Einführung in die Informatik II

Sommersemester 2002, Volker Claus

2. Datenstrukturen

Manches in diesem Kapitel kennen Sie bereits aus der Vorlesung Einführung in die Informatik I von Prof. Lagally!

2.1 Mengen und elementare Typen

Die meisten Probleme kann man mit Hilfe von Mengen und auf ihnen definierten Operationen und Relationen beschreiben.

"Elementare" Mengen: Zeichen, Wahrheitswerte, natürliche Zahlen, ganze Zahlen, reelle Zahlen.

Symbol	Menge	Ada-Bezeichnung
A	Menge der (Tastatur-) Zeichen	Character
IB	{false, true}	Boolean
$\mathbb{N}_0$	{0, 1, 2, 3, 4, ...}	(zum Teil:) Natural
9	{..., -2, -1, 0, 1, 2, 3, ...}	(zum Teil:) Integer
IR	Menge der reellen Zahlen	(zum Teil:) Float

Hinzu kommen selbstdefinierte Mengen (Aufzählungstyp).

Auf diesen Mengen gibt es Operationen. In der Programmier-sprache Ada werden u.a. folgende verwendet:

Menge der Zeichen (Character): **A**
Die Menge umfasst 256 Zeichen (darstellbar durch ein Byte). Die Elemente sind (entsprechend der 8-Bit-Reihenfolge) angeordnet. Daher gibt es die Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Prinzipiell können auch bei Character die für Aufzählungstypen gültigen Funktionen verwendet werden:

pred(X) = das Zeichen vor X in der Character-Reihenfolge
succ(X) = das Zeichen nach X in der Character-Reihenfolge
pos(X) = das wievielte Zeichen ist X in der Character-Reihenfolge? (beginnend mit 0)
val(n) = n-tes Zeichen (beginnend mit 0)

Endliche selbstdefinierte Mengen: Aufzählungen
Wir erläutern dies am Beispiel. Es soll die Menge
D = {a, A, Do, 365, 7, xyz, Quadrat} eingeführt werden:
type mengeD **is** (a, A, Do, 365, 7, xyz, Quadrat);

Die Elemente sind entsprechend der Auflistung angeordnet, also a < A < Do < 365 < 7 <xyz < Quadrat. Zugleich sind die Elemente durchnummeriert, beginnend mit 0.
Es gibt die sechs Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Für Aufzählungstypen gibt es weiterhin folgende Funktionen:
pred(X) = das Element vor X in der Auflistung
succ(X) = das Element nach X in der Auflistung
pos(X) = das wievielte Element ist X in der Auflistung?
val(n) = n-tes Element (beachte: beginnend mit 0)

Menge der Wahrheitswerte (Boolean): IB

Konstanten sind **true** und **false**.
Es gibt die üblichen logischen Operatoren:
Negation: **not**,
Konjunktion: **and**, Disjunktion: **or**, Exklusives Oder: **xor**.

Zusätzlich stehen and und or auch als „Kaskadenoperationen“ **and then** und **or else** zur Verfügung, die man mit Vorsicht verwenden sollte (z.B. wegen Fehlern und exceptions):

a **and then** b entspricht:
falls a nicht zutrifft, ist das Ergebnis false, anderenfalls b.

a **or else** b entspricht:
falls a zutrifft, ist das Ergebnis true, anderenfalls b

Menge der ganzen Zahlen (Integer): 9

Einschränkung: Es gibt eine größte und eine kleinste darstellbare Zahl, die implementierungsabhängig sind, aber mindestens $\pm 2^{15}$ betragen. Es kann also nur dieser endliche Bereich verwendet werden. Damit sind dann auch die Operationen nicht mehr assoziativ, wenn man die Bereichsgrenzen überschreitet.

Die üblichen Operationen: Negatives eines Zahl -, Absolutbetrag **abs**, Addition +, Subtraktion -, Multiplikation *, ganzzahlige Division /, Rest bei Division (**mod** oder **rem**), Exponentiation ** (rechts muss eine natürliche Zahl stehen).
Es gibt die sechs Vergleichsoperatoren:
=, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Menge der natürlichen Zahlen (Natural): $\mathbb{N}_0$

In Ada werden die natürlichen Zahlen als Untertyp der ganzen aufgefasst, d.h., es gilt:

subtype Natural **is** Integer **range** 0..Integer‘Last;
Somit gibt es auch eine größte darstellbare natürliche Zahl.

Addition +, Subtraktion - (sofern das Ergebnis nicht negativ ist), Multiplikation *, ganzzahlige Division /, Rest bei Division (**mod**), Exponentiation **. Es gibt die sechs Vergleichsoperatoren wie bei Integer.

Menge der reellen Zahlen (Float): $\mathbb{R}$

Da es nur endlich lange Zahldarstellungen im Rechnern gibt, lässt sich nur eine Teilmenge der rationalen Zahlen darstellen. Diese Zahlen werden durch die jeweilige Zahldarstellung (z.B. Vorzeichen, Mantisse, Exponent) und die Anzahl der verwendeten Dezimalstellen festgelegt. (Eine Darstellung zu einer beliebigen Basis m>2 ist möglich.)

Kann hierbei der Dezimalpunkt an unterschiedlichen Stellen stehen, so spricht man von der floating point (Gleitpunkt- oder Gleitkomma-) Darstellung. bei fester Position des Dezimalpunkts liegt die fixed point (Festkomma-) Darstellung vor.

In Ada kann man auch die Zahl der Ziffern und die minimal erforderliche Genauigkeit vorgeben (Sprachelemente: digits und delta).

Für Gleitkomma- und Festkommadarstellungen gibt es die Operationen unäres + und -, Absolutbetrag, Addition +, Subtraktion -, Multiplikation *, Division / und Exponentiation ** (rechts von ** muss eine ganze Zahl stehen).
Es gibt die sechs Vergleichsoperatoren: =, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist.

Hinweis: Verwenden Sie "=" oder "/=" nicht bei Gleitkommazahlen, da wegen der Rundungsfehler kein korrektes Ergebnis erwartet werden kann.

Hinweis: Beachten Sie, dass das Ergebnis der Operationen vom Typ der Operanden abhängt und in der Regel keine automatische Typanpassung erfolgt! (Stichwörter: Überladen, Typkonversion.)

2.2. Datenkonstruktoren

Aus Basisdaten lassen sich neue Datenbereiche zusammensetzen. Typische "Konstruktoren" sind:

Kartesisches Produkt und Vektorräume bilden.

Mengen disjunkt vereinigen.

Potenzmenge bilden.

Menge der endlichen Folgen bilden ("freies Monoid").

Menge der Abbildungen zwischen Mengen bilden.

Zusammengesetzte Mengen: kartesisches Produkt, Vereinigung, Vektoren, freie Monoide (Wortmengen), Potenzmenge

Operation	Ada-Bezeichnung
$M_1 \times M_2$	**type** <name> **is record** <sel_1> : <Typ_1>; <sel_2> : <Typ_2>; **end record**;
$M_1 \cup M_2$	**type** <name> **is record** **case** <name> **is** **when** <name1> => <sel_1> : <Typ_1>; **when** <name1> => <sel_2> : <Typ_2>; **end case**; **end record**; (sog. varianter record)
M^n	**type** <name> **is array** (1..n) **of** <Typ>;
M^*	(das freie Monoid wird über Pointer dargestellt)
2^M	(die Potenzmenge lässt sich nicht direkt darstellen)

2.3. Aufbau von Programmiersprachen

Wenn Sie eine Programmiersprache definieren müssten ...

1. Elementare Datentypen:

   Festlegung der "atomaren" Mengen, also derjenigen Mengen, die in der Sprache zur Verfügung stehen und die nicht mehr aus kleineren Mengen aufgebaut werden sollen.
   Festlegung der auf ihnen zulässigen Operationen. Für jede dieser Mengen formuliere man einen elementaren Datentyp einschl. möglicher Beschränkungen durch die Implementierung.

   Beispiel: Menge der natürlichen Zahlen. Operationen +, *, >, =, mod, ... Typname: Natural. Die maximal darstellbare Zahl ist Natural'Last; die Operationen sind daher auf den Bereich (0..Natural'Last) einzuschränken. Zusätzlich kann man elementare Datentypen durch Aufzählung ihrer Elemente einführen; Operationen müssen dann über Programmstücke (Prozeduren, Funktionen) festgelegt werden.

2. Datenstrukturen:

   Festlegung der zulässigen Konstruktoren, um aus Mengen neue Mengen zu gewinnen, z.B. record, case, array, access, Subtyp.
   Festlegung, wie man auf die einzelnen Komponenten zugreifen kann. Lege weiterhin fest, welche neuen Operationen für diese neuen Strukturen definiert sein sollen. Gib die erforderlichen Beschränkungen an.

   Beispiel: Bilde zu einer Menge seine Potenzmenge, also die Menge aller Teilmengen. Mengenkonstruktor: set of <Menge>. Neue Konstanten und Operationen: leere Menge, in (Element-Beziehung), Enthalten-Sein, Komplement, Vereinigung, Durchschnitt, Anzahl der Elemente, ...
   Beschränkung: Es sind nur Teilmengen der Grundmenge <Menge> zugelassen, die höchstens ... Elemente besitzen.
   [Diese Datenstruktur "Potenzmenge" ist in Ada nicht zugelassen; sie kann aber über Bitvektoren, Bäume usw. simuliert werden.]

3. Ausdrücke:

   Festlegung der zulässigen Ausdrücke und ihrer Ergebnistypen. In der Regel baut man diese (wie logische oder arithmetische Ausdrücke) rekursiv aus den bereits festgelegten Operationen auf, wobei man Verträglichkeiten genau definieren muss. Die erforderlichen Beschränkungen sind anzugeben.

   Beispiel: Ganzzahlige Ausdrücke (Integer Expression, abgekürzt *IntExp*):

   a. Jede Variable und Konstante vom Typ Integer oder vom Typ Natural ist ein *IntExp*.
   b. Jeder Funktionsaufruf mit dem Ergebnistyp Integer oder Natural ist ein *IntExp*.
   c. Wenn P ein *IntExp* ist, dann auch (P), +P, -P, **P und abs(P).
   d. Wenn P und Q *IntExp* sind, dann auch (P+Q), (P-Q), (P*Q), (P/Q), (P mod Q) und (P rem Q).
   e. Alle Ausdrücke in *IntExp* müssen durch die Regeln a. bis d. herleitbar sein.

   Ergebnistyp ist Integer; er ist jedoch Natural, falls nur Natural-Typen an den Operationen beteiligt waren und der Natural-Bereich während der Berechnungen nicht verlassen wurde.

4. Elementare Anweisungen:

   In imperativen Sprachen sind dies:

- Leere Anweisung ("skip" oder ein allein stehendes Semikolon).
- Wertzuweisung "X := ß;" wobei ß ein Ausdruck ist, dessen Ergebnistyp gleich dem Typ der Variablen X ist.
- Prozeduraufruf.
- Sprung (sofern in der Sprache zugelassen; "goto").

Hinzu kommen spezielle Anweisungen wie exit (Beendigung von Schleifen), return (expliziter Rückkehr aus Einheiten), raise (Aktivieren einer Ausnahmebehandlung), delay, code, abort usw.

5. Kontrollstrukturen:

Die elementaren Anweisungen werden mit Konstruktoren zu neuen Anweisungen zusammengesetzt.
Es seien a, a_1, a_2 Anweisungen und b eine Bedingung (= Ausdruck vom Ergebnistyp Boolean).

- Hintereinanderausführung: a_1 **;** a_2 (Semikolon).
- Alternative: **if** b **then** a_1 **else** a_2 **end if**;
- Auswahl: **case** <ausdruck> **is**
  when <auswahl$_1$> => a_1;
  when <auswahl$_2$> => a_2; ...
  when <auswahl$_n$> => a_n; **others** => a_{n+1}; **end case**;
- Laufschleife: **for** K **in** (1..n) **loop** a **; end loop;**
- Schleife: **while** b **loop** a **;end loop;**
- Ausnahme: **raise** ..., bezogen auf ein **exception** ... **end**;
- Nichtdeterminismus: **select** ... **or** ... **else** ... **end select**;
- Nebenläufigkeit, Synchronisation (Taskaufruf, entry, accept, ...)

Achten Sie bei der Definition der Kontrollstrukturen darauf, dass mit ihnen die bereits definierten Datenstrukturen möglichst optimal durchlaufen werden können. Mit obigen Kontrollstrukturen wird genau dies angestrebt:

- Die Auswahl *case ... end case* korrespondiert mit der Datenstruktur record .... end record und zugleich mit dem "varianten" record (Vereinigung von Mengen).
- Die Laufschleife *for K in (1..n) loop a ; end loop* erlaubt das Durchlaufen von arrays.
- Die Schleife *while b loop a ;end loop* ermöglicht den Durchlauf durch eine Folge von Elementen (Listen).

6. Zusammenfassung zu Einheiten:

Algorithmen und Daten können zu einer Einheit zusammengefügt und anschließend als Ganzes genutzt werden. *Parametrisierungen* sind erlaubt; diese können Daten, Typen, Einheiten, ... sein.

Einheiten können sein: Prozeduren und Funktionen, Moduln (Pakete: Spezifikation und Rumpf), Prozesse (Tasks), Objekte und Klassen mit Vererbungen und Interaktionen. Hinzu kommen Bibliotheken, in denen ausformulierte Einheiten stehen, die in andere Programme eingebunden werden können.

Einheiten haben oft die grobe Struktur: Zweiteilung in Spezifikation (auch Schnittstelle genannt) und Implementierungsteil (Rumpf). Der Rumpf besteht meist aus einer Deklarationen gefolgt von Anweisungen. Hierbei ist festzulegen, was deklariert werden muss und was nicht und was wie gekapselt (= nach außen hin verborgen) wird.

7. Programme:

Programme sind in der Regel Folgen von Einheiten, wobei ein Startpunkt anzugeben ist.

Für die konkrete Ausführung müssen noch viele zusätzliche Angaben gemacht werden, z.B.:

Einbindung in die Datenverwaltung des jeweiligen Systems: Angabe der Ein- und Ausgabebereiche, der Filesysteme, Anschluss an Netze, Nutzung von Geräten ...

Hinweise an den Compiler, Nutzung von Gegebenheiten (pragmas). Angabe von Übersetzungseinheiten (separate).

Einbinden fremder Programmstücke und vordefinierte Klassen mittels **with** und **use**.

Und dann gibt es noch viele weitere "Features" .....

Anregung an Sie:

Definieren Sie sich Ihre eigene Programmiersprache für einen bestimmten Bereich, z.B.

- Sprache zum Rechnen mit (beliebigen) Mengen,
- Sprache für die Geometrie,
- Sprache für den Bau eines Eigenheims,
- Sprache zur Beschreibung von Schaltwerken,
- Sprache für das Kochen,
- Sprache für das Schachspiel,
- Sprache für Arbeitsprozesse

usw. Machen Sie sich klar, wie einfach dies *im Prinzip* ist, wie komplex die konkrete Ausgestaltung wird und welche Schwierigkeiten beim Zusammenwirken der verschiedenen Sprachelemente und bei der Implementierung in einer Programmiersprache entstehen.

Überlegen Sie auch, wie Ihre Sprache umgestaltet werden muss, um beispielsweise folgende Punkte behandeln zu können:

Gute Erlernbarkeit. Nähe zu Anwendungen.
Gute Bedienoberflächen. Visualisierbarkeit.
Erstellen einer guten Dokumentation.
Zuverlässigkeit und Robustheit der Programme.
Wartung, Pflege, Austauschbarkeit.
Wiederverwendbarkeit von Programmstücken.
Zusammenwirken mit anderen Programmen über Netze.
Verteilung eines Programms auf viele verschiedene Rechner.
Beachtung von Standards und Normen.
Und was die Kunden sonst noch alles gerne hätten ....