Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 5: Die vordefinierten Typen von SML

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

Der Begriff „Typ“

- Ein Typ (oder Datentyp) ist eine Menge von Werten.
- Mit einem Typ werden Operationen, oder Prozeduren, zur Bearbeitung der Daten des Typs angeboten:

  Eine n-stellige Operation über einem Typ T ist eine Funktion $T^n \rightarrow T$, wobei T^n für $T \times ... \times T$ (n Mal) steht.

Die ganzzahlige Addition ist z.B. eine binäre (d.h. zweistellige) Operation über dem Typ „ganze Zahl“.

Vordefinierte und selbst definierte Typen

- Ein Typ kann vordefiniert sein, d.h. von der Programmiersprache als Wertemenge angeboten werden:

  Mit solchen Typen werden die Operationen, Funktionen oder Prozeduren angeboten, die zur Bearbeitung von Daten des Typs üblich sind.

- In modernen Programmiersprachen können Programmierer über die vordefinierten Typen hinaus selbst Typen definieren.

Beispiele für selbst definierte Typen

- Typ „Wochentag: Wertemenge {Montag; Dienstag; …; Sonntag}
- Typ „Uhrzeit“: Wertemenge $\{h: m: s \mid h \in \mathrm{N}, 0 \leq h < 24, m \in \mathrm{N}, 0 \leq m < 60, s \in \mathrm{N}, 0 \leq s < 60\}$
- Typ „komplexe Zahl“:
  Wertemenge $\{a + i * b \mid a$ und b vom Typ „reelle Zahlen“$\}$
- Typ „Übungsgruppe“ zur Bündelung der folgenden Merkmale einer Übungsgruppe:
  - Der Name des Übungsleiters (als Zeichenfolge)
  - Die Namen der in dieser Übungsgruppe angemeldeten Studenten (als Zeichenfolgen)
  - Der wöchentliche Termin der Übung (als Paar aus Wochentag und Uhrzeit)
  - Der Ort, wo die Übungsgruppe sich trifft (als Zeichenfolge)

Ausblick

- Die Bildung von neuen Typen wird in den Kapiteln 8 und 11 behandelt.
- In diesem Kapitel werden die vordefinierten Typen von SML behandelt.
- Diese Typen von SML bilden eine sehr „klassische“ Auswahl an vordefinierten Typen, die sich von dem Angebot an vordefinierten Typen in anderen Programmiersprachen nur unwesentlich unterscheidet.

Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

Basistypen:

- „ganze Zahlen“: `int` (integer)
- „reelle Zahlen“: `real`
- „boole´sche Werte“: `bool`
- „Zeichenfolgen“: `string`
- „Zeichen“: `char`
- „null-stelliger Vektor“: `unit`

Ganze Zahlen

- Der SML-Typ `int` steht für die „ganzen Zahlen“.
- Das Minusvorzeichen der negativen ganzen Zahlen wird in SML ~ geschrieben: z.B. `~89`.
- Führende Nullen sind erlaubt: z.B. `007`, `~082`.

Binäre Operationen über `int`

`+`	(infix) Addition
`-`	(infix) Subtraktion
`*`	(infix) Multiplikation
`div`	(infix) ganzzahlige Division
`mod`	(infix) Rest der ganzzahligen Division

Vergleichsoperationen über `int`

`=`	(infix) gleich
`<>`	(infix) nicht gleich
`<`	(infix) echt kleiner
`<=`	(infix) kleiner-gleich
`>`	(infix) echt größer
`>=`	(infix) größer-gleich

- Die Vergleichsoperatoren von `int` sind Funktionen vom Typ `int * int` → `bool`.

Die Funktion `real` vom Typ `int` → `real` dient zur Konvertierung einer ganzen Zahl in eine Gleitkommazahl mit demselben mathematischen Wert.

Reelle Zahlen

- Der SML-Typ `real` bezeichnet die Gleitkommazahlen, die auch inkorrekterweise „reelle Zahlen“ genannt werden.
- Gleitkommazahlen stellen rationale Zahlen dar, aber nur eine endliche Teilmenge davon und mit Gesetzmäßigkeiten der Arithmetik, die stark von den mathematischen Gesetzmäßigkeiten abweichen können.

Mehr über Gleitkommazahlen erfährt man in Informatik 3.

Konstrukte zur Darstellung von Gleitkommazahlen

- Es 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 oder E-Notation zur Darstellung von Zehnerpotenzen:
  z.B. `123E5, 123.0E~3, 123.0e~3, ~0.899e4`
- In der Mantisse-Exponent-Notation kann das „`e`“ sowohl groß als auch klein geschrieben werden.

Führende Nullen und korrekte Schreibweise

- SML lässt führende Nullen in Dezimalbruchzahlen sowie in Mantissen und Zehnerexponenten zu.
- Vor dem Punkt einer Dezimalbruchzahl verlangt SML eine Ziffer:

  .89 ist nicht zulässig (es muss z.B. 0.89 heißen)

Binäre Operationen über `real`

`+`	(infix) Addition
`-`	(infix) Subtraktion
`*`	(infix) Multiplikation
`/`	(infix) Division

Arithmetik mit Gleitkommazahlen

- Die Arithmetik mit Gleitkommazahlen folgt ihren eigenen Gesetzmäßigkeiten und führt oft zu anderen Ergebnissen als die Arithmetik mit rationalen Zahlen oder gar reellen Zahlen im mathematischen Sinn:

```
- 1234567890.0 + 0.005;
val it = 1234567890.01 : real

- 1234567890.0 + 0.0005;
val it = 1234567890.0 : real
```

- Deshalb ergeben arithmetische Berechnungen mit Gleitkommazahlen im Allgemeinen bestenfalls Approximationen der tatsächlichen Werte.

Vergleichsoperationen über `real`

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

floor, ceil, trunc

- Die Funktion `floor` vom Typ `real`→ `int` konvertiert eine Gleitkommazahl in eine ganze Zahl und rundet sie dabei nach unten.
- Die Funktionen `ceil` vom gleichen Typ rundet nach oben.
- Die Funktion `trunc` vom gleichen Typ rundet in Richtung Null, indem sie einfach alle Nachkommastellen weglässt.

inf (*infinite*) nan (*not-a-number*)

- Der SML-Typ `real` enthält außer den „normalen“ Gleitkommazahlen noch einige 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
```

infinite: unendlich
not-a-number: kein Zahlenwert

Wie die letzten beiden Beispiele andeuten, sind alle Operationen des Typs `real` auch definiert, wenn diese speziellen Werte als Argument auftreten. Die Einzelheiten dieser Definitionen folgen einem internationalen Standard für Gleitkommazahlen in Programmiersprachen (IEEE standard 754-1985 und ANSI/IEEE standard 854-1987).

Bool´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

`andalso, orelse`

- 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).

Zeichenfolgen

- Der SML-Typ `string` ist die Menge der endlichen Zeichenfolgen.
- Zeichenfolgen werden eingeklammert zwischen zwei `"` geschrieben.
- `""` bezeichnet die leere Zeichenfolge.
- Das Zeichen `"` wird innerhalb einer Zeichenfolge `\"` geschrieben:
  z.B `"ab\"cd"` bezeichnet in SML die Zeichenfolge `ab"cd`.

Darstellung von Sonderzeichen

- Weitere „escape sequences“, die mit dem Zeichen \ anfangen, dienen zur Darstellung von Sonderzeichen:

`\n`:	newline
`\t`:	tab
`\\`:	\

Spezielle Notation mit \

- Die Notation

  \ *gefolgt von white-space-Zeichen gefolgt von* \

  ermöglicht es, sogenannte white-space-Zeichen wie newline, tab oder Leerzeichen innerhalb einer SML-Zeichenfolge zu ignorieren.
- Beispiele:

```
- "aaaa\        \b";
val it = "aaaab" : string

- "ccc\
=
=
= \d";
val it = "cccd" : string
```


Operationen über `string`

`size`	(präfix, unär) Länge einer Zeichenfolge
`^`	(infix, binär) Konkatenation (Aneinanderfügen) zweier Zeichenfolgen

Vergleichsoperationen über `string`

`=`	(infix) gleich
`<>`	(infix) nicht gleich
`<`	(infix) echt kleiner
`<=`	(infix) kleiner-gleich
`>`	(infix) echt größer
`>=`	(infix) größer-gleich

- Diese Operatoren beziehen sich auf die sogenannte lexikographische Ordnung, nach der zum Beispiel `"a" < "aa" < "b"` ist.

Zeichen

- Der Typ `char` besteht aus der Menge der Zeichen.
- Man beachte den Unterschied zwischen Zeichen und Zeichenfolgen der Länge 1:
  - Die einelementige Menge `{2}` ist nicht dasselbe wie die ganze Zahl `2`.
  - So ist das *Zeichen* `b` nicht dasselbe wie die *Zeichenfolge* `b`.
- In SML wird ein Zeichen `z` als `#"z"` geschrieben (`#` gefolgt von der Zeichenfolge `z`, also `"z"`).

Funktionen über `char`

- `chr` und `ord`:

  `chr: int → char`

  für `0 ≤ n ≤ 255` liefert `chr(n)` das Zeichen mit Code `n`

  `ord: char → int`

  liefert den Code eines Zeichens `z`

Beispiele zu `chr,ord`

```
- 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
```

Als Basis der Kodierung von Zeichen, d.h. der Zuordnung numerischer Codes zu Zeichen des Datentyps `char`, dient der ASCII-Code (American Standard Code for Information Interchange).

Der ASCII-Code enthält keine Buchstaben, die im amerikanischen Englisch ungebräuchlich sind, wie Umlaute oder ç, à.

Funktion `str, String.sub`

- Zur Konvertierung zwischen Zeichen und Zeichenfolgen bietet die Standardbibliothek von SML die unäre Funktion `str` und die binäre Funktion `String.sub` an:

```
- str(#"a");
val it = "a" : string
- String.sub("abcd", 0);
val it = #"a" : char
- String.sub("abcd", 2);
val it = #"c" : char
```


ASCII Tabelle

- Informationen über den ASCII Zeichensatz findet man unter:

http://www.ascii-table.com/

unit

- Der SML-Typ `unit` besteht aus der einelementigen Wertemenge `{()}`.
- Der Wert `()` wird oft *unity* ausgesprochen.
- Dieser einzige Wert des Typs `unit` wird als Wert von Prozeduraufrufen verwendet.

Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

Vektoren (Tupel)

- Sind $n \geq 0$ und $t_1, t_2, \ldots, t_n$ SML-Typen und $A_1, A_2, \ldots, A_n$ Ausdrücke der Typen $t_1, t_2, \ldots, t_n$, so ist $(A_1, A_2, \ldots, A_n)$ ein n-stelliger Vektor (n-Tupel) vom Typ $t_1 * t_2 * \ldots * t_n$.

Dabei bezeichnet „ * “ in SML das kartesische Produkt von Typen.

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.

Gleichheit über Vektoren der selben Länge

- Die Gleichheit 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
```


Gleichheit über Vektoren der Länge 0 und 1

- Ein einstelliger Vektor ist in SML mit seiner (einzigen) Komponente identisch:

```
- (3) = 3;
val it = true : bool
```

- Der 0-stellige Vektor `()` ist der (einzige) Wert des SML-Typs `unit`.

„Mehrstellige“ Funktionen

- In SML hängen Vektoren und Argumente von „mehrstelligen“ Funktionen so zusammen:

  In einer Funktionsanwendung `f(`a_1`,`a_2`,`a_3`)` stellt `(`a_1`,`a_2`,`a_3`)` einen Vektor dar, so dass die Funktion `f` einstellig ist (und auf dreistellige Vektoren angewandt wird).

Beispiel

```
- 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!

Selektion einer Komponente mit Pattern Matching

- Zur Selektion der Komponenten eines Vektors kann in SML der Musterangleich (*Pattern Matching*) verwendet werden:

```
- 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 einer Komponente aus einem Vektor

- Zur Selektion der Komponenten eines Vektors können in SML auch die Funktionen #1, #2, ... verwendet werden:

```
- #1("a", "b", "c");
val it = "a" : string

- #3("a", "b", "c");
val it = "c" : string
```

Deklaration eines Vektortyps

t_1 * t_2 * … * t_n wird so ein Name gegeben:

```
- 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
```


Bemerkungen zur Deklaration eines Vektors

- `(real * real) * (real * real)` ist der Typ des aktuellen Parameters der vorangehenden Funktionsanwendung (die Funktion ist einstellig, und ihr Argument ist ein Paar von Paaren von Gleitkommazahlen).
- Erinnern Sie sich an die Übungsaufgabe 2-2: Ein Rechteck wurde durch ein Paar von Punkten angegeben. Den Typ `rechteck` könnten wir nun auch explizit spezifizieren:
  - `type rechteck = punkt * punkt;`

Verbunde (records) (1)

- Ein *n*-stelliger Vektor ist eine geordnete Zusammensetzung von *n* Komponenten, so dass die Komponenten durch ihre Position bestimmt werden.
- Man kann einen dreistelligen Vektor vom Typ `string*char*int`, wie z.B. `("Meier", #"F", 2210)`, als eine Zusammensetzung von einer Zeichenfolge, einem Zeichen und einer ganzen Zahl beschreiben, so dass gilt:
  - die ganze Zahl hat die Position 3,
  - die Zeichenfolge hat die Position 1,
  - das Zeichen hat die Position 2.

Verbunde (records) (2)

- Anstelle von nummerierten Positionen kann man Bezeichner verwenden:
  - Nachname
  - Vornamenbuchstabe
  - Durchwahl

Diese Idee liegt den Verbunden (*records*) zu Grunde.
(Nebenbei: Verbunde sind den „structures“ der Programmiersprache C und den „records“ der Programmiersprachen Pascal und Modula ähnlich)

Beispiel mit SML – Verbund

```
- val adressbucheintrag =
{Nachname = "Meier", Vornamenbuchstabe = #"F",
Durchwahl = "2210"};

val adressbucheintrag =
{Durchwahl = "2210", Nachname = "Meier",
  Vornamenbuchstabe = #"F"}
  : {Durchwahl : string, Nachname : string,
  Vornamenbuchstabe : char}
```

Die Reihenfolge der Komponenten eines Verbunds spielt keine Rolle!
Das folgt aus der Identifizierung der Komponenten eines Verbundes mit Bezeichnern statt mit Positionen wie bei Vektoren.

Bezeichner im Typ des Verbunds

- Die Bezeichner der Komponenten eines Verbundes kommen im Typ des Verbundes vor.
- 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}
```

Vergleich von Verbunden Selektion von Komponenten

- Verbunde werden komponentenweise verglichen:

```
- {a=1, b=2} = {b=2, a=1};
val it = true : bool
```

- Zur Selektion der Komponentenwerte mit Hilfe der Komponentenbezeichner eines Verbundes bietet SML die Funktion *#Bezeichner* an:

```
- #bbb({aaa = 1, bbb = 2});
val it = 2 : int

- #a({a = 1, b = 2});
val it = 1 : int
```


Kurzschreibweise für Deklarationen in SML (1)

```
- val information =
  {Nachname = "Meier", Vorname = "Franz"};
val information =
{Nachname="Meier", Vorname="Franz"}
: {Nachname:string, Vorname:string}

- val {Nachname, Vorname} = information;
val Nachname = "Meier" : string
val Vorname = "Franz" : string
```

- So werden Variablen deklariert, die dieselben Namen wie die Komponentenbezeichner haben.

Kurzschreibweise für Deklarationen in SML (2)

- Folgendes ist nicht zulässig:

```
- val {nn, vn} = information;

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 information
           of {nn=nn,vn=vn} => (nn,vn))
```


Deklaration eines Vektor- oder Verbundstyps

- Wie für Vektoren gibt es in SML die Möglichkeit einem Verbundtyp einen Namen zu geben, z.B.:
  - `type complex = real * real;`
  - `type dateieintrag = {Vorname:string, Nachname:string};`
- Der Name ist lediglich ein Synonym für den Verbundtyp.

Vektoren als Verbunde

- In SML sind Vektoren Verbunde mit besonderen Komponentenbezeichnern:

```
- {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
```


Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

Ausblick über den Abschnitt

- Der Begriff „Liste“ kommt in den meisten Programmiersprachen und in vielen Algorithmen vor (mit einigen unwesentlichen Unterschieden vor allem in der Syntax).
- Zunächst wird der Begriff „Liste“ unabhängig von jeglicher Programmiersprache erläutert.
- Dann wird der SML-Typ „Liste“ eingeführt.

Der Begriff „Liste“

- Betrachte den Begriff „Liste“ in Algorithmen-spezifikations- und Programmiersprachen:
  - Eine Liste ist eine endliche geordnete Folge von Elementen.
  - Listen werden oft so dargestellt:
    `[1, 2, 3]` oder `["a", "bcd", "e", "fg"]`.
  - Es gibt eine leere Liste: `[ ]` .

Die Funktion `cons`

- die Funktion `cons` („*list constructor*“) bildet Listen:

  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.
- Angewandt auf `5` und die Liste `[9,8]` bildet `cons` die Liste `[5,9,8]`.

  →`[5,9,8]` ist der Wert von `cons(5,[9,8])`.
- Die Funktion `cons` wird oft infix geschrieben:

  `5 cons [9, 8]` (statt `cons(5, [9, 8])`)

Vereinfachte Schreibweise

- Aus der Definition von `cons` folgt, dass eine Liste wie `[5,9,8]` auch `5 cons(9 cons(8 cons[]))` notiert werden kann.
- Ist zudem die Infixfunktion `cons` rechtsassoziativ (wie in vielen Programmiersprachen), so kann eine Liste wie `[5,9,8]` auch notiert werden als:

  `5 cons 9 cons 8 cons []`
- Die meisten Programmiersprachen bieten eine Notation wie `[5,9,8]` als Ersatz für den weniger lesbaren Ausdruck `5 cons 9 cons 8 cons []`.

Die Funktionen `head, tail`

- Mit diesen zwei Funktionen können auf Werte aus einer Liste zugegriffen werden:
  - 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.

    (Angewandt auf `[5,9,8]` liefern also `head` den Wert `5` und `tail` den Wert `[9,8]`)
- Die Funktionen `head` und `tail` sind auf Listen nicht total, weil sie für die leere Liste nicht definiert sind.

Das letzte Element einer Liste

- Mit `head`, `tail` und Rekursion lässt sich eine Funktion spezifizieren, die das letzte Element einer nichtleeren Liste liefert:

  Das letzte Element `E` einer nichtleeren Liste `L` ist definiert als:
  - Falls `L` eine 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 spezifizieren mit:

  Eine Liste `L` enthält (genau) ein Element genau dann, wenn `tail(L) = []` ist.

Allgemeines über Listen

- Listenelemente können die Werte von atomaren oder zusammengesetzten Ausdrücken sein:

  Der Wert des Ausdrucks `[eins,zwei]` ist die Liste `[1,2]`, wenn `1` der Wert von `eins` und `2` der Wert von `zwei` ist.

- Listen von Listen sind möglich:

  `[[1,2],[1,5]]`.

- Die Gleichheit für Listen ist elementweise definiert:

  `[a,b,c] = [d,e,f]` genau dann,

  wenn `a = d` und `b = e` und `c = f` gilt.

Listen in SML

- Eine SML-Liste ist eine endliche Folge von Werten desselben Typs.
- In SML ist es nicht möglich, Listen von Werten aus verschiedenen Typen zu bilden.
- Für jeden gegebenen Typ `'a` (`'a` wird oft *alpha* ausgesprochen) bezeichnet in SML `'a list` den Typ der Listen von Werten vom Typ `'a` .
- Ist z.B. `'a` der Typ `int`, so ist `int list` der Typ, der aus den Listen von ganzen Zahlen besteht.

Notation für Listen

- SML bietet zwei Notationen für Listen:
  1. mit dem Listkonstruktor `cons`, der in SML „ `::` “ geschrieben wird und rechtsassoziativ ist, und der leeren Liste, die in SML `nil` geschrieben wird.
  2. unter Verwendung der Listenklammern „ `[]` “ mit Komma „ `,` “ als Trennzeichen zwischen den Listenelementen.

Diese Notation ist eine Kurzform für die Notation mit dem Listenkonstruktor `::` (`cons`).

Beispiel zur ersten Notation

- In der ersten Notation kann die Liste der ersten vier natürlichen Zahlen geschrieben werden als:

```
0 :: (1 :: (2 :: (3 :: nil))),
```

d.h. dank der Rechtsassoziativität von `::` auch als:

```
0 :: 1 :: 2 :: 3 :: nil
```

```
- 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
```

Beispiel zur zweiten Notation

- In der zweiten Notation werden die Liste der ersten vier natürlichen Zahlen als `[0,1,2,3]` und die leere Liste als `[]` dargestellt:

```
- [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
```


Der SML –Typ der leeren Liste

- 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
```

- `'a` ist eine Typvariable (eine Variable, die als Wert einen Typ erhalten kann).
- Man sagt, dass `nil` ein „*polymorphes Objekt*“ ist (es gehört mehreren Typen an).
- Das hat den Vorteil, dass es nur *eine* leere Liste gibt.
- Wäre `nil` kein *polymorphes Objekt*, dann müsste es für jeden möglichen Typ `'a` eine leere Liste für den Typ `'a list` geben!

Gleichheit über Listen

- 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
```


Mono- und Polytypen

- Ein Typausdruck wie `'a` oder `'a list` wird „polymorpher Typ“ oder „*Polytyp*“ genannt, weil ein Ausdruck für mehrere (griechisch „poly“) Typen steht: Mögliche Instanzen von `'a` sind z.B.:
  - `int`
  - `bool`
  - `int list`
- Mögliche Instanzen von `'a list` sind z.B.:
  - `int list`
  - `bool list`
  - `(int list) list`
  - `(int * bool) list.`
- Ein Typ, der kein Polytyp ist, wird „*Monotyp*“ genannt.

Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

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
```

Letztes Element einer nicht leeren Liste (1)

```
- 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
```


Letztes Element einer nicht leeren Liste (2)

- Das SML-System erkennt, dass die Deklaration der Funktion `letztes_element` keinen Fall für die leere Liste hat, d.h. dass diese Funktion über einem Typ `'a list` nicht total ist.
- Deshalb gibt es eine Warnung, da nichttotale Funktionen manchmal fehlerhaft sind.
- Da aber die Deklaration einer nichttotalen Funktion in manchen Fällen - wie hier - notwendig ist, wird eine solche Deklaration nicht abgelehnt.

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
```


n-tes Element aus einer Liste

```
- fun ntes_element(1, x :: _) = x
    | ntes_element(n, _ :: L) =
                        ntes_element(n-1, L);
```

Über welcher Menge ist diese Funktion total?

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
```


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
```

Erinnerung an Kapitel 1

```
- val max = fn (x, y, ord) =>
    if ord(x, y) then x
    else y;
- val rec groesstesElement =
    fn (liste, ord) =>
    if length(liste) = 1 then hd(liste)
    else
      max(hd(liste),
          groesstesElement(tl(liste), ord),
          ord);
```


append

- Die vordefinierte SML-Funktion append, infix „@“ notiert, fügt zwei Listen aneinander:

```
- [1,2,3] @ [4,5];
val it = [1,2,3,4,5] : int list
```

- append kann in SML so 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
```

Beispiel zu den Berechnungsschritten von `append`

- 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`

- Man kann als Zeiteinheit die Anzahl der Aufrufe der Funktion `cons` (`::`) oder die Anzahl der rekursiven Aufrufe von `append` wählen.
- Beide Zahlen stehen 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 `append` die Funktion `cons` (`::`) genau n mal auf.
- Die Länge des zweiten Parameters `L'` beeinflusst den Zeitbedarf von `append` nicht.
- Ist n die Länge des ersten Parameters von `append`, so gilt: `append` $\in O(n)$.

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
```

- „*reverse*“ kann in SML so 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
```


Berechnungsschritte

- 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` (1)

- Zur Schätzung der Größe des Problems „Umkehrung einer Liste“ kann man die Länge der Liste wählen.
- Wie zur Schätzung des Zeitbedarfs der Funktion `append` kann man als Zeiteinheit die Anzahl der rekursiven Aufrufe von `naive_reverse` oder die Anzahl der Aufrufe der Funktion `cons` (`::`) wählen.

Zeitbedarf von `naive_reverse` (2)

- Gegeben sei eine Liste `L` der Länge n mit $n \geq 1$.
- Während des Aufrufs von `naive_reverse(L)` wird die Funktion `naive_reverse` $n + 1$ mal rekursiv aufgerufen.
- Zuerst mit einer Liste der Länge n - 1 als Parameter, dann mit einer Liste um ein Element kürzer als Parameter bei jedem weiteren Aufruf.
- Wegen (*) (siehe die Schätzung des Zeitbedarfs von `append`) wird zur Zerlegung der Eingabeliste die Funktion `cons` (`::`) also $(n - 1) + (n - 2) + \ldots + 1$ mal aufgerufen.
- Zum Aufbau der zu berechnenden Liste wird `cons` (`::`) zudem n mal aufgerufen.

Zeitbedarf von `naive_reverse` (3)

- Die Gesamtanzahl der Aufrufe von `cons` (`::`) lautet:

  $n + (n - 1) + (n - 2) + \ldots + 1 = n * (n + 1) / 2$

- Ist n die Länge des Parameters von `naive_reverse`, so gilt:

  `naive_reverse` $\in O(n * (n + 1) / 2)$
- also

  `naive_reverse` $\in O(n^2)$

`reverse`

- Der quadratische Zeitbedarf von `naive_reverse` ist nicht zufriedenstellend, weil eine Liste in einer Zeit in umgekehrte Reihenfolge gebracht werden kann, die linear von der Listenlänge abhängt.
- Idee: Man fängt mit zwei Listen an:
  - die linke Liste ist die Eingabeliste
  - die rechte Liste ist anfangs leer

  Das erste Element der linken Liste wird von dieser Liste entfernt und am Anfang der rechten Liste eingefügt.

  Mit den weiteren Elementen wird genauso verfahren.

  Dabei werden nur Operationen verwendet, die der Typ `Liste` anbietet.

  Nach so vielen Schritten, wie die Eingabeliste lang ist, ist die linke Liste leer und die rechte die Liste in umgekehrten Reihenfolge.

Veranschaulichung der Idee

Schritt	Linke Liste	Rechte Liste
0	[1,2,3]	[]
1	[2,3]	[1]
2	[3]	[2,1]
3	[]	[3,2,1]

`reverse` in SML: `aux_reverse`

```
- 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
```

`reverse` in SML

- Die gewünschte einstellige `reverse`-Funktion folgt unmittelbar aus der Spezikation von `aux_reverse`:

```
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

- 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)` insgesamt n rekursiver Aufrufe sowie n Aufrufe der Funktion `cons` (`::`).
- Es gilt also: `aux_reverse` $\in O(n)$
- Folglich gilt auch: `reverse` $\in O(n)$

Bemerkung zu `reverse`

- Der zweite Parameter der Funktion `aux_reverse`, der der rechten Liste aus unserem Beispiel entspricht, ist ein sogenannter *Akkumulator*.
- Die Nutzung eines Akkumulators wurde bereits im Kapitel 4 erläutert, um die Berechnung der Fakultät einer natürlichen Zahl in einem iterativen Prozess zu berechnen.
- Wie die Funktion `fak_funktional` ist die Funktion `reverse` endrekursiv, so dass sie einen iterativen Berechnungsprozess auslöst.

Überblick

1. Was sind Typen?
2. Die Basistypen von SML
3. Zusammengesetzte Typen in SML
4. Listen
5. Beispiele: Grundlegende Listenfunktionen
6. Hinweis auf die Standardbibliothek von SML

Standardbibliothek von SML

- Die Standardbibliothek von SML, die unter der URI http://www.smlnj.org/doc/basis/

  zugreifbar ist, bietet für die Basistypen von SML Funktionen an, die herkömmliche Operationen über diesen Typen in SML implementieren.