die Denotation eines Wertes der Gestalt `k1(.)` ein Paar $(k1, d_1)$ mit $d_1 \in D_1$ (auch $k1(d_1)$ geschrieben).
Diese Denotation ermöglicht, den Wert $(k1, \perp_{D_1})$ vom Wert $\perp$ (als Element der Denotation von `t`) zu unterscheiden.

Die Denotation von `t` ist dann: $(\{k1\} \times D_1)_\perp \oplus \ldots \oplus (\{kn\} \times D_n)_\perp$.

Wenn $D_1 = \ldots = D_n$, dann gilt:

$$(\{k1\} \times D_1)_\perp \oplus \ldots \oplus (\{kn\} \times D_n)_\perp = (\{k1, \ldots, kn\} \times D)_\perp$$

Semantische (Wert-)Domäne zur Denotation von Polytypen

Die Denotation des Polytyps `'a list` besteht aus den (unendlich vielen) Denotationen der Listentypen, die sich aus der Bindung der Typvariable `'a` an einem Typausdruck ergibt, also den Denotationen der Typen `int list`, `bool list`, `real list`, `int list list`, usw.

Die Verallgemeinerung dieses Ansatzes auf beliebige Polytypen stellt keine prinzipielle Schwierigkeit dar.

Ein polymorpher Ausdruck wie die leere Liste `nil` (oder `[]`) gehört zu jedem Listentyp. Ihre Denotation muss also Element der Denotation jedes Listentyps sein.

Die Denotation eines Polytyps besteht also in einer (unendlichen) Menge von Denotationen, deren Durchschnitt die Menge der polymorphen Ausdrücke des Polytyps ist.

Semantische Domäne zur Denotation von rekursiven Typen

Beruht die Programmiersprache auf die Auswertung in applikativer Reihenfolge, so ist die Denotation der folgenden Deklaration eines rekursiven Typs

```
datatype t = k1 | k2 of t;
```

eine Menge D, die die folgende rekursive Gleichung erfüllen muss (vgl. auch mit „angehobenen Domänen“):

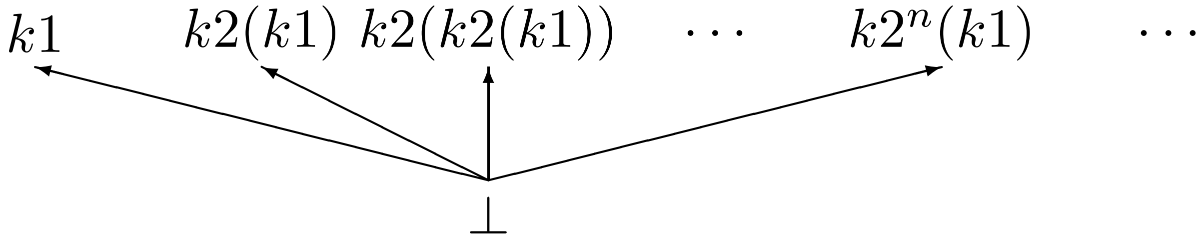
$$D = \{k1\}_\perp \oplus (\{k2\} \times D)_\perp$$

Diese Gleichung entspricht genau der Deklaration des Typs.

Eine solche rekursive Gleichung stellt aber keine Definition dar!

Semantische Domäne zur Denotation von rekursiven Typen (II)

Es kann jedoch bewiesen werden, dass die folgende flache Domäne die passende Lösung dieser rekursiven Gleichung ist:



Ferner kann bewiesen werden, dass jede rekursive Gleichung, die sich aus der Deklaration eines rekursiven Typs ergibt, eine Lösung hat. So kann also jeder Deklaration eines rekursiven Typs eine semantische Domäne zugewiesen werden.

Für Programmiersprachen mit verzögerter Auswertung wird eine andere, kompliziertere Art von (auch rekursiven) Gleichungen verwendet, mit denen ein rekursiver Typ die Deklaration von „unendlichen“ (d.h. zyklischen) Werten ermöglicht.

Auch diese komplizierteren rekursiven Gleichungen haben immer Lösungen.

Funktionsdomäne, Monotonie und Stetigkeit

Es bietet sich an, als Denotation einer Funktionsdeklaration mit Typ `t1 -> t2` die Menge der (math.) Funktionen von $D1$ in $D2$ zu betrachten, wenn $D1$ die Denotation von `t1` und $D2$ die Denotation von `t2` ist.

- Nicht alle (math.) Funktionen sind aber als Denotation von Funktionsdeklarationen sinnvoll, weil nicht alle (math.) Funktionen berechnet werden können.
- Die Denotation eines Funktionstyps `t1 -> t2` enthält also nur die Funktionen von der Denotation von `t1` in die Denotation von `t2`, die einige Eigenschaften besitzen, die die Berechenbarkeit der (math.) Funktionen sicherstellen.

  Es handelt sich dabei um zwei Eigenschaften: Monotonie und Stetigkeit.

Monotonie

Sei $f : D1_\perp \to D2_\perp$ eine (math.) Funktion mit beliebigen Mengen $D1$ und $D2$ und den angehobenen Domänen $D1_\perp$ und $D2_\perp$.

Wenn gilt

1. $f(\perp) = d2 \in D2$ (also $d2 \neq \perp$) und
2. $f(d1) = \perp$ für ein $d1 \in D1$ (also $d1 \neq \perp$)

dann eignet sich f nicht als Denotation einer Funktionsdeklaration:

Terminiert die Auswertung einer Funktionsanwendung auf einen Ausdruck ohne Wert (1.), so muss ebenfalls die Auswertung derselben Funktion auf einen Ausdruck mit Wert terminieren, was der (2.) widerspricht.

Die Approximationsordnung ermöglicht eine einfache Charakterisierung der (math.) Funktionen, die als Denotation einer Funktionsdeklaration zulässig sind, nämlich die monotonen (math.) Funktionen.

Monotonie (II)

Definition (monotone (math.) Funktionen)
Seien $D1$ und $D2$ zwei semantische Domänen und f eine Funktion von $D1$ in $D2$.
f heißt *monoton*, wenn gilt:
Für alle $x \in D1$ und $y \in D1$, wenn $x \sqsubseteq y$, dann $f(x) \sqsubseteq f(y)$.

Die Funktion f des vorangehenden Beispiels ist nicht monoton,
weil $f(d1) = \perp \sqsubseteq d2 = f(\perp)$ trotz $\perp \sqsubseteq d1$.

Monotonie (III)

Die Monotoniebedingung hat weitreichende Folgen; am Beispiel:

```
fun pluseins x = x + 1;
```

Ist f die Denotation von `pluseins`, so muss gelten:

$$f(\perp) \sqsubseteq f(0) \quad \text{und} \quad f(\perp) \sqsubseteq f(1)$$

weil $\perp \sqsubseteq 0$ und $\perp \sqsubseteq 1$. Da aber in der flachen Domäne $\mathbb{Z} \cup \{\perp\}$, die die Bildmenge von f ist, $\perp$ das einzige Element ist, das ≤ 0 und ≤ 1 ist, muss gelten:

$$f(\perp) = \perp \quad \text{und} \quad f(0) \neq f(1)$$

Das heißt, dass die Denotation von `pluseins` eine strikte Funktion sein muss — und dies unabhängig davon, auf welcher Auswertungsform die Programmiersprache beruht.

Stetigkeit

Eine (math.) Funktion f von $D1$ in $D2$ heißt *stetig* genau dann, wenn f den Limeswert (d.h. die kleinste obere Schranke) einer $\sqsubseteq$–Kette in $D1$ $(d_i)_{i \in \mathbb{N}}$ auf den Limeswert (d.h. die kleinste obere Schranke) der $\sqsubseteq$–Kette in $D2$ $(f(d_i))_{i \in \mathbb{N}}$ abbildet.

Ist eine Funktion stetig, so ist sie auch monoton:

- Man betrachtete eine $\sqsubseteq$–Kette in der Ursprungsmenge einer stetigen Funktion f, die höchstens zwei unterschiedliche Elemente enthält:

  z.B. $x = d_0 \sqsubseteq y = d_1 = d_2 = \ldots$.
- Offenbar ist y die kleinste obere Schranke der $\sqsubseteq$–Kette.
- Wenn f stetig ist, dann gilt $f(x) \sqsubseteq f(y)$.

Sind $D1$ und $D2$ zwei semantische Domänen, so bezeichnet

- $[D1 \to D2]$ die Menge der stetigen Funktionen von $D1$ in $D2$ und
- $[D1 \to_\perp D2]$ die Menge der stetigen und strikten Funktionen von $D1$ in $D2$.

(Semantische) Umgebung

Eine Umgebung bildet eine endliche Menge von Namen (oder Bezeichner, Variablen) auf Werte ab.

Die Denotation einer Umgebung (*(semantische) Umgebung* genannt) ist also keine beliebige (math.) Funktion, sondern eine partielle Funktion mit endlichem Bereich.

Der Bereich $Bereich(f)$ einer partiellen Funktion $f : A \to B$ wie folgt definiert ist:

$$Bereich(f) = \{\ a \mid \exists b \in B : (a, b) \in f\ \}$$

Die Menge der partiellen (math.) Funktionen mit endlichem Bereich wird bezeichnet:

$$[A \to_{fin} B]$$

(Semantische) Umgebung (II)

Bezeichnet Var die Menge der Namen (oder Variablen) der Programmiersprache und Val die verquickende Vereinigung der Denotationen der (unendlich vielen) Typen, so ist die Denotation einer Umgebung eine partielle Funktion aus der Menge

$$[Var \rightarrow_{fin} Val]$$

Sind $env1$ und $env2$ zwei (semantische) Umgebungen, so bezeichnet $env1 + env2$ die folgende (semantische) Umgebung:

$$(env1 + env2)(x) = \begin{cases} env2(x) & \text{falls } x \in Bereich(env2) \\ env1(x) & \text{andernfalls} \end{cases}$$

Die (sem.) Umgebung $env2$ „überschattet“ die Umgebung $env1$ in $(env1 + env2)$. $(env1 + env2)$ wird verwendet, wenn eine Deklaration (durch $env2$ dargestellt) eine vorhandene Umgebung ($env1$) verändert.

(Semantische) Umgebung (III)

Ist $\mathtt{N}$ ein Name und w ein (semantischer) Wert, so bezeichnet

$$[w/\mathtt{N}]$$

die (semantische) Umgebung mit Bereich $\{\mathtt{N}\}$, die $\mathtt{N}$ auf w abbildet. Die (semantische) Umgebung $[w/\mathtt{N}]$ liefert also eine Abbildung für keinen weiteren Namen als $\mathtt{N}$.

Die „Summe" $(env + [w/\mathtt{N}])$ einer (semantischen) Umgebungen env mit einer (semantischen) Umgebungen $[w/\mathtt{N}]$ entspricht also der Veränderung einer Umgebung, die aus folgender Deklaration folgt:

```
val N = w;
```

13.3.9. Denotationelle Semantik einer rein funktionalen Programmiersprache

- Es wäre sehr aufwändig, wenn für jede Funktionsdeklaration in einem Programm eine Denotation angegeben werden müsste.
- Statt dessen wird die Denotation systematisch aus jeder Funktionsdeklaration generiert. Dies wird dadurch möglich, dass der Implementierung der Programmiersprache selbst eine Denotation gegeben wird.

  $\leadsto$ *Semantikfunktion* für die Programmiersprache
- Die Anwendung der Semantikfunktion auf eine Funktionsdeklaration liefert die Denotation dieser Funktionsdeklaration.

Im Folgenden definieren wir als Beispiel eine Semantikfunktion für die Sprache SMalL (ohne Ausnahmen).

Zunächst werden die syntaktischen und semantischen Domänen sowie die abstrakte Syntax festgelegt, auf die sich die Semantikfunktion bezieht.

Syntaktische (Wert-)Domäne

Exp	: die Menge der SMalL-Ausdrücke
Var	: die Menge der Namen (oder Variablen, Bezeichnern)
$ConsExp$	: die Menge der konstanten Ausdrücke
Def	: die Menge der SMalL-Deklarationen

Jede dieser vier Mengen ist unendlich, aber abzählbar. Sie umfassen jeweils alle möglichen Ausdrücke, Namen, etc, nicht nur die in einem Programm verwendeten.

Für SMalL enthält $ConsExp$ nur Ausdrücke für ganze Zahlen wie etwa `007` und `~12`, da SMalL nur einen Konstantentyp, die ganzen Zahlen, umfasst.

Weil sowohl Namen als auch Deklarationen Ausdrücke sind, gilt:

$$
\begin{aligned}
Var &\subset Exp \\
ConsExp &\subset Exp \\
Def &\subset Exp
\end{aligned}
$$

Abstrakte Syntax

Die Definition einer Semantikfunktion für eine Programmiersprache bezieht sich auf die abstrakte Syntax dieser Programmiersprache, die formal beschrieben werden sollte (*hier nicht explizit angegeben*).

Die Umwandlung eines SMalL-Programms in konkreter Syntax in ein SMalL-Programm in abstrakter Syntax sollte ebenfalls formal festgelegt werden (*hier auch nicht explizit angegeben*).

Eine (math.) Funktion $[\![\, . \,]\!]$ wird eingeführt, die einen SMalL-Ausdruck (z.B. ein SMalL-Programm) in konkreter Syntax `A` auf den entsprechenden SMalL-Ausdruck in abstrakter Syntax $[\![\, \mathtt{A} \,]\!]$ abbildet.

Semantische (Wert-)Domäne

$ConsVal$:	die Domäne der semantischen Konstantenwerte; $ConsVal = \mathbb{Z}_\perp$, weil alle SMalL-Konstanten vom Typ `int` sind;
$FnVal$:	die Domäne der semantischen Funktionswerte;
Val:	die Domäne aller möglichen semantischen Werte;
Env:	die Menge der (semantischen) Umgebungen.

Seien die (unendlich vielen) Funktionsdomänen D_{ij} mit $i, j \in \mathbb{N}$ wie folgt definiert:

$$
\begin{array}{lll}
D_{11} & = D(z, z) & = [\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp] \\
D_{12} & = D(z, (z, z)) & = [\mathbb{Z}_\perp \to_\perp D(z, z)] \\
D_{21} & = D((z, z), z) & = [D(z, z) \to_\perp \mathbb{Z}_\perp] \\
D_{22} & = D((z, z), (z, z)) & = [D(z, z) \to_\perp D(z, z)] \\
\vdots & \vdots & \vdots
\end{array}
$$

Semantische (Wert-)Domäne (II)

Da SMalL Funktionen höherer Ordnung zulässt und auf Auswertung in applikativer Reihenfolge beruht, ist $FnVal$ wie folgt definiert:

$$FnVal = D(z,z) \oplus D(z,(z,z)) \oplus D((z,z),z) \oplus D((z,z),(z,z)) \oplus \ldots$$

Daraus folgt die Menge Val, die Domäne aller möglichen semantischen Werte:

$$Val = ConsVal \oplus FnVal$$

Die Menge der (semantischen) Umgebungen Env entspricht $[Var \rightarrow_{fin} Val]$.

Da die Mengen $ConsVal$, $FnVal$, Val und Env semantische Domänen sind, enthält jede ein bottom-Element und ist jeweils eine Ordnung $\sqsubseteq$ definiert, bezüglich der das bottom-Element minimal ist.

Die Einschränkungen der Programmiersprache SMalL ermöglichen einfache Definitionen dieser semantischen Domänen.

Die Semantikfunktion $SemVal$

- $SemVal$ ist eine curried Funktion vom Typ: $Exp \to Env \to Val$.
- $SemVal$ erhält also als Argument einen SMalL-Ausdruck in abstrakter Syntax und liefert eine Funktion, die auf eine (semantische) Umgebung angewandt einen semantischen Wert liefert.

  Ist `A` ein SMalL-Ausdruck in konkreter Syntax und env eine semantische Umgebung, so ist also

  $$SemVal([\![\, \texttt{A} \,]\!])\ env \qquad \text{auch geschrieben: } SemVal\ [\![\, \texttt{A} \,]\!]\ env$$

  ein semantischer Wert.

$SemVal$ ist keine Programmfunktion, sondern eine (math.) Funktion.

Auch wenn es in der Mathematik nicht üblich ist, Funktionen in curried Form zu definieren, ist es hier im Zusammenhang mit funktionaler Programmierung sinnvoll.

Die Semantikfunktion $SemVal$ (II)

Die Definition von $SemVal$ wird durch die folgenden Fälle skizziert:

- $SemVal\ [|\,\texttt{C}\,|]\ env = c \in ConsVal = \mathbb{Z}_\perp$ falls $\texttt{C} \in ConsExp$

  Jedem konstanten Ausdruck ist das passende Element in $ConsVal$ zuzuweisen.

- $SemVal\ [|\,\texttt{unop A}\,|]\ env$

  $$= \begin{cases} unop'\ SemVal\ [|\,\texttt{A}\,|]\ env & \text{falls } SemVal\ [|\,\texttt{A}\,|]\ env \neq \perp \\ \perp & \text{andernfalls} \end{cases}$$

  wobei die unäre Operation $unop' \in ConsVal$ dem Ausdruck `unop` entspricht.

- $SemVal\ [|\,\texttt{A1 binop A2}\,|]\ env$

  $$= \begin{cases} SemVal\ [|\,\texttt{A1}\,|]\ env\ binop'\ SemVal\ [|\,\texttt{A2}\,|]\ env \\ \qquad \text{falls } SemVal\ [|\,\texttt{A1}\,|]\ env \neq \perp \\ \qquad \text{und } SemVal\ [|\,\texttt{A2}\,|]\ env \neq \perp \\ \perp \qquad \text{andernfalls} \end{cases}$$

  wobei die Binäroperation $binop' \in ConsVal$ dem Ausdruck `binop` entspricht.

Die Semantikfunktion $SemVal$ (III)

- $SemVal\ [\![\,\texttt{if Test then A1 else A2}\,]\!]\ env$

$$= \begin{cases} SemVal\ [\![\,\texttt{A1}\,]\!]\ env & \text{falls die Auswertung von } \texttt{Test} \text{ terminiert} \\ & \text{und } \texttt{Test} \text{ in } env \text{ erfüllt ist} \\ SemVal\ [\![\,\texttt{A2}\,]\!]\ env & \text{falls die Auswertung von } \texttt{Test} \text{ terminiert} \\ & \text{und } \texttt{Test} \text{ in } env \text{ nicht erfüllt ist} \\ \bot & \text{falls } \texttt{Test} \text{ den semantischen Wert } \bot \text{ hat} \end{cases}$$

  Die Erfüllung oder Nichterfüllung der Bedingung `Test` in einer gegebenen Umgebung muss auch noch mathematisch definiert werden (*hier nicht angegeben*).

- $SemVal\ [\![\,\texttt{V}\,]\!]\ env = env(\texttt{V})$ falls $\texttt{V} \in Var$

- $SemVal\ [\![\,\texttt{let D in A end}\,]\!]\ env = SemVal\ [\![\,\texttt{A}\,]\!]\ env'$

  wobei $env' = env + SemEnv\ [\![\,\texttt{D}\,]\!]\ env$ ($SemEnv$ wird noch definiert).

Die Semantikfunktion $SemVal$ (IV)

- $SemVal\ [\![\, \texttt{fn N => A} \,]\!]\ env = f \in FnVal$

  mit: für jedes $w \in Val$ gilt $f(w) = SemVal\ [\![\, \texttt{A} \,]\!]\ (env + [w/\texttt{N}])$

- $SemVal\ [\![\, (\texttt{fn N => A1})\ \texttt{A2} \,]\!]\ env = SemVal\ [\![\, \texttt{A1} \,]\!]\ (env + [w2/\texttt{N}])$

  wobei $w2 = SemVal\ [\![\, \texttt{A2} \,]\!]\ env$.

- $SemVal\ [\![\, \texttt{A1 A2} \,]\!]\ env$

$$= \begin{cases} f(SemVal\ [\![\, \texttt{A2} \,]\!]\ env) & \text{falls } SemVal\ [\![\, \texttt{A1} \,]\!]\ env = f \in FnVal \\ \perp & \text{andernfalls} \end{cases}$$

- $SemVal\ [\![\, \texttt{A1 ; A2} \,]\!]\ env$

$$= \begin{cases} SemVal\ [\![\, \texttt{A2} \,]\!]\ env & \text{falls } \texttt{A1} \notin Def \\ SemVal\ [\![\, \texttt{A2} \,]\!]\ env1 & \text{andernfalls, wobei } env1 = SemEnv\ [\![\, \texttt{A1} \,]\!]\ env \end{cases}$$

- $SemVal\ [\![\, \texttt{val N = A} \,]\!]\ env = SemVal\ [\![\, \texttt{A} \,]\!]\ env$

Die Semantikfunktion $SemEnv$

$SemEnv$ ist eine (math.) Funktion in curried Form vom Typ $Def \to Env \to Env$.

Sie bildet also eine Deklaration auf eine Funktion ab, die eine (semantische) Umgebung auf eine (semantische) Umgebung abbildet.

Ist `D` eine SMalL-Deklaration in konkreter Syntax und env eine semantische Umgebung, so definiert $SemEnv\ [|\,\texttt{D}\,|]\ env$ lediglich die Bindungen, die aufgrund der Deklaration `D` die Umgebung env verändern.

Die (math.) Umgebung, die sich aus einer (math.) Umgebung env und einer Deklaration `D` ergibt, ist also $(env + SemEnv\ [|\,\texttt{D}\,|]\ env)$.

Die Semantikfunktion $SemEnv$ (II)

Die Definition der Funktion $SemEnv$ kann wie folgt skizziert werden:

- $SemEnv\ [|\,\texttt{D1 D2}\,|]\ env = (env1 + env2)$

  wobei $env2 = SemEnv\ [|\,\texttt{D2}\,|]\ (env + env1)$ und $env1 = SemEnv\ [|\,\texttt{D1}\,|]\ env$

- $SemEnv\ [|\,\texttt{D1 ; D2}\,|]\ env$ wird wie im vorangehenden Fall definiert.

- $SemEnv\ [|\,\texttt{local D1 in D2 end}\,|]\ env = env2$

  wobei $env2\ = SemEnv\ [|\,\texttt{D2}\,|]\ (env + env1)$ und $env1\ = SemEnv\ [|\,\texttt{D1}\,|]\ env$

  In der konkreten Syntax von SMalL entsprechen `local`–Ausdrücke Ausdrücken der abstrakten Syntax, die Sequenzen von Deklarationen sind.

Die Semantikfunktion $SemEnv$ (III)

- $SemEnv\ [|\,\texttt{val N = A}\,|]\ env = (env + [w/\texttt{N}])$

  wobei $w = SemVal\ [|\,\texttt{A}\,|]\ env$

- $SemEnv\ [|\,\texttt{val rec N = A}\,|]\ env = env'$

  wobei env' durch die folgende (rekursive!) Gleichung definiert ist:

  $env' = (env + [w/\texttt{N}])$ mit $w = SemVal\ [|\,\texttt{A}\,|]\ env'$

  [Man erkennt hier dieselbe Art zyklischer Verweise wie in der operationalen Umgebung.]

Die Semantikfunktion $SemEnv$ (IV)

Die vorangehende Spezifikation von $SemEnv$ $[|$ `val rec N = A` $|]$ wirft zwei Fragen auf:

1. Existiert immer für jede (semantische) Umgebung env eine (semantische) Umgebung env', die der vorangehenden Spezifikation entspricht?
2. Falls ja, kann diese Umgebung berechnet werden?

Kann die erste Frage nicht positiv beantwortet werden, dann ist die vorangehende Spezifikation ein Fehlschlag, weil sie eben das Ziel verfehlt, die sich aus einer `val rec`–Deklaration ergebende Umgebung (formal) zu definieren.

Eine positive Antwort auf die zweite Frage soll sicherstellen, dass ein Algorithmus zur Generierung der Umgebung env' aus $[|$ `val rec N = A` $|]$ und env bekannt ist.

Bemerkung über die vorangehenden Definitionen der Semantikfunktionen $SemVal$ und $SemEnv$

Die Semantikfunktionen $SemVal$ und $SemEnv$ sind hier nicht vollständig definiert worden.

Vollständige Definitionen würden u.a. eine präzisere Behandlung der bottom-Elemente benötigen und eine sorgfältige Unterscheidung zwischen inkorrekten Ausdrücken, die statisch erkannt werden, und inkorrekten Ausdrücken, die erst dynamisch als solche erkannt werden, erfordern.

13.3.10. Fixpunktsemantik rekursiver Funktionen

Die Verwendung rekursiver Gleichungen zur Definition ist prinzipiell problematisch. Rekursive Gleichungen werden oft erst dann verstanden, wenn deren Auswertung betrachtet wird.

Wenn rekursive Funktionsdeklarationen passende operationale Spezifikationen darstellen, sind sie keineswegs zufriedenstellende deklarative, d.h. vom Auswertungsalgorithmus unabhängige, Definitionen.

Es lohnt sich also zu untersuchen, was die Denotation, d.h. die mathematisch formalisierte Bedeutung, einer beliebigen rekursiven Funktionsdeklaration eigentlich ist.

Der Einfachheit halber wird im Folgenden keine wechselseitige rekursive Funktionsdeklarationen betrachtet.

Von der Rekursion zu einer Definition am Beispiel der Fakultätsfunktion

Betrachten wir die folgende Implementierung in SML der Fakultätsfunktion:

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

Die folgende Programmfunktion höherer Ordnung `fak'` ist nichtrekursiv, liefert jedoch die „Struktur“ der rekursiven Programmfunktion `fak`:

```
- val fak' = fn g => (fn x => if x = 0
                              then 1
                              else x * g (x - 1));
val fak' = fn : (int -> int) -> int -> int
```

Von der Rekursion zu einer Definition . . . (II)

Unter Verwendung der nichtrekursiven Programmfunktion höherer Ordnung `fak'` lässt sich die vorangehende rekursive Programmfunktion `fak` als `h` neu definieren:

```
 - val rec h = (fn x => fak' h x);
val h = fn : int -> int

- h 4;
val it = 24 : int
```

Die Programmfunktion `h` weist dasselbe operationale Verhalten wie `fak` auf, obwohl `h` und `fak` syntaktisch unterschiedlich sind. Die Denotationen von `h` und `fak` müssen daher dieselbe (math.) Funktion aus $[\mathbb{Z}_\perp \to_\perp \mathbb{Z}_\perp]$ sein.

`fak'` kann zur Deklaration nichtrekursiver Programmfunktionen verwendet werden, die die (rekursive) Programmfunktion `fak` approximieren (im Sinne der Approximationsordnung).

Von der Rekursion zu einer Definition . . . (III)

Wir deklarieren zunächst eine Programmfunktion `undefiniert: int -> int`, die nicht terminiert, wenn sie auf irgendeine ganze Zahl angewandt wird:

```
- val rec undefiniert : int -> int = fn x => undefiniert x;
val undefiniert = fn : int -> int
```

Die Denotation der Programmfunktion `undefiniert` ist offensichtlich die folgende (math.) Funktion aus $[\mathbb{Z}_\perp \rightarrow_\perp \mathbb{Z}_\perp]$:

$$\begin{array}{rccc} b: & \mathbb{Z}_\perp & \rightarrow & \mathbb{Z}_\perp \\ & n & \mapsto & \perp \end{array}$$

Die flache Domäne $\mathbb{Z} \cup \{\perp\}$ ist die Denotation des Typs `int`.

Man beachte, dass die (math.) Funktion b das bottom-Element der semantischen Domäne $[\mathbb{Z}_\perp \rightarrow_\perp \mathbb{Z}_\perp]$ der stetigen und strikten (math.) Funktionen der flachen Domäne $\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp$ ist.

Von der Rekursion zu einer Definition . . . (IV)

Seien die folgenden (unendlich vielen!) nichtrekursiven Programmfunktionen `faki` (`i` $\in \mathbb{N}$) wie folgt (gedanklich!) deklariert:

```
- val fak0 = undefiniert;
- val fak1 = fak' fak0;
- val fak2 = fak' fak1;
- val fak3 = fak' fak2;
- val fak4 = fak' fak3;
- val fak5 = fak' fak4;
...
```

Offenbar hätten die folgenden nichtrekursiven Deklarationen dieselben Programmfunktionen wie die vorangehenden Deklarationen definiert:

```
val fak1 = fn x => if x = 0 then 1 else x * fak0 (x - 1);
val fak2 = fn x => if x = 0 then 1 else x * fak1 (x - 1);
val fak3 = fn x => if x = 0 then 1 else x * fak2 (x - 1);
...
```

Von der Rekursion zu einer Definition . . . (V)

Man kann sich leicht davon überzeugen, dass das Folgende gilt:

(*) Für alle $\mathtt{i} \in \mathbb{N}\ \{0\}$:

1. für alle $\mathtt{j} \in \mathbb{Z}$ mit $0 \leq \mathtt{j} \leq \mathtt{i} - 1$ terminiert die Auswertung von `faki j` und liefert den Wert `j!`;
2. für alle $\mathtt{j} \in \mathbb{Z}$ mit $\mathtt{j} < 0$ oder $\mathtt{j} \geq \mathtt{i}$ terminiert die Auswertung von `faki j` nicht.

Unter Verwendung der (nichtrekursiven) Programmfunktionen $(\mathtt{faki})_{\mathtt{i} \in \mathbb{N}}$ kann die rekursive Programmfunktion `fak` wie folgt induktiv neu definiert werden:

Basisfall:	für $\mathtt{i} < 0$	$\mathtt{fak\ i} = \mathtt{fak0\ i}$
Induktionsfall:	für $\mathtt{i} \geq 0$	$\mathtt{fak\ i} = \mathtt{fak}_{(\mathtt{i}+1)}\ \mathtt{i}$

oder auch wie folgt:

Basisfall:	für $\mathtt{i} < 0$	$\mathtt{fak\ i} = \mathtt{fak0\ i}$
Induktionsfall:	für $\mathtt{i} \geq 0$	$\mathtt{fak\ i} = (\mathtt{fak}'^{(\mathtt{i}+1)})\ \mathtt{fak0\ i} = (\mathtt{fak}'^{(\mathtt{i}+1)})\ \perp$

Von der Rekursion zu einer Definition . . . (VI)

Die vorangehende induktive Neudefinition der Programmfunktion `fak` ist als Funktionsdeklaration nicht brauchbar, weil SML (wie alle Programmiersprachen) keine induktive Funktionsdefinition ermöglicht.
Programmiersprachen bieten dafür rekursive Deklarationen an.

Die induktive Form liefert die Denotation der rekursiven Programmfunktion `fak`:

Sei f_i die Denotation der nichtrekursiven Programmfunktion `faki`. Diese Denotation ist unproblematisch zu ermitteln, weil `faki` nicht rekursiv ist.
Aus (*) folgt (3. folgt aus der applikativen Auswertungsreihenfolge in SML):

(**) Für alle $i \in \mathbb{N} \setminus \{0\}$ gilt:

1. für alle $j \in \mathbb{Z}$ mit $0 \leq j \leq i - 1$ gilt: $f_i(j) = j!$;
2. für alle $j \in \mathbb{Z}$ mit $j < 0$ oder $j \geq i$ gilt: $f_i(j) = \perp$;
3. $f_i(\perp) = \perp$.

Von der Rekursion zu einer Definition . . . (VII)

Es gilt ferner:

$$\perp = f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq f_{n+1} \sqsubseteq \ldots$$

Die Folge $(f_i)_{i \in \mathbb{N}}$ ist eine $\sqsubseteq$–Kette in $D = [\mathbb{Z}_\perp \rightarrow_\perp \mathbb{Z}_\perp]$.

Jede solche $\sqsubseteq$–Kette von (math.) Funktionen aus einer Menge $D = [D1 \rightarrow_\perp D2]$ oder $D = [D1 \rightarrow D2]$ hat eine kleinste obere Schranke $s \in D$ hat (*Beweis folgt*).

Diese kleinste obere Schranke s ist also eine strikte bzw. nicht notwendigerweise strikte (math.) Funktion aus D, je nach dem, ob $D = [D1 \rightarrow_\perp D2]$ oder $D = [D1 \rightarrow D2]$.

Da s eine obere Schranke der $\sqsubseteq$–Kette $(f_i)_{i \in \mathbb{N}}$ ist, folgt aus (**):

Für alle $i \in \mathbb{N}$ gilt:

1. für alle $j \in \mathbb{Z}$ mit $0 \leq j \leq i - 1$ gilt: $f_i(j) = j! \sqsubseteq s(j)$;
2. für alle $j \in \mathbb{Z}$ mit $j < 0$ oder $j \geq i$ gilt: $f_i(j) = \perp \sqsubseteq s(j)$;
3. $s(\perp) = \perp$ (weil sonst s keine kleinste obere Schranke der Kette $(f_i)_{i \in \mathbb{N}}$ wäre).

Von der Rekursion zu einer Definition . . . (VIII)

Folglich:

1. Für alle $j \in \mathbb{Z}$ mit $0 \leq j$ gilt: $s(j) = j!$;
2. Für alle $j \in \mathbb{Z}$ mit $j < 0$ gilt: $s(j) = \bot$ (weil sonst s keine kleinste obere Schranke der Kette $(f_i)_{i \in \mathbb{N}}$ wäre);
3. $s(\bot) = \bot$.

Man beachte, dass die kleinste obere Schranke s der Kette $(f_i)_{i \in \mathbb{N}}$ die gewünschte Denotation von `fak` ist.

Im Folgenden wird gezeigt, dass die Denotation jeder rekursiven Funktionsdeklaration sich in ähnlicher Weise definieren lässt.

Systematische Herleitung einer Funktionsdefinition aus einer beliebigen rekursiven Funktionsdeklaration — Der Fixpunktsatz

Die zuvor betrachtete Vorgehensweise zur Herleitung der Denotation der rekursiven Programmfunktion `fak` aus ihrer Deklaration, lässt sich problemlos auf jede beliebige rekursive Funktionsdeklaration übertragen.

Allgemeine Form einer rekursiven Funktionsdeklaration (`expr` ist ein Ausdruck):

```
val rec funk = expr
```

Ähnlich wie `fak'` aus der rekursiven Deklaration von `fak` gewonnen werden kann, kann eine Programmfunktion höherer Ordnung `funk'` aus einer beliebigen rekursiven Deklaration einer Programmfunktion `funk` hergeleitet werden:

```
val funk' = fn funk => expr
```

Dabei wird angenommen, dass der Name `funk'` im Ausdruck `expr` nicht vorkommt.

Systematische Herleitung . . . (II)

Das Überschatten hat zur Folge, dass im definierenden Ausdruck

```
fn funk => expr
```

der Deklaration von `funk'` der Name `funk` ein formaler Parameter ist.

Wie im Fall der Programmfunktion `fak` kann eine (beliebige) Programmfunktion `funk` unter Verwendung der Programmfunktion höherer Ordnung neu definiert werden:

(#)   `val rec funk = funk' funk`

Ähnlich wie bei `fak` kann hier `funk'` verwendet werden, eine Folge nichtrekursiver Programmfunktionen $(\mathtt{funk}_i)_{i \in \mathbb{N}}$ zu definieren

Systematische Herleitung . . . (III)

[`undefiniert` ist die Programmfunktion, die auf keinem Wert definiert ist:]

```
funk0 = undefiniert
funk1 = funk'(funk0) = funk'(undefiniert)
funk2 = funk'(funk1) = funk'(funk'(undefiniert))
    ⋮
```

$$\texttt{funki = funk'(funki-1) = } \underbrace{\texttt{funk}'(\texttt{funk}' \ldots (\texttt{funk}'}_{\texttt{i mal}}\ \texttt{undefiniert}) \ldots))$$

Aus (#) folgt, dass die Denotationen von `funk` und `funk'(funk)` gleich sein sollen:

Denotation von `funk` = Denotation von `funk'(funk)`

Ist also F die Denotation der (nichtrekursiven) Programmfunktion `funk'` und f die Denotation der Programmfunktion `funk`, so soll gelten: $f = F(f)$.

Das heißt, die Denotation f der (rekursiven) Programmfunktion `funk` soll eine Lösung der rekursiven Gleichung $g = F(g)$ sein.

Systematische Herleitung . . . (IV)

Eine Lösung der Gleichung $g = F(g)$ wird *Fixpunkt von F* genannt.

Ist ein Fixpunkt von F für die Approximationsordnung kleiner als jeder andere Fixpunkt von F, so spricht man von einem *kleinsten Fixpunkt*.

Es stellen sich zwei Fragen:

1. Hat die rekursive Gleichung $g = F(g)$ immer mindestens eine Lösung?
2. Hat die Gleichung $g = F(g)$ mehrere Lösungen, welche dieser Lösungen soll als Denotation der Funktionsdeklaration `funk` ausgewählt werden?

Der folgende Fixpunktsatz von Kleene liefert Antworten auf beide Fragen:
Die Denotation f der (rekursiven) Programmfunktion `funk` ist der kleinste Fixpunkt der rekursiven Gleichung $g = F(g)$ ist.

Systematische Herleitung . . . (V)

Bevor der Satz angegeben und bewiesen wird, wird zunächst eine Notation eingeführt.

Ist $D1$ die Denotation der Ursprungsmenge der Programmfunktion `funk` und $D2$ die Denotation der Bildmenge von `funk`, so ist die (math.) Funktion f ein Element von $D = [D1 \rightarrow D2]$ und die (math.) Funktion F ein Element von $[D \rightarrow D]$.

Man definiert Potenzen von F wie folgt:

$$
\begin{array}{rll}
F^0 & = \perp_D, & \text{d.h. die konstante (math.) Funktion, die jedes Element} \\
 & & \text{von } D1 \text{ auf das bottom-Element von } D2 \text{ abbildet.} \\
F^1 & = F(F^0) & \\
\vdots & \vdots & \\
F^{i+1} & = F(F^i) &
\end{array}
$$

Eine Funktion F^i ist also ein Element von $[D1 \rightarrow D2]$.

Fixpunktsatz von Kleene

Fixpunktsatz (Kleene)

Sei D eine Domäne, $F \in [D \to D]$, d.h. eine stetige Funktion von D in D.

1. Die rekursive Gleichung $g = F(g)$ mit einer Unbekannten $g \in D$ hat immer mindestens eine Lösung.
2. Die kleinste obere Schranke der Folge $(F^i)_{i \in \mathbb{N}}$ ist eine Lösung der rekursiven Gleichung $g = F(g)$, die für die Approximationsordnung $\sqsubseteq$ über D kleiner ist als jede andere Lösung dieser rekursiven Gleichung.

Fixpunktsatz von Kleene (II)

Beweis:

a. Die Folge $(F^i)_{i \in \mathbb{N}}$ ist eine $\sqsubseteq$–Kette in $D = [D1 \to D2]$, wobei $\sqsubseteq$ die Approximationsordnung über $D = [D1 \to D2]$ ist.

   Nach Definition von $F^0 = \bot_D$ gilt: $F^0 \sqsubseteq F^1$.

   Sei $i \in \mathbb{N}$, $d \in D1$. $F^{i+1}(d) = F(F^i(d))$. Da F stetig, ist F monoton. Folglich ist $F^i(d) \sqsubseteq F(F^i(d)) = F^{i+1}(d)$. Da i und d beliebig gewählt wurden, ist $(F^i)_{i \in \mathbb{N}}$ eine $\sqsubseteq$–Kette.

b. Da D nach Annahme eine Domäne ist, besitzt die $\sqsubseteq$–Kette $(F^i)_{i \in \mathbb{N}}$ eine kleinste obere Schranke s in D. Wir zeigen nun, dass s ein Fixpunkt von F, d.h. eine Lösung der rekursiven Gleichung $g = F(g)$ ist:

   Da F nach Annahme stetig ist, ist $F(s)$ (nach Definition der Stetigkeit) die kleinste obere Schranke der Folge $(F(F^i))_{i \in \mathbb{N}}$.

   Die Folge $(F(F^i))_{i \in \mathbb{N}}$ ist aber identisch mit der Folge $(F^i)_{i \in \mathbb{N} \setminus \{0\}}$.

   Die kleinste obere Schranke s der Folge $(F^i)_{i \in \mathbb{N}}$ ist ebenfalls die kleinste obere Schranke der Folge $(F^i)_{i \in \mathbb{N} \setminus \{0\}}$, d.h. $s = F(s)$. [qed]

Fixpunktsatz von Kleene (III)

c. Aus b. folgt, dass die rekursive Gleichung $g = F(g)$ immer mindestens eine Lösung besitzt (Punkt 1 des Fixpunktsatzes).

d. Aus b. folgt auch, dass die kleinste obere Schranke s der Folge $(F^i)_{i \in \mathbb{N}}$ eine Lösung der Gleichung $g = F(g)$ ist.
Wir zeigen, dass s für die Approximationsordnung $\sqsubseteq$ über D kleiner als jede andere Lösung von $g = F(g)$ ist.

   Sei t eine Lösung von $g = F(g)$, d.h. $t = F(t)$ und $t \in D$.

   1. Wir zeigen durch vollständige Induktion, dass für jedes $i \in \mathbb{N}$ gilt: $F^i \sqsubseteq t$
      Basisfall: $F^0 = \perp_D \sqsubseteq t$.
      Nach Definition einer Domäne gilt $\perp_D \sqsubseteq d$ für jedes $d \in D$. Nach Annahme ist F monoton. Es folgt also: $F^0 = \perp_D \sqsubseteq t$.
      Induktionsfall:
      Induktionsannahme: Für ein $i \in \mathbb{N}$, $F^i \sqsubseteq t$.
      Wir zeigen nun, dass $F^{i+1} \sqsubseteq t$.

Fixpunktsatz von Kleene (IV)

Nach Induktionsannahme und da F stetig, also folglich auch monoton ist, gilt:

$$F(F^i(t)) \sqsubseteq F(t)$$

Nach Annahme gilt $t = F(t)$, also:

$$F^{i+1} = F(F^i(t)) \sqsubseteq t.$$

Der Induktionsfall ist also bewiesen.

2. Da nach Annahme s die kleinste obere Schranke der Kette $(F^i)_{i \in \mathbb{N}}$ ist, ist s kleiner als t für die Approximationsordnung, d.h. $s \sqsubseteq t$.

qed.

Der Fixpunktsatz besagt also, dass die Denotation einer rekursiven Funktionsdeklaration (als Gleichung betrachtet) ihr kleinster Fixpunkt ist.

Praktische Relevanz des Fixpunktsatzes

Der Fixpunktsatz ist nicht nur zur (math.) Formalisierung der Semantik der Rekursion vom Belang. Sein Beweis beruht auf einer Berechnungstechnik, die in vielen Bereichen der praktischen Informatik Anwendung findet.

Ein Verfahren, dass Lösungen berechnet, bis keine neue hinzukommen, kann als *Fixpunktberechnung* formalisiert werden. Solche Fixpunktberechnungen können endlich sein, wenn es endlich viele Lösungen gibt, oder unendlich sein, wenn es unendlich viele Lösungen gibt.

Auch wenn es unendlich viele Lösungen gibt, kann eine Fixpunktberechnung nützlich sein. Unter gewissen Voraussetzungen liefert eine Fixpunktberechnung jede einzelne Lösung nach endlicher Zeit, so dass sie zur Auswahl einer (nach gewissen Kriterien) passenden Lösung verwendet werden kann.

Beispiele solcher Fixpunktberechnungen sind u.a. bei Deduktionssystemen, deduktiven Datenbanksystemen und bei der Ähnlichkeitsuche in Datenbanken zu finden.