1 Grundbegriffe

1.1 Datentypen

Unter einem *Datentyp* oder kurz *Typ* (im dt. auch häufig *Rechenstruktur*) versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit. Liegt der Schwerpunkt auf den Eigenschaften, die die Operationen und Wertebereiche besitzen, so spricht man von *abstrakten Datentypen*; steht dagegen die Realisierung in einer Programmiersprache im Vordergrund, so handelt es sich um *konkrete Datentypen*.
Der Zweck von Datentypen besteht in der Zerlegung des Universums der in einer Programmiersprache vorkommenden Werte in Klassen mit gemeinsamen Merkmalen bezgl Darstellung und erlaubten Operationen.

Datentypen werden oft aus primitiven Datentypen mithilfe gewisser Konstruktionsprinzipien gebildet. Die primitiven Datentypen bezeichnet man dabei als **elementare Datentypen** oder **Grundtypen**. Die wichtigsten elementaren Datentypen sind:

Name	Wertebereich	Operationen
bool	B	true, false, and, or, not, =, ≠
nat	$\mathbb{N}_0$	0, succ, +, -, ∗, div, mod, =, ≠, <, >, ...
int	Z	0, 1, +, -, ∗, div, mod, =, ≠, <, >, ...
real	IR	0, +, -, ∗, /, sin, cos, tan, ln, exp, sqrt, =, ≠, <, >, ...
char	{"a",...,"z","A",...,"Z", "$","∗",...,"0","1",...,"9"}	·, ord, chr, ...

Die übrigen Datentypen nennt man **zusammengesetzte Datentypen**, **strukturierte Datentypen** oder kurz **Datenstrukturen**. Interessant ist dabei, daß einige wenige Konstruktionsvorschriften (sog. **Konstruktoren**) ausreichen, um beliebig umfangreiche Datenstrukturen aufzubauen. Die wichtigsten Konstruktoren sind:

- Enumeration (=Bildung von Aufzählungstypen)
- Restriktion (=Einschränkung der Wertemenge eines Datentyps, Vererbung der für den Obertyp zugelassenen Operationen auf den eingeschränkten)
- Aggregation, Produktbildung (=Bildung des kartesischen Produkts von Datentypen)
- Generalisation, Summenbildung (=Vereinigung von disjunkten Datentypen)
- Rekursion (=Übergang zu abzählbar unendlichen Wertebereichen, deren Elemente rekursiv aus einfacheren Elementen des Wertebereichs aufgebaut sind)
- Potenzmengenbildung (=Übergang zur Potenzmenge der Wertemenge eines Datentyps)
- Funktionsräume, Potenzbildung (=Übergang zur Menge aller Abbildungen von einem Datentyp in einen anderen; dieser Konstruktor spielt im Rahmen der funktionalen Programmierung eine herausragende Rolle).

Diese Konstruktoren werden wir später ausführlicher im Zusammenhang mit der funktionale Programmiersprache ML behandeln, soweit sie dort gegeben sind.

Bei der Definition eines Datentyps mittels eines Konstruktors werden meist implizit gewisse Funktionssymbole mitdefiniert, die den Konstruktionsprozeß wieder rückgängig machen und es ermöglichen, einen in einer Datenstruktur versteckten elementaren Wert "wiederaufzuspüren". Solche Funktionen nennt man **Selektoren**.

1.2 Typisierung

In diesem Abschnitt betrachten wir zwei jeweils konträre Aspekte des Typkonzepts von Programmiersprachen: strenge versus schwache Typisierung resp. statische versus dynamische Typisierung.

Definition A:
Eine Programmiersprache heißt *streng typisiert*, wenn in jedem Programm der Sprache alle Größen zu genau einem Datentyp gehören, andernfalls *schwach typisiert*.

Die Wertemengen aller Datentypen einer streng typisierten Sprache sind also paarweise disjunkt. Eine Ausnahme bilden lediglich Typen, die durch Restriktion eines anderen Typs hervorgegangen sind.
In streng typisierten Sprachen können auf die Objekte eines Typs nur diejenigen Operationen angewendet werden, die für den Typ definiert sind, und sonst keine.

Beispiel:
In einer streng typisierten Sprache sind folgende Konstruktionen *nicht* erlaubt, in einer schwach typisierten sind sie erlaubt:

- der Ausdruck $\sin(7)$, da $\sin$: real→real definiert ist, 7 jedoch ein int-Ausdruck ist;
- der Ausdruck $x+\underline{\text{true}}$, da die Addition die Funktionalität int×int→int oder real×real→real besitzt, zumindest der zweite Parameter $\underline{\text{true}}$ jedoch weder zu int noch zu real gehört. In einer schwach typisierten Sprache stellt sich erst während der Laufzeit des Programms heraus, ob die rechnerinternen Darstellungen von x und $\underline{\text{true}}$ Zahlen entsprechen, die addiert werden können;
- der Ausdruck $17*12.5$, da die Multiplikation die Funktionalität int×int→int oder real×real→real besitzt, jedoch in beiden Fällen einer der beiden Parameter den falschen Typ besitzt.
  Dieser Ausdruck läßt sich korrigieren, indem man auf 17 eine Funktion zur *Typkonversion* anwendet, z.B.
  makereal: int→real.

Einige Programmiersprachen, z.B. PASCAL, besitzen automatische Typanpassungen.

Strenge Typisierung erhöht bei der Software-Entwicklung i.a. die Sicherheit, da Typverletzungen bereits bei der Übersetzung erkannt und entsprechende Programme vom System abgewiesen werden. Zudem besitzen die Programme eine größere Portabilität, da sie keinen Bezug auf rechnerinterne Darstellungen nehmen können.

Definition B:
Eine Programmiersprache heißt *statisch typisiert*, falls alle oder die meisten Typüberprüfungen zur Übersetzungszeit durchgeführt werden können. Werden die Typprüfungen während der Laufzeit des Programms durchgeführt, so spricht man von *dynamischer Typisierung*.

Während LISP eine klassische Programmierpsrache mit dynamischer Typisierung ist, ist ma in den letzten Jahren mehr und mehr zu Programmiersprachen mit statischer Typisierung übergegangen. Denn einerseits erhöht das die Lesbarkeit und Übersichtlichkeit der Programme, und es steigert die Effizienz, da in das übersetzte Programm kein Code für die Typprüfungen eingebunden werden muß, andererseits stellen aktuelle funktionale Programmiersprachen leistungsfähige *Typinferenzsysteme* bereit, die aus den meisten Ausdrücken den zugehörigen Datentyp ableiten können, so daß Typdeklarationen in vielen Fällen unterbleiben können.