Programmierung mit abstrakten Datentypen unter Verwendung von VDM

Jan Peter Mohr
Wirtschaftsinformatik

10. September 1996

Inhaltsverzeichnis

1 Hinweise für die Leserschaft **1**

1.1 Wer soll dieses Handbuch lesen ? 1

1.2 Aufbau dieses Handbuches 1

1.3 VDM - eine kurze Historie 2

2 Abstrakte Datentypen **5**

2.1 Abstraktion beim Programmieren 5

2.1.1 Anlaß der Abstraktion 5

2.1.2 Modularisierung 5

2.1.3 Datenabstraktion 6

2.1.4 „Geheimnisprinzip“ 6

2.2 Was sind abstrakte Datentypen ? 7

2.2.1 Definition 7

2.2.2 Vor- und Nachteile von ADT 8

2.2.3 Formale Methoden 9

2.3 Der modellorientierte Ansatz 9

2.3.1 Repräsentationale Abstraktion 9

2.3.2 Operationale Abstraktion 10

2.3.3 Konkretisierung 11

3 Vienna Development Method **13**

3.1 Struktur einer Modellspezifikation 13

3.1.1 Terminologie 13

3.1.2 Die VDM-Spezifikation 15

3.2 Das VDM–System unter *UNIX* 25

3.2.1 Die Basistypen in der VDM-Bibliothek 26

3.2.2 Die Domain Specification Language 27

3.2.3 Der VDM-Compiler . . . 28
3.2.4 Informationen mit *vdminfo* . . . 28
3.3 Programmentwicklung mit VDM . . . 29
3.3.1 Design-Phase – Detailentwurf . . . 29
Übersetzung der Domain-Gleichungen . . . 31
3.3.2 Die Codier-Phase . . . 32
3.3.3 Übersetzung . . . 33
VDM–Compiler . . . 33
C–Compiler . . . 35
3.3.4 Projektverwaltung mit *make* . . . 36
3.4 Vor- und Nachteile von VDM . . . 39
4 Domain Specification Language (DSL) 41
4.1 Die Gesamtstruktur . . . 41
4.2 Die Domain-Gleichungen . . . 42
4.3 Die Implementationsbeschreibungen . . . 44
5 Die VDM–Basistypen 47
5.1 Speicherverwaltung . . . 48
5.2 Generelle Operationen . . . 51
5.3 Elementare Datentypen . . . 52
5.3.1 Die vordefinierten Zahlentypen . . . 52
5.3.2 Der logische Datentyp *Bool* . . . 53
5.3.3 Der vordefinierte Datentyp *Str* . . . 54
5.3.4 Selbstdefinierte elementare Datentypen . . . 55
5.4 Sets . . . 57
5.4.1 Was ist ein Set ? . . . 57
5.4.2 Set-Implementation . . . 59
5.4.3 Set-Operationen . . . 59
5.4.4 Beispiel: „Coarsest Partitioning“ . . . 64
5.5 Tuples . . . 68

5.5.1 Was ist ein Tuple ? 68
5.5.2 Tuple-Implementationen 70
5.5.3 Tuple-Operationen 70
5.5.4 Beispiel: „Pascal-Dreieck“ 82
5.6 Maps 84
5.6.1 Was ist eine Map ? 84
5.6.2 Map-Implementationen 87
5.6.3 Map-Operationen 88
5.6.4 Beispiel: Stücklistenverwaltung 98
5.6.5 Beispiel: Heap-Speicherverwaltung 102
5.7 Optionals 103
5.7.1 Was sind Optionals ? 103
5.7.2 Optional-Implementationen 105
5.7.3 Optional-Operationen 105
5.8 Trees 107
5.8.1 Was ist ein Tree ? 107
5.8.2 Tree-Implementationen 110
5.8.3 Tree-Operationen 110
5.8.4 Beispiel: Der Binärbaum 112
5.9 Unions 116
5.9.1 Was ist eine Union ? 116
5.9.2 Union-Implementationen 118
5.9.3 Union-Operationen 119

6 Rekursive Typdefinition 123
6.1 Verwaltung von variablen Datenmengen 123
6.1.1 Sequentielle Organisation 123
6.1.2 Rekursionskonzepte 124
6.2 Implementation rekursiver Strukturen 127
6.3 Beispiel: Syntaxdefinition von Programmiersprachen 128

7 **Ein größeres Beispiel** 131

7.1 Quantitative Stückliste — das Datenmodell 131

7.2 Die semantischen Funktionen 132

7.2.1 Laden und Speichern 132

7.2.2 Eingabe einer Teil-Beschreibung 135

7.2.3 Struktur-Stückliste 139

7.3 Das Steuerprogramm 140

8 **Online-Hilfe mit *vdminfo*** 143

8.1 Die Struktur der VDM-Bibliothek 143

8.2 Der Aufruf von *vdminfo* 144

ANHANG 147

A DSL-Syntax 147

B Die Operationen der ADT 149

C Die Implementationen 171

D Der VDM–Compiler 183

Glossar 187

Literaturhinweise 193

Abbildungsverzeichnis 194

Index 195

1. Hinweise für die Leserschaft

1.1 Wer soll dieses Handbuch lesen ?

Da VDM die Programmentwicklung insbesondere in den Phasen Detailentwurf, Codierung, Test und Wartung unterstützt, ist das Erlernen des Umgangs mit dem System eigentlich für alle interessant, die programmieren, also Software-Entwicklerinnen oder Informatik-Studentinnen[1]. Ein VDM-System ist an der FH Wedel installiert, das die Erstellung von Programmen in der Sprache *C* unterstützt; es bildet die Basis dieses Handbuches. Da es keine Tools zur Verarbeitung der formalen Sprache von VDM (META-IV) bereitstellt, liegt der Schwerpunkt dieses Handbuchs auf dem Einsatz der abstrakten Datentypen von VDM bei der Erstellung von Applikationen. Aus der der Struktur dieses VDM-Systems leiten sich einige Vorkenntnisse ab, die Sie als Leserin dieses Handbuches haben sollten:

Vorkenntnisse

- Sie müssen die Programmiersprache *C* beherrschen, da VDM Module bereitstellt, die für die Verwendung in *C*-Programmen konzipiert sind.
- Sie sollten das Konzept der Modularisierung von Programmen erfaßt haben.
- Sie sollten mit der grundsätzlichen Funktionsweise von Modul-Bibliotheken vertraut sein und ihren Vorteile kennen.
- Die Arbeit unter dem Betriebssystem *UNIX* sollte Ihnen geläufig sein. Das betrifft Login, Bedienung der Shell, Editierung von Programmtexten, Aufruf des Compilers und die Verwaltung größerer Programmprojekte mit MAKE.
- Kenntnisse über den Compilerbau erleichtern das Verständnis der internen System-Abläufe und werden auch für einige Beispiele vorausgesetzt; sie sind aber nicht *unbedingt* erforderlich.

1.2 Aufbau dieses Handbuches

Das Handbuch soll die Einarbeitung in die Programmierung mit VDM ermöglichen, aber auch der erfahrenen Anwenderin noch nutzbringend sein. Dazu ist zunächst ein umfangreicher, ausführlicher **Einarbeitungsteil** vor-

Einarbeitung

[1]Ohne die männlichen Leser diskriminieren zu wollen, wird hier aus Vereinfachungsgründen nur die *weibliche* Form verwendet.

handen, den Sie zum Kennenlernen von VDM möglichst von vorn nach hinten durchlesen sollten. Im einzelnen werden folgende Aspekte behandelt (in den entsprechenden Kapiteln):

1. Hinweise für die Leserschaft und kurze Historie von VDM
2. Das hinter VDM stehende Konzept der *abstrakten Datentypen*
3. Die Programmentwicklung mit VDM
4. Die Spezifikation von abstrakten Datentypen mit der *Domain Specification Language*, abgekürzt DSL
5. Die VDM-Basistypen und ihre Anwendung in *C*
6. Rekursive Typdefinition
7. Ein größeres Anwendungsbeispiel
8. Die Anwendung des VDM-Hilfesystems *vdminfo*

Referenz

Der Anhang stellt eine **Referenz** dar, also eine kurze, vollständige Zusammenfassung bestimmter Themen von VDM: Eine Definition der *DSL-Syntax*, eine Gesamtübersicht über die abstrakten Datentypen und ihre Operationen und eine Beschreibung der verfügbaren *Implementationen*. Ein Verzeichnis der *Abbildungen* folgt. Ein *Glossar* erklärt an zentraler Stelle die wichtigen Begriffe, für die eiligen Leserinnen ist ein umfassender *Index* vorhanden.

1.3 VDM - eine kurze Historie

Das Thema dieses Handbuches ist die **Vienna Development Method**, abgekürzt VDM. Sie entstand im Vienna Laboratory von IBM im Rahmen einer Forschungsgruppe, die sich mit der formalen Spezifikation von Programmiersprachen beschäftigte, deren Zweck die Unterstützung der Entwicklung eines Compilers ist mit der daraus erwachsenden Möglichkeit, die Korrektheit des Compilers zu beweisen.

META-IV

1974 gelang es mit Hilfe einer funktionalen Sprache namens META-IV, die von IBM entworfene Sprache PL/1 formal zu definieren. Die formale Methode der Spezifikation mit META-IV wurde VDM getauft.

Bjørner und Jones

Die im Zusammenhang mit META-IV bzw. VDM am haüfigsten genannten Forscher sind der Däne Dines Bjørner und der Brite Cliff B. Jones. Sie verließen bald darauf das Vienna Laboratory, um an Hochschulen zu arbeiten, sind aber weiterhin an der Entwicklung von VDM beteiligt.

Anfang der achtziger Jahre erregte VDM, obwohl es noch keine genormte Spezifikation gab, das Interesse der Industrie. Einige Firmen adaptierten VDM nach ihren Bedürfnissen und entwickelten Programmsysteme damit. Die positive Reaktion bewog Bjørner und Jones, eine Anwender-Vereinigung ins Leben zu rufen; 1987 entstand **VDM–Europe**, die allen Anwendern von VDM offensteht und einerseits ein Register aller VDM-bezogenen Aktivitäten in Europa verwaltet, andererseits Symposien über das Umfeld von VDM abhält. Die Themengebiete der Symposien sind:

VDM–Europe

1. Ausbildung, Training, Technologie-Transfer
2. Austausch von Erfahrungen in der Anwendung von VDM
3. Tools zur Unterstützung der Entwicklung mit VDM
4. Weiterentwicklung der Methode, Grundlagenforschung
5. Standardisierung

Mittlerweile ist VDM eine Methode mit sehr breitem Anwendungsspektrum geworden; ein Normungsprojekt ist gestartet worden und es existieren einige Implementationen, die die zunächst rein formalen Spezifikationen in eine ablauffähige Form bringen. VDM unterstützt besonders gut Detailentwurf, Codierung, Test und Wartung eines Systems — die benötigten Funktionen werden auf sehr abstraktem Niveau entworfen, die entstehenden Module sind gut wiederverwendbar und durch die Möglichkeit, die Korrektheit der Implementation zu beweisen (oder den Beweis zumindest zu unterstützen), sinkt der Aufwand für Test und Wartung des Systems ganz erheblich.

Normung

2. Abstrakte Datentypen

2.1 Abstraktion beim Programmieren

2.1.1 Anlaß der Abstraktion

Ab einer gewissen Größe ist die Komplexität eines Software-Systems von der Entwicklerin nicht mehr vollständig zu erfassen, wenn sie alle Details des Systems berücksichtigen muß. Indem sie sich von Details des Systems löst, also eine grundsätzlichere Sichtweise des Systems und der zugrundeliegenden Struktur anstrebt, kann sie die Komplexität verringern und das System besser verstehen; die Kommunikation über das System ist einfacher. Sie benutzt dazu die Technik der **Abstraktion**. Abstraktion, so sagt der Duden, ist das Absondern des Allgemeinen vom Einzelnen.

Beispiel: Bei der Entwicklung eines Compilers wird zunächst die grundlegende Struktur definiert, die einzelnen Elemente — lexikalische Analyse, Syntaxanalyse, Semantikanalyse, Zwischencode- und Code-Erzeugung, eventuelle Code-Optimierungen und die dazu nötige Symboltabellenverwaltung und Fehlerbehandlung. Wie diese Elemente im Detail realisiert sind, wird zunächst nicht berücksichtigt; schließlich sind ja zum Teil mehrere Realisierungsmöglichkeiten vorhanden, die aber an der grundsätzlichen Struktur eines Compilers nichts ändern.

2.1.2 Modularisierung

Ein Ansatz zur Abstraktion ist die **Modularisierung**. Sie dient zur Aufteilung eines komplexen Programms in eine Reihe von einfacheren Unterprogrammen, deren Zusammenwirken die gleiche Funktion erfüllt. Zur Beschreibung der Funktionsweise des Programms können dann die einzelnen Funktionen der Unterprogramme dienen, wobei die Art und Weise, *wie* das Unterprogramm seine Funktion realisiert (welcher Algorithmus also verwendet wird), zunächst nicht relevant ist.

Die Modularisierung kann, je nach Komplexität der entstehenden Teilfunktionen, fortgesetzt werden, bis die Realisierung trivial ist. Das Gesamtsystem wird also schrittweise verfeinert, eine in der Praxis sehr verbreitete Methode.

Beispiel: Während der lexikalischen Analyse eines Compilerlaufes wird der Eingabetext in einen Strom von Symbolen umgewandelt, die innerhalb der Programmiersprache eine bestimmte Bedeutung haben (Schlüsselworte, Bezeichner, mathematische oder logische Operatoren usw.). Der Text muß gelesen, die

Symbole erkannt und in irgendeiner Weise gespeichert werden. Wie diese drei Teilfunktionen Einlesen, Erkennen, Symbol abspeichern realisiert werden, ist zur Beschreibung der Gesamtfunktion „lexikalische Analyse“ unwichtig.

2.1.3 Datenabstraktion

Es gibt allerdings noch eine andere Möglichkeit der Abstraktion. Ein Programm besteht — vereinfacht — aus Funktionen und Daten. Die Eingabedaten des Programms werden mit Hilfe der Funktionen verarbeitet, dabei werden Ausgabedaten erzeugt. Die Modularisierung betrifft die Funktionen des Programms, die **Datenabstraktion** die zu manipulierenden Daten. Für die Realisierung des Programms sind zunächst nur die Operationen, mit denen die Daten manipuliert werden, notwendig; diese Operationen sind also spezielle Hilfsfunktionen, die von allen Funktionen, die auf die Daten in irgendeiner Form zugreifen, benutzt werden können und sollen. Die internen Strukturen der Datenobjekte müssen daher nicht offengelegt werden! Auch die Beschreibung von Daten ist auf diese Weise vollständig durch ihr „Verhalten“ möglich, d.h. durch Informationen, auf welche Weise sie manipuliert werden können.

Operationen

Beispiel: Die Symboltabelle eines Compilers sammelt Informationen über sämtliche Bezeichner des Programms (Variablen, Konstanten, Funktionsnamen). Die Beschreibung der Symboltabelle ist vollständig mit den notwendigen Operationen möglich (z.B. Erzeugung einer leeren Symboltabelle, Eintreten in einen „Block“ der Programmiersprache, Verlassen eines Blocks, Eintragen eines Bezeichners, Test auf Existenz eines Bezeichners, Test gewisser Attribute eines Bezeichners). Welche Informationen gesammelt werden müssen, kann aus den erforderlichen Operationen abgeleitet werden.

Wie erläutert, ist die Modularisierung durch schrittweise Verfeinerung eine verbreitete Methode zur Abstraktion. Die Datenabstraktion ist weniger bekannt, nicht zuletzt deshalb, weil viele Programmiersprachen keine Methoden zur Datenabstraktion anbieten.

2.1.4 „Geheimnisprinzip“

Guter Programmierstil ist die Beachtung des „Geheimnisprinzips“: Auf einer bestimmten Ebene der Entwicklung ist nur das *Verhalten* der untergeordneten Funktionen und Daten bekannt, nicht aber die Form der Realisierung (ihre Implementation); zudem ist auch die Verwendung der auf dieser Ebene entwickelten Funktionen in höheren Ebenen nicht bekannt. Beispiel auf der obersten Betrachtungsebene: ein Compiler verarbeitet einen Text in einer Quellsprache und erzeugt einen Text in einer Zielsprache.

2.2 Was sind abstrakte Datentypen ?

2.2.1 Definition

Wie „funktioniert“ Datenabstraktion überhaupt? Der zentrale Begriff für die Datenabstraktion ist der **abstrakte Datentyp**, abgekürzt ADT. Ein ADT ist eine Klasse von Objekten, die vollständig und ausschließlich durch die Operationen, die mit den Objekten vorgenommen werden können, definiert ist. Die Spezifikation eines ADT besteht folglich aus einer Spezifikation der Operationen. Die Daten sind dadurch untrennbar mit den Operationen, die sie manipulieren, verbunden. Für die Verwendung eines ADT sind Informationen über seine Implementation nicht nötig. Alle Manipulationen einzelner Objekte sind nur mit Hilfe von Operationen möglich, damit wird der Mißbrauch von evtl. vorhandenem Wissen über die Implementation unmöglich gemacht.

ADT

Beispiel: Ein in allen Programmiersprachen existierender ADT ist die Ganzzahl (Integer): die Repräsentation einer Ganzzahl ist unbekannt (oder sollte zumindest unbekannt sein), die nötigen Operatoren für Ganzzahlarithmetik werden bereitgestellt ($+, -, \times, \div, MOD$), außerdem eine Sammlung von mathematischen Funktionen wie *ln, sin, cos, tan*, $\sqrt{x}$, $|x|$ usw. Benötigt man allerdings zusätzliche Funktionen, müssen sie aus den gegebenen Operationen konstruiert werden[1].

Die meisten benötigten ADT sind allerdings nicht so „einfach“, sondern stellen eine strukturierte Sammlung von Datenelementen dar. **Datenelemente** sind also die Untereinheiten, die einen strukturierten Datentyp bilden. Ein Datenelement *kann* ebenfalls strukturiert sein; auf der untersten Ebene sind nur die **elementaren Datentypen** erlaubt, die von der Sprache bereitgestellt werden. In *C* sind dies ganze Zahlen mit unterschiedlichen Wertebereichen, Zeichen (als Sonderfall von ganzen Zahlen) und Fließkommazahlen.

Datenelemente

Die Operationen haben die signifikanten Merkmale *Name*, *Parameter* und *Rückgabewert*. Die von der Definition her naheliegende Schlußfolgerung, daß zur Spezifikation eines ADT keine Angabe über die Datenelemente nötig ist, ist daher falsch: zumindest einige Operationen beeinflussen Datenelemente, in ihren Parameterlisten müssen diese Datenelemente also auch spezifiziert sein. Allerdings werden in der Regel für die interne Verwaltung des ADT zusätzliche Datenelemente notwendig; um diese braucht sich die Programmiererin nicht zu kümmern.

Ein ADT definiert häufig zunächst nicht nur eine bestimmtes Objekt, sondern eine Klasse von Objekten; dann wird er auch **Typgenerator** genannt. Der Datentyp „Liste“ ist ein solcher Typgenerator, da die Elemente der Liste noch nicht festgelegt sind. Erst nach dieser Festlegung (z.B. „Liste von reellen Zahlen“) ist aus dem Typgenerator ein Datentyp geworden.

Typgenerator

[1]Aufgrund ihrer Einfachheit wird die Ganzzahl allerdings üblicherweise *nicht* als abstrakter Datentyp bezeichnet.

2.2.2 Vor- und Nachteile von ADT

Folgende Vorteile ergeben sich aus der Verwendung von ADT:

- Da nur die Operationen Informationen über die Implementation benötigen, ist es leicht möglich, eine Implementation zu ersetzen, wenn eine bessere Repräsentation des ADT gefunden wird. Nur die Operationen müssen neu erstellt werden, alle Programme, die diesen ADT verwenden, bleiben unverändert.
- Die Programmiererin braucht während der Programmentwicklung auch keinerlei Rücksicht auf die Implementation des ADT zu nehmen, sie kann den bestmöglichen Algorithmus zur Lösung des Problems verwenden.
- Die tatsächliche Implementation des ADT kann zeitlich nach hinten verlagert werden, so daß bei der Auswahl mehr Informationen über die Programmanforderungen an den ADT vorliegen; auch die Operationen können also mit „maßgerechten“ Algorithmen realisiert werden.
- Universelle ADT können leicht in anderen Programmen wiederverwendet werden. Dies ist umso häufiger möglich, je „genereller“ der ADT ist — wie z.B. Listen, Kellerspeicher, Bäume.
- Die Realisierung des ADT kann von anderen Programmiererinnen durchgeführt werden, die Arbeitsteilung wird also unterstützt.
- Viele Methoden zur Datenabstraktion stellen Mittel zum Beweis der Korrektheit von Programmen zur Verfügung, indem die abstrakte Spezifikation in mehreren Schritten konkretisiert wird, die einzeln bewiesen werden können. Selbst wenn der Beweis zu aufwendig ist, leisten sie in der Regel Hilfestellung bei der Auswahl von Testfällen.

Unterstützung durch Sprachen

Leider unterstützen die am meisten verwendeten Programmiersprachen (*C, Pascal, COBOL, FORTRAN* usw.) die Programmierung mit abstrakten Datentypen nicht oder nur eingeschränkt. In *C* gibt es zwar die Typgeneratoren „Tabelle“, „Struktur“ und „Variante“, die jeweils durch einen Satz Operationen unterstützt werden und die auch für viele Zwecke ausreichen; die Datenstrukturen können also als ADT bezeichnet werden. Meist bestehen jedoch implementationsabhängige Randbedingungen (z.B. maximale Größe, Verbot bestimmter Datentypen als Datenelemente), die die Verwendung für einige Problemstellungen unmöglich machen. Der Ausweg, den diese Programmiersprachen zum Teil anbieten, ist die Benutzung von dynamischen Datenstrukturen unter Verwendung von Pointern, die bei zunehmender Komplexität aber schwierig zu implementieren sind; ihr Test muß sehr sorgfältig erfolgen, denn die Speicherverwaltung dynamischer Datenstrukturen ist extrem fehlerträchtig.

2.2.3 Formale Methoden

Es gibt eine Reihe formaler Methoden zum Umgang mit ADT. Einige basieren auf theoretischer Mathematik, andere sind verwandt mit gängigen Programmiersprachen. Meist gibt es allerdings keine Implementation der Methode, so daß der Einsatz sich auf die formale Spezifikation des ADT beschränkt. Der durch VDM repräsentierte **modellorientierte Ansatz** hingegen liegt in einer Implementation unter UNIX vor, die die Sprache *C* um abstrakte Datentypen erweitert. Die theoretischen Grundlagen sind im Abschnitt 2.3 kurz beschrieben.

→ 2.3

Korrektheit von Programmen

Formale Methoden der Abstraktion sind hervorragend dazu geeignet, die Korrektheit von Programmen zu beweisen. Ist die abstrakte Spezifikation beweisbar die korrekte Lösung des Problems und ist jede Konkretisierung beweisbar äquivalent zur vorhergehenden Abstraktion, folgt daraus, daß auch die Implementation korrekt ist; ein Programmtest ist dann unnötig. In der Praxis ist der vollständige Beweis noch nicht möglich, aber eine formale Methode kann den Korrektheitsbeweis unterstützen und zumindest teilautomatisieren.

2.3 Der modellorientierte Ansatz

Formale Sprache

Der modellorientierte Ansatz stellt eine formale Sprache bereit, in der diese Spezifikation in Form einer repräsentationalen und einer operationalen Abstraktion geschrieben wird, entsprechend den beiden Bestandteilen eines ADT (Daten und Operationen).

2.3.1 Repräsentationale Abstraktion

Die **repräsentationale Abstraktion** ist die Spezifikation von Klassen und Objekten abstrakter Datentypen im Hinblick auf ihre Eigenschaften, aber unabhängig von ihrer Repräsentation. Sie besteht aus dem Namen des Datentyps und der Spezifikation der notwendigen Datenelemente. Die Beschreibung des inneren Aufbaus soll so abstrakt wie möglich sein; allerdings orientiert sich die Entwicklerin sehr frühzeitig an den von der formalen Sprache bereitgestellten, sehr mächtigen *Modellen* von Datenstrukturen (daher der Name des Ansatzes), das sind universelle Typgeneratoren, die keinerlei Restriktionen unterworfen sind. Dies sind z.B. die Datentypen Menge, Liste, Abbildung, Verbund[2] und Variante (die auch VDM bereitstellt).

Modelle

Man könnte nun argumentieren, das damit das ADT-Konzept torpediert wird; man kann den ADT *nicht* völlig selbständig und losgelöst definieren, sondern wird gezwungen, sich an zwar sehr mächtigen, aber letztlich doch beschränkten Typgeneratoren zu orientieren. Dem ist allerdings folgendes zu entgegnen:

[2] In *C* heißt dieser Typgenerator „Struktur“

eine Entwicklerin, die eine „reine“, der obigen Definition entsprechende Spezifikation erhält und sie implementieren soll, orientiert sich ebenfalls an bestimmten, ihr geläufigen Datenstrukturen. Einige eignen sich besonders gut zur Repräsentation des geforderten ADT, andere überhaupt nicht. Schließlich wird die Entwicklerin die Datenstruktur wählen, deren Eigenschaften den geforderten am nächsten kommen; einige der Anforderungen lassen sich vielleicht nur umständlich und ineffizient erfüllen, da der gewünschte ADT sicherlich zumindest in Details von dem gewählten „Standard-Typ“ abweicht. Demzufolge ähnelt die grundsätzliche Vorgehensweise in der Praxis auch bei einem völlig selbständig definierten ADT dem Vorgehen des modellorientierten Ansatzes. Der einzige Unterschied könnte sein, daß die Entwicklerin mehr Datenstrukturen kennt, als die formale Sprache anbietet. Daher sollte die formale Sprache um neue Typgeneratoren erweitert werden können.

2.3.2 Operationale Abstraktion

Die **operationale Abstraktion** ist die Spezifikation der Operationen und Funktionen, die auf Objekte angewendet werden, ebenfalls unabhängig von ihrer Realisierung. Diese *Funktionen* stellen komplexere Manipulationen an den Objekten dar, die allerdings aus elementaren Operationen zusammengesetzt werden können; sie zählen also nicht zu den ADT-Operationen, die den ADT „charakterisieren“, und brauchen demnach auch nicht spezifiziert zu werden. Es schadet allerdings nichts, wenn die Programmiererin, die die Implementation vornehmen soll, schon über komplexere Anwendungen des Typs informiert ist — die Auswahl der Repräsentation kann dadurch entscheidend beeinflußt werden.

Zur Spezifikation müssen folgende Elemente vorhanden sein:

- Name der Operation
- Argumentmenge (in Form einer Liste von Parametern, bestehend aus Name und Typ, wie von den Funktionsdefinitionen in *C* bekannt)
- Zielmenge in Form eines Rückgabewertes
- konstruktive oder implizite Spezifikation der Semantik

Semantik und Syntax

Die **Semantik** ist die „Bedeutung“ einer Operation, beschreibt also ihre Auswirkung, während die ersten drei Punkte die **Syntax** der Operation definieren, d.h. Informationen, wie man die Operation korrekt benutzt.

Die **konstruktive Spezifikation** ist eine Beschreibung der Manipulation, die von der Operation durchgeführt wird, in der formalen Sprache. Die **implizite Spezifikation** hingegen beschreibt die Wirkung in Form zweier Bedingungen: die Prä-Kondition (*precondition*) muß erfüllt sein, damit die Funktion aufgerufen werden darf — sonst liefert sie einen Fehlercode. Die Post-

Kondition (*postcondition*[3]) ist nach dem ordnungsgemäßen Ende der Operation erfüllt.

2.3.3 Konkretisierung

→ S. 9

Durch eine schrittweise Verfeinerung wird nun die Ausgangsspezifikation in eine lauffähige Implementation umgewandelt. Schritt für Schritt nimmt die Abstraktion ab, die Repräsentation der Datenobjekte und die Algorithmen für die Operationen werden konkreter. Dabei kann jeder Schritt durch einen formalen Korrektheitsbeweis (siehe S. 9) abgesichert werden; die endgültige Implementation ist dann eine korrekte Repräsentation der ursprünglichen Spezifikation. Aufgrund der Orientierung an Typgeneratoren (Modellen) ist die Konkretisierung relativ schnell möglich. Bei entsprechender Auslegung der formalen Sprache ist auch der Übergang von der konstruktiven Spezifikation der Operationen zu einer Implementation in einer beliebigen Programmiersprache einfach möglich.

[3] Im folgenden werden die englischen Begriffe benutzt.

3. Vienna Development Method

Die Vienna Development Method (VDM) ist ein Vertreter des modellorientierten Ansatzes. In diesem Kapitel wird beschrieben, wie VDM den modellorientierten Ansatz umsetzt und wie man praktisch mit VDM arbeitet. Zunächst wird die generelle Struktur einer Modellspezifikation in VDM erläutert. Die Installation unter *UNIX* an der FH Wedel und ihre Besonderheiten sind Thema des Abschnitts 3.2 — der Prozeß der Programmentwicklung mit Hilfe dieser Installation von VDM folgt im Abschnitt 3.3. Zum Schluß werden die Vorteile der Arbeit mit VDM zusammengefaßt.

3.1 Struktur einer Modellspezifikation

3.1.1 Terminologie

Eine Modellspezifikation besteht in VDM aus einer Klassen-Definition (der sogenannten Domain-Gleichung), der Bedingung für die Wohlgeformtheit eines Objekts dieser Klasse und der Definition aller Funktionen, die mit diesen Objekten umgehen. Der Begriff **Klasse** bzw. **Domain** wird meist synonym mit dem Begriff „abstrakter Datentyp“ verwendet, ist aber allgemeiner und umfaßt *alle* möglichen Datentypen. In diesem Abschnitt werden diese Begriffe zunächst definiert; die konkrete Darstellung in der VDM-Notation folgt im nächsten Abschnitt.

Definition: Eine **Domain-Gleichung** gibt den Namen und die syntaktische Struktur einer Klasse von Objekten an, d.h. aus welchen Bestandteilen ein Objekt besteht und wie diese Bestandteile strukturiert sind. Sie dient also als „Bauplan“ für die Objekte.

Beispiel: Die Definition neuer Datentypen in *C* mit dem typedef-Befehl ist in diesem Sinne auch eine Domain-Gleichung: Es wird ein Name definiert, der als Strukturbeschreibung einer Klasse von Objekten dient. Um eine Klasse *IMAG* von imaginären Zahlen zu definieren, dient folgender Befehl:

```
typedef struct {
        double  re, im;
} IMAG;
```

In VDM ist der Begriff „Domain-Gleichung“ etwas enger gefaßt, er bezieht sich nur auf die mit Hilfe von VDM definierten abstrakten Datentypen — wie eine „echte“ Domain-Gleichung aussieht, erfahren Sie im Abschnitt 3.1.2.

Abstrakte Syntax

Die Sammlung aller Domain-Gleichungen, die für eine Applikation benötigt werden, heißt **abstrakte Syntax**.

Mit Daten repräsentiert ein Programm Objekte der Realität. Selbst wenn Sie die Struktur eines Objekts richtig erkannt und formalisiert haben, ergeben sich daraus viele mögliche Zustände, die ein Objekt in der Realität *nie* einnehmen wird. Das Programm sollte mit solchen Zuständen nicht umgehen, da die Repräsentation der Realität in diesem Fall nicht korrekt ist.

Wohlgeformtheit

Die Bedingung für die **Wohlgeformtheit** ist eine Funktion, die die realen Zustände eines Objekts formalisiert und somit erlaubt, die Korrektheit eines Datums im Programm diesbezüglich zu testen. Ist der Zustand erlaubt, ist das Datum wohlgeformt. Die Domain-Gleichung enthält keine Informationen darüber; daher ist es sinnvoll, ein Kriterium dafür zu definieren. Alle Funktionen, die ein neues Objekt der Klasse erzeugen oder ein existierendes manipulieren, müssen diese Bedingung kennen und einhalten. Eine andere Bezeichnung für diese Bedingung ist **Datentyp-Invariante**.

Beispiel: Die oben definierte Klasse von imaginären Zahlen erlaubt auch Zahlen, deren Imaginär-Teil gleich Null ist, also reelle Zahlen; in diesem Fall hätte das Objekt einen Zustand, der sinnlos ist, da es ja nur die imaginären Zahlen repräsentieren soll. Abhilfe schafft die Funktion

```
Bool is_IMAG (IMAG i)
{ return i.im != 0.0 }
```

Semantische Funktionen

Die Funktionen, die mit den Objekten einer Klasse umgehen, diese also manipulieren, werden auch **semantische Funktionen** genannt. Im Gegensatz zum allgemeinen modellorientierten Ansatz können dies auch komplexe Manipulationen sein, die die Applikation benötigt. Wenn Sie allerdings einen ADT definieren (und das ist fast immer der Sinn einer Modellspezifikation), sollten Sie sich trotzdem die Mühe machen, bestimmte Funktionen zu definieren, die allein den ADT „charakterisieren" (also *Operationen* im Sinne der ADT-Definition) — damit erreichen Sie eine gewisse Unabhängigkeit des ADT von dem spezifischen Problem, wodurch die **Wiederverwendbarkeit** des ADT zunimmt. Die komplexen Funktionen, die die Applikation benötigt, geben dazu Hinweise: sie führen meist mehrere, elementare Manipulationen an den Objekten aus. Diese „Hilfsfunktionen" sollten Sie in *allen* komplexen Funktionen identifizieren — sie bilden in der Regel eine Menge von für den ADT „charakteristischen" Funktionen, passen also in unsere Definition für Operationen!

Wiederverwendbarkeit

Hilfsfunktionen

Beispiel: Für unsere imaginären Zahlen benötigen wir natürlich Funktionen, um mit ihnen zu rechnen, also etwa imaginäre Addition, Subtraktion, Multiplikation usw. Hier sei eine Additionsfunktion definiert:

```
IMAG add_IMAG (IMAG i, IMAG j)
{
  i.re  +=  j.re;
  i.im  +=  j.im;
  return i;
}
```

Konstruktor

Eine Funktion müssen Sie in jedem Fall definieren, den **Konstruktor** für ein wohlgeformtes Objekt der Klasse. Natürlich kann es mehrere verschiedene Konstruktoren für eine Klasse geben, die sich durch die Parameter unterscheiden. Beispiel für einen sinnvollen Konstruktor einer imaginären Zahl:

```
IMAG mk_IMAG (double re, double im)
{
  IMAG new;
  new.re  =  re;
  new.im  =  im;
  return new;
}
```

Ein anderer Konstruktor könnte die neue Imaginärzahl mit dem Inhalt einer bereits existierenden initialisieren.

Wohlgeformtheit?

In den beiden letzten Beispielen wurde übrigens — Sie haben es vielleicht bemerkt — die Datentyp-Invariante nicht beachtet ! Zur Lösung dieses Problems wird jeweils eine zweite Funktion benötigt, die die Imaginärzahl als Referenzparameter („Call by Reference“) hat und einen Wahrheitswert zurückliefert. Am Beispiel des Konstruktors:

```
Bool test_mk_IMAG (IMAG * i, double re, double im)
{
  if (im != 0.0) {
      (*i).re = re;
      (*i).im = im;
  }
  return (im != 0.0);
}
```

3.1.2 Die VDM-Spezifikation

Nachdem nun die Begriffe bekannt sind, wollen wir sehen, wie solch eine Spezifikation in VDM aussieht. VDM unterstützt die repräsentationale Abstraktion durch die *Domain-Gleichungen*, die operationale Abstraktion durch die *Funktions-Spezifikation*.

Domain-Gleichung

Mit der Domain-Gleichung wird die innere Struktur modelliert. Dafür stehen eine Reihe von Datenstrukturen zur Verfügung, um eine optimale Anpassung des benötigten ADT an die von der Applikation gestellten Anforderungen zu erreichen: die **VDM-Basistypen**. Sie können diese Datentypen beliebig schachteln, auch rekursive Strukturen sind möglich. Allerdings ist folgendes zu bedenken:

VDM-Basistypen

> Der modellorientierte Ansatz fordert bei der Definition der ADT die möglichst frühzeitige Bezugnahme auf die in der Methode vorhandenen „Modelle“ von Datenstrukturen (wie im Abschnitt 2.3 dargestellt). Gleichwohl *kann* die Entwicklerin zunächst eine losgelöste Spezifikation erstellen, wenn die Methode das sprachlich unterstützt. Die FH-Installation von VDM bietet eine solche Unterstützung leider nicht an — Sie sollten also nur die VDM-Basistypen verwenden, die aber so mächtig sind, daß eine Lösung damit in der Regel leichtfällt.

Notation

Für jeden VDM-Basistyp gibt es eine spezielle Notation. Sie lernen jetzt diese Datentypen kennen, allerdings nur in knapper Form, da Kapitel 5 sich ausführlich mit allen VDM-Basistypen befaßt. Die englischen Namen werden verwendet, weil sie üblich sind — dadurch werden Sie mit der Nomenklatur der FH-Installation vertraut.

Den **Set** (Menge) kennen Sie aus dem Bereich der Mathematik: er enthält eine beliebige Anzahl von Elementen des Elementtyps ohne festgelegte Reihenfolge. Einen Set $\mathcal{M}$ von Elementen des Typs $\mathcal{A}$ definieren Sie mit

Set $\mathcal{M} = \mathcal{A}$ –set

Das **Tuple** (Liste) ist eine Erweiterung der aus C bekannten eindimensionalen Tabelle, also einer Sequenz von gleichartigen Elementen. Allerdings brauchen Sie *keine* Obergrenze festzulegen! Einen Tuple $\mathcal{L}$ von Elementen des Typs $\mathcal{A}$ definieren Sie mit

Tuple $\mathcal{L} = \mathcal{A}^*$

Die **Map** (Abbildung) ist eine Funktion mit endlicher Argumentmenge, also die Menge aller Paare (a, b), für die gilt: jedem a ist höchstens ein b zugeordnet. die a heißen *Argumentwerte*, die b heißen *Map-Werte* (analog zum Begriff „Funktionswert“). In der Terminologie von VDM heißt die Argumentmenge auch *Argument-Domain*, die Zielmenge *Co-Domain*. Eine Map $\mathcal{F}$ von Paaren (a, b) mit $a \in \mathcal{A}$ und $b \in \mathcal{B}$ definieren Sie mit

Map $\mathcal{F} = \mathcal{A} \xrightarrow[m]{} \mathcal{B}$

Der **Tree** (Verbund) ist eine geordnete Sammlung von Elementen unterschiedlichen Typs, die einzeln benannt sind, ähnelt also dem *struct* in *C*. Einen Tree $\mathcal{T}$ der Elementtypen $\mathcal{A}_1 \ldots \mathcal{A}_n$ definieren Sie mit

Tree $\mathcal{T} :: \mathcal{A}_1 \ldots \mathcal{A}_n$

Benötigen Sie den Tree nur als Elementtyp innerhalb eines anderen ADT, brauchen Sie ihm keinen Namen geben — stattdessen wird die Folge der Elementtypen in runde Klammern eingeschlossen. Ein solcher Tree heißt **anonym**.

$(\mathcal{A}_1 \ldots \mathcal{A}_n)$

Die **Union** (Variante) enthält *ein* Datenelement, das aber einer Reihe verschiedener Datentypen angehören kann und den Elementtyp auch während seiner Existenz wechseln kann, entspricht also der gleichnamigen *union* in *C*. Eine Union $\mathcal{U}$ der Elementtypen $\mathcal{A}_1 \ldots \mathcal{A}_n$ definieren Sie mit

Union $\mathcal{U} = \mathcal{A}_1 \mid \ldots \mid \mathcal{A}_n$

Das **Optional** enthält zu einem bestimmten Zeitpunkt entweder ein Objekt des Elementtyps oder den Wert **nil**, der soviel bedeutet wie „gar nichts“. In *C* gibt es z.B. den speziellen Wert *NULL* für Pointer, die gerade auf kein Objekt zeigen. Technisch wird er bei fast allen *C*-Compilern durch den *long*-Wert 0 dargestellt, weil die Speicheradresse 0 für Daten eines Anwenderprogramms nicht zugänglich ist; könnte auch dort ein Datum gespeichert sein, müßte *NULL* auf andere Weise definiert sein, der Zahlentyp *long* wäre ungeeignet. In solchen Fällen ist der Typ Optional sehr nützlich. Ein Optional $\mathcal{O}$ von Elementen des Typs $\mathcal{A}$ definieren Sie mit

Optional $\mathcal{O} = [\ \mathcal{A}\ ]$

Um eigene Datentypen zu modellieren, können Sie diese VDM-Basistypen beliebig verschachteln !

Datentyp-Invariante

Die Bedingung für die Wohlgeformtheit eines Objekts ist ein sogenanntes **Prädikat**: die Beschreibung einer Eigenschaft, die ein Objekt haben kann oder nicht. Sie läßt sich als Funktion auffassen, die einen logischen Wahrheitswert zurückliefert. Zur Beschreibung eignen sich demzufolge die logischen Verknüpfungen, die Vergleichsoperatoren und alle weiteren Operatoren, die **true** und **false** als Ergebnis haben (hier eine Auswahl):

- AND ($\wedge$)
- OR ($\vee$)
- NOT ($\neg$)

- Implikation ($\supset$ oder $\Rightarrow$) — „wenn...dann..."
- Äquivalenz ($\equiv$) — „...genau dann, wenn..."
- Gleichheit / Ungleichheit
- $<$ / $\leq$
- $>$ / $\geq$
- $\in$ / $\notin$
- $\subset$ / $\subseteq$ / $\not\subset$ / $\not\subseteq$
- $\forall$ / $\exists$

Außerdem können Sie die Sprachelemente verwenden, die für die Funktionsspezifikation zur Verfügung stehen, denn die Datentyp-Invariante ist letztlich nur eine spezielle Funktion.

Beispiel: Die allgemeine Darstellung der Datentyp-Invarianten einer imaginären Zahl sieht so aus:

inv-IMAG(re, im) = (im $\neq$ 0.0)

Sie sollten die Datentyp-Invariante so genau wie möglich definieren; durch die Anwendung in Ihrer Applikation werden undefinierte Zustände vermieden, also wird deren Konsistenz und Korrektheit wahrscheinlicher. Haben Sie die Datentyp-Invariante als Funktion implementiert, sollten Sie sie bei jeder Manipulation eines Objekts aufrufen und eine entsprechende Fehlerbehandlung vorsehen (wie im Beispiel mit den imaginären Zahlen in der Funktion test_mk_IMAG), wenn dadurch die Laufzeit nicht unzumutbar ansteigt — im Zweifel ist ein sicheres, langsames Programm besser als ein schnelles, aber fehlerhaftes, gerade wenn es mit sensiblen Daten arbeitet, und Probleme bei der Performance lassen sich mit leistungsfähigerer Hardware meist beseitigen. Es ist in jedem Fall guter Programmierstil, daran zu denken.

Funktionsspezifikation

VDM propagiert die beiden üblichen Möglichkeiten der **konstruktiven** und der **impliziten Spezifikation** von semantischen Funktionen. Bei beiden Arten definieren Sie zunächst die Syntax der Funktion und ihre *precondition*.

Syntaxdefinition: Zunächst muß die **Syntax** der Funktion mit der TYPE-Klausel definiert werden:

TYPE Name (Argumentliste) : Argumenttyp-Liste $\longrightarrow$ Resultattyp

Die Argumentliste enthält die Namen der Argumente, wie sie in der weiteren Spezifikation verwendet werden; werden sie nicht gebraucht, können Sie sie weglassen. Die Argumenttyp-Liste legt die Definitionsmenge der Funktion fest, der Resultattyp (wie zu erwarten war) die Zielmenge. Handelt es sich um einen Konstruktor ohne Argumente oder um eine Prozedur ohne Resultat, können Sie den entsprechenden Teil ebenfalls weglassen.

Beispiele

Die ersten drei Beispiele beziehen sich auf Operationen des VDM-Basistyps *Set*, das letzte definiert die Syntax der imaginären Addition.

TYPE mk_empty_aset (): $\longrightarrow$ Aset
TYPE is_empty_aset(aset): Aset $\longrightarrow$ Bool
TYPE print_aset (aset): Aset $\longrightarrow$
TYPE add_IMAG (i,j): IMAG IMAG $\longrightarrow$ IMAG

precondition

Die **precondition** stellt eine Vorbedingung für den Funktionsaufruf dar — ist sie nicht erfüllt, darf die Funktion nicht aufgerufen werden, da sonst undefinierte Zustände entstehen können. Die *precondition* kann mit den gleichen Mitteln spezifiziert werden wie die Datentyp-Invariante, da beides Prädikate sind. ! $\rightarrow$ Die Datentyp-Invariante ist übrigens eine implizite *precondition*, da die Funktion nur mit sinnvollen Zuständen der Objekte auch sinnvoll umgehen kann. Format der *precondition*:

pre: *Bedingung*

Die Bedingung wird als logischer Ausdruck formuliert, der entweder true oder false ergibt. Ist keine *precondition* nötig, schreiben Sie pre: true oder Sie lassen sie weg.

Beispiel:

Die *precondition* der Addition zweier Imaginärzahlen muß ausdrücken, daß die Summe wieder eine Imaginärzahl sein wird, mit einen Imaginärteil ungleich 0:

pre: i.im + j.im $\neq$ 0

Implizit...

Die **implizite Spezifikation** fügt nur noch eine *postcondition* hinzu, ein weiteres Prädikat, das eine Aussage über den Programmzustand nach der Abarbeitung der Funktion macht, vorzugsweise natürlich über das Resultat der Funktion. Wie die Funktion dieses Resultat erreicht, wird nicht angegeben. Diese Art der Beschreibung nennt man **deskriptiv**[1]. Sie ist in der Regel ziemlich einfach zu erstellen, da man den Sinn der Funktion (ihre Semantik) bei ihrer Spezifikation schon kennt — sonst bräuchte man sie ja gar nicht. Leider ist diese Form der Spezifikation nicht sehr hilfreich, wenn es an die Implementation geht, daher sollte sie in jedem Falle nur als erster Ansatz gesehen werden; im weiteren Verlauf der Entwicklung muß daraus eine konstruktive Spezifikation entstehen. Format der *postcondition*:

[1] SQL und Prolog sind Vertreter deskriptiver Sprachen. Sie beschreiben das gewünschte Ergebnis einer Funktion ohne „Anleitung“, wie man zu diesem Ergebnis kommt; das muß der Übersetzer herausfinden.

post: *Bedingung*

Beispiel: Eine mögliche, nicht sehr geistreiche *postcondition* der Addition zweier Imaginärzahlen ist

post: sum = Complex-Sum(i,j) $\wedge$ is-IMAG(s)

Complex-Sum sei ein informaler Name für die komplexe Addition, die ja für Imaginärzahlen verwendet werden muß, is-IMAG ist ein Prädikat, wie es in einem der ersten Beispiele dieses Kapitels kurz angesprochen wurde.

Konstruktiv... Die **konstruktive Spezifikation** beschreibt die Arbeitsweise der Funktion möglichst exakt; VDM stellt dafür die Sprache META-IV mit einer Kombination aus funktionalen und prozeduralen Sprachelementen bereit[2]. Das Funktionsergebnis wird durch *einen* Ausdruck definiert, der beliebig tief verschachtelt sein kann; innerhalb der Funktion können lokale Variablen erzeugt werden, mit denen in einer tieferen Stufe gearbeitet wird. Eine Sequenz von Aktionen kann nur durch diese hierarchische Schachtelung ausgedrückt werden. Komplexe Unterfunktionen werden ausgelagert und separat spezifiziert.

Anhand von zwei Beispielen werden die Sprachelemente hier eingeführt und in einer Übersicht vervollständigt.

Beispiel: Sie möchten in einem Tuple ein Element suchen und als Ergebnis dessen Position erhalten; ist es mehrfach vorhanden, soll die *erste* Position geliefert werden — eine Anwendung wäre z.B. das Durchsuchen eines Strings (Tuple von Zeichen) nach einem bestimmten Zeichen. Die Operation index löst das Problem:

TYPE index (tup,elem): Domain Element $\rightarrow$ Nat0
index =
  let eset = { i | i $\in$ ind (tup) $\wedge$ tup (i) = elem } **in**
    if eset = { }
      then 0
      else min (eset)

tup ist (wie man in der TYPE-Klausel sieht), das Tuple von Objekten des Typs Element, zu denen elem gehört. Die Funktion wird wie folgt durchlaufen:

1. Zunächst wird das lokale Objekt *eset* durch Auswertung der mathematisch notierten Konstruktionsanweisung für eine Menge ermittelt (mit der **let**-Klausel). Der Ausdruck wird gelesen „die Menge aller i für die gilt: i ist Element der Menge aller Indizes des Tuple (Tuple-Operation ind) und das Element an der Position i (Tuple-Operation Index-Zugriff) ist das Gesuchte“.

[2]Funktionale Sprachen wie LISP basieren auf Funktionsaufrufen und Rekursion, es gibt keine freien Variablen. Prozedurale Sprachen wie C basieren auf Anweisungen, Variablen und Verzweigungen zur Steuerung des Kontrollflusses (*while*, *if-then-else* und auch *goto*).

2. Mit dem für *eset* ermittelten Wert wird der Ausdruck ausgewertet, **in** den er „hineingegeben“ wird —

3. Wenn (**if**) *eset* leer ist, dann (**then**) ist das Resultat des Ausdrucks (und damit das von index) 0, sonst (**else**) ist es der kleinste Wert (Set-Operation min) in *eset*.

Sprachelemente

Nach diesem einführenden Beispiel folgen nun die wichtigsten Konstrukte der Sprache:

Das Konstrukt **let** *Ausdruck* **in** wertet den Ausdruck aus (damit wird häufig ein strukturiertes Objekt in seine Elemente zerlegt oder, im Gegenteil, aus Elementen konstruiert) und übergibt die dabei temporär erzeugten Objekte (im Beispiel *eset*) dem „Unterprogramm“, das auf **in** folgt. In *C* kann man das Konstrukt sehr einfach mit folgenden Makros realisieren:

```
#define LET
#define IN ;
```

Nun kann man jede Zuweisung mit „LET...IN“ klammern und darauf eine Anweisungsfolge mit der Auswertung der zugewiesenen Variablen folgen lassen. Strenggenommen ist das natürlich nur eine Hilfstechnik; *C* ist schließlich keine funktionale Sprache.

Die Auswahl kann durch Angabe einer Bedingung erfolgen, wie die temporären Objekte gewählt werden sollen. Dazu dient die Klausel **let**... **be s.t.**... **in**, das bedeutet „wähle xxx so aus, daß folgendes gilt:“.

Sehr oft werden Elemente aus zusammengesetzten Objekten entnommen. Das sieht im einzelnen so aus:

Sets: **let** $elem \in Set$ **in** — zufällige Auswahl eines Elements

Tuples: **let**
$elem =$ hd Tup **in** — Auswahl des ersten Elements
$subtup =$ tl Tup **in** — Auswahl des Tuples ohne „head“
$elem =$ last Tup **in** — Auswahl des letzten Elements
$elem = Tup(i)$ **in** — Auswahl des i.ten Elements

Maps: **let**
$elem1 \in$ dom Map **in** — zufällige Auswahl eines Argumentwertes
$elem2 \in$ rng Map **in** — zufällige Auswahl eines Map-Wertes
$elem2 = Map(elem1)$ **in** — Auswahl eines bestimmten Map-Wertes

Trees: **let**
$Tree = mk-tree(a_1, \ldots, a_n)$ **in** — Konstruktion eines Tree aus Komponenten
$mk-tree(a_1, \ldots, a_n) = Tree$ **in** — Zerlegung eines Tree in Komponenten

Es gibt die Möglichkeit, globale, statische Variablen zu definieren mit **dcl**; ihnen wird mit dem Zuweisungsoperator := ein Wert zugewiesen, auf ihren Inhalt wird mit **c** (Contents-Operator) zugegriffen. Beispiel:

Erzeugung mit Initialisierung und Datentyp:
dcl var := init-value **type** Typname
Zuweisung:
var := new-value
Zugriff auf den Inhalt:
var := func (**c** var)

if *Ausdruck* **then** *Klausel* **else** *Klausel* entspricht der aus C bekannten **if**–Struktur.

Der **switch**-Struktur entspricht folgendes Konstrukt:

*(***cases** *Ausdruck0:*
Ausdruck1 $\rightarrow$ *Klausel1,*
Ausdruck2 $\rightarrow$ *Klausel2, ...*
Ausdruck-n $\rightarrow$ *Klausel-n,*
T $\rightarrow$ *Default-Klausel)*

Für verschachtelte **if**... **then**... **else if**... gibt es die **case**–ähnliche Struktur

(Ausdruck1 $\rightarrow$ *Klausel1,*
Ausdruck2 $\rightarrow$ *Klausel2, ...*
Ausdruck-n $\rightarrow$ *Klausel-n,*
T $\rightarrow$ *Default-Klausel)*

Der **for**–Struktur in C ähneln die folgenden Konstrukte:

1. ($\forall$ a $\in$ *Set* [(*Ausdruck*)]) (*Klausel*) — für alle Elemente eines Set, für die der (optionale) Ausdruck erfüllt ist (der Set kann z.B. auch die Argumentmenge einer Map sein !)

2. **for** i **in** *Tuple* **do** (*Klausel*) — für alle Elemente einer Liste

Die **while**–Struktur ist ebenfalls vorhanden:

while *Ausdruck* **do** (*Klausel*)

Für Tuples, Sets und Maps stehen Konstruktoren zur Verfügung, die die Elemente durch ein Prädikat beschreiben (im obigen Beispiel wurde ein solcher für *eset* verwendet):

Sets:	{ a | Ausdruck (a) }
Tuples:	< a | Ausdruck (a) >
Maps:	[a $\mapsto$ b | Ausdruck (a,b)]

Die üblichen arithmetischen und logischen Verknüpfungen stehen zur Verfügung: $+$ $-$ $\times$ $\div$ sowie $\vee$ $\wedge$ $\neg$

Test, ob alle Elemente einer Menge eine bestimmte Eigenschaft haben:

$(\forall\ e \in\ A)\ (Predicate\,(e))$

Test, ob mindestens ein Element einer Menge eine bestimmte Eigenschaft hat:

$(\exists\ e \in\ A)\ (Predicate\,(e))$

Undefinierte Zustände (in denen irgendeine Fehlerbehandlung nötig wird), werden einfach mit **undefined** gekennzeichnet.

Beispiel: Das abschließende Beispiel verlangt eine kleine Einleitung:

Alle Operationen müssen in jeder Implementation definiert sein. Wenn eine neue Implementation geschaffen wird, geht also die Anpassung aller Operationen an deren spezifische Eigenschaften los...

Tuples können z.B. als Tabelle oder als verkettete Liste definiert werden. Die Operation tl (Tail), die ein Tuple ohne das erste Element liefert, muß im ersten Fall (Tabelle) eine neue, kleinere Tabelle erzeugen, das Ende der alten Tabelle umkopieren und diese dann löschen; im zweiten Fall (verkettete Liste) wird der Zeiger auf den Kopf des Tuple einfach auf den Nachfolger „verbogen" und das erste Element gelöscht. Die Operation len ermittelt die Länge eines Tuple; bei der Tabelle wird einfach die Gesamtgröße durch die Elementgröße dividiert, die verkettete Liste jedoch muß einmal bis zum Ende durchlaufen werden. Man sieht: die Algorithmen sind für die beiden Implementationen völlig unterschiedlich — beide Operationen müssen neu codiert werden.

Es gibt eine Möglichkeit, unnötige Neu-Codierungen (z.B. von len) zu umgehen: man kann len und auch einige andere Operationen nämlich mit einem *Funktional* so programmieren, daß sie nicht von der Implementation abhängen, sondern nur von anderen Operationen — hat man diese codiert, ist len also ebenfalls vorhanden. Aber was ist ein Funktional ?

Funktional: Ein **Funktional** ist eine Funktion, die als Ergebnis wieder eine Funktion liefert. Um möglichst vielseitig einsetzbar zu sein, hat es als Parameter meist andere Funktionen; durch die Auswahl dieser Funktionen ist das Funktional steuerbar.

Das folgende Funktional für Tuples ist in ähnlicher Form im VDM-System vorhanden. Es hat drei Parameterfunktionen; sind sie determiniert, erhält man eine Funktion, die ein Tuple-Objekt auf ein Resultat abbildet — der Resultattyp ergibt sich allerdings erst durch die Festlegung der drei Parameterfunktionen, so daß beliebige Funktionen realisiert werden können!

TYPE fold (cvfct,nullfct,foldfct) (arg):
((Element → Result) (→ Result) (Result Result → Result))
⟶ (Domain → Result)

```
fold =
  let userfct (tup) =
      if tup =< >
          then nullfct
          else let head = hd (tup), tup2 = tl (tup) in
              foldfct (cvfct (head), userfct (tup2))
  in userfct (arg)
```

Realisierung von len:

```
TYPE len (tup): Domain → Nat0
len = fold (one, zero, plus) (tup)
```

Parameterfunktionen:

```
TYPE one (elem): Element → Result
one = 1

TYPE zero ():          → Nat0
zero = 0

TYPE plus (e1,e2): Nat0   Nat0   →   Nat0
plus = e1 + e2
```

Diese Spezifikation ist etwas schwerer zu entschlüsseln: Das Funktional fold liefert eine Funktion, die ein Objekt des Datentyps „Tuple“ auf ein Objekt des Datentyps „Result“ (der durch einen Implementationsparameter festgelegt wird) abbildet. Das Argument wird zunächst geprüft, ob es leer ist (die spitzen Klammern bezeichnen ein leeres Tuple); wenn ja, ist das Resultat das Ergebnis der konstanten Funktion nullfct. Wenn nicht, wird die Funktion foldfct auf die beiden Argumente vom Typ „Result“ angewendet — das erste ergibt sich aus der Anwendung der Funktion cvfct auf das erste Element des Tuple, das zweite wird durch einen rekursiven Aufruf der mit fold generierten Funktion mit dem Rest des Tuple als Argument erzeugt.

Die drei Parameterfunktionen sind sehr einfach und entsprechen ihrem Namen: zero liefert konstant den Wert 0 zurück, one liefert konstant den Wert 1. plus addiert zwei Zahlen. len ergibt also für das leere Tuple den Wert 0, bei einem Tuple mit einem Element wird plus mit den Argumenten 1 und len (< >) aufgerufen — das Prinzip dürfte erkennbar sein.

Die verschiedenen existierenden Funktionale eignen sich für solche Vorgänge, die mit den Worten „für alle“ zu beschreiben sind, denn diese lassen sich durch rekursive Funktionen sehr gut abbilden. Ebenso können die Quantoren der Mengentheorie ($\forall$ und $\exists$) mit Funktionalen erzeugt werden. Ihr Nachteil ist natürlich die schlechte Performance — eine Funktion wie len sollte nicht unbedingt mit fold realisiert werden, denn eine iterative Lösung ist wesentlich schneller. len diente hier nur zur Verdeutlichung des Konzepts.

Wohlgeformtheit

Zur vollständigen Spezifikation einer Funktion gehört der **Beweis der Er-**

haltung der Datentyp-Invariante — aus wohlgeformten Objekten, die die Funktion als Argumente erhält, müssen wohlgeformte Objekte hervorgehen. Dies ist demzufolge eine implizite *postcondition*. Der Beweis kann formal oder argumentativ durchgeführt werden; meist ist er offensichtlich.

Notationsvorschläge

Wie erstellen Sie nun solch eine Modellspezifikation ?

Domain-Gleichungen

Die **Domain-Gleichungen** sollten Sie in der angegebenen Notation schreiben; daraus entsteht ohne große Änderungen die Spezifikation in der *Domain Specification Language*, die das Thema von Kapitel 4 ist.

Datentyp-Invariante

Die **Datentyp-Invariante** wird von der FH-Installation nicht berücksichtigt. Daher könnten Sie sie gleich als *C*-Funktion schreiben. Die angegebene Notation ist aber meist sehr einfach in eine *C*-Funktion umzusetzen und hat den Vorteil, auch für *C*-Laien, die aber mit mathematischen Notationen vertraut sind, verständlich zu sein.

Funktions-Spezifikation

Sie haben die grundlegenden Elemente der funktionalen Sprache von VDM kennengelernt und auch zwei Beispiele gesehen; im Kapitel 5 werden Sie weitere Spezifikationen begegnen. Ein praktisches Problem sei nicht verschwiegen: die FH-Installation unterstützt diese Notation *nicht*, Sie müssen sie von Hand in *C*-Quelltext umsetzen. Das ist allerdings nicht übermäßig schwierig, und mit den Funktionalen stehen Ihnen Konstruktionsmittel für komplexe Formulierungen zur Verfügung. Die Syntaxdefinition können Sie direkt in *C*-Funktionsköpfe (sogenannte *Prototypen*) umsetzen.

Die Bindung an die formale Sprache ist also nicht sehr stark; Sie könnten auch prozedurale Spezifikationen erstellen. Doch die Sprachmittel von META-IV sind so mächtig, daß Sie damit kürzere Entwurfszeiten erreichen; die Umsetzung in *C* ist eine Sache der Übung und mit zunehmender Erfahrung wird die Entwicklungszeit insgesamt wesentlich kürzer sein.

Wenn Sie Ihr Wissen über META-IV noch vertiefen möchten, beachten Sie die Literaturhinweise.

3.2 Das VDM–System unter *UNIX*

In diesem Abschnitt werden zunächst nur die Elemente und die Struktur des VDM-Systems an der FH Wedel behandelt und die zum Verständnis des Systems nötigen technischen Grundlagen. Wie Sie praktisch damit arbeiten und Programme entwickeln, ist das Thema des Abschnitts 3.3.

3.2.1 Die Basistypen in der VDM-Bibliothek

Die VDM-Basistypen — also die abstrakten Datentypen Set, Tuple, Map, Tree, Union und Optional — wurden mit Hilfe dynamischer Datenstrukturen in *C* programmiert, ebenso die zu diesen ADT vorhandenen Operationen. Dadurch wird die Größe eines Objekts nur durch den verfügbaren Hauptspeicher (in einer Umgebung mit virtueller Speicherverwaltung sogar nur durch den Festplattenspeicher) beschränkt.

Mehrere Implementationen

Der Clou: Zu jedem VDM-Basistyp gibt es *mehrere* Implementationen, die sich im Hinblick auf die benutzte Speicherallokationsstrategie und die zugrundeliegenden Datenstrukturen unterscheiden, woraus sich unterschiedliche Anforderungen an Laufzeit und Speicherplatz ergeben. Das grundsätzliche Verhalten einer Implementation können Sie meist noch durch bestimmte *Parameter* im Detail steuern. Eine Anpassung an die besonderen Anforderungen der Applikation ist damit möglich. Ein Beispiel: wenn Sie bereits wissen, mit wievielen Objekten eines ADT Ihr Programm maximal arbeitet, kann die Verwaltung des ADT daran angepaßt werden.

Der für einen VDM-Basistyp definierte Satz an Operationen ist grundsätzlich in *allen* Implementationen vorhanden, Sie können eine Implementation also jederzeit leicht austauschen, ohne in Ihrem Programm Änderungen vornehmen zu müssen; das ist sehr nützlich, um die beste Kombination aus Performance und geringem Speicherbedarf zu ermitteln. Eine Übersicht über die aktuellen Implementationen finden Sie im Anhang C.

→ Anhang C

Operationen

Die Implementation der Operationen muß berücksichtigen, daß jeder VDM-Basistyp ein Typgenerator ist, also nicht immer die gleichen Elementtypen hat — es muß also eine Möglichkeit vorgesehen sein, die Parametertypen der Operationen erst *bei der Einbindung in die Applikation* festzulegen !

Ein weiteres Problem: Sie dürfen jeden VDM-Basistyp mehrfach für die Konstruktion eigener ADT verwenden, z.B. den Set für je eine Menge Intset von Zahlen und eine Menge Boolset von Wahrheitswerten. Jede Mengen-Operation (z.B. mk_empty, die Erzeugung der leeren Menge) muß dann zweimal vorhanden sein, jeweils mit unterschiedlichen Parametern und Resultaten — mit der Konsequenz, daß der Name mk_empty allein nicht ausreicht, um die richtige Operation auszuwählen.

Die zunächst vielleicht naheliegende Lösung einer bereits compilierten Modulbibliothek scheidet aus diesen Gründen leider aus, da *C* keine Mittel zur Umgehung dieser Probleme bei compilierten Modulen anbietet; stattdessen muß der Quellcode der Funktionen entsprechend geändert werden.

VDM-Bibliothek

Aus diesem Grunde wurde die **VDM-Bibliothek** als Quellcode-Bibliothek konzipiert. In ihr ist der Quellcode aller Operationen, die für den verschiedenen VDM-Basistypen definiert sind, in einer effizienten Struktur abgelegt, die das schnelle Auffinden unterstützt. Sie wählen die benötigten Operationen aus – das VDM-System generiert daraus ein passendes Quellmodul, das Sie zusammen mit der Applikation compilieren.

Die Lösung der beiden Probleme ist jetzt einfach:

1. Die Datentypen der Parameter und des Resultats werden aus der Spezifikation des ADT ermittelt und an der entsprechenden Stelle im Quellcode eingetragen.

2. Für jeden von Ihnen spezifizierten ADT wird auf diese Weise ein *eigener Satz* Operationen generiert, die natürlich auch für jeden ADT einen eigenen Namen bekommen. Die **generischen Namen**[3] der Operationen werden daher erweitert, und zwar um den Namen des ADT — das ist eindeutig, da alle ADT unterschiedliche Namen haben (Ausnahme: ein ADT, der aus Trees oder Unions besteht, braucht für den Zugriff auf die Komponenten mehrere Selektions-Funktionen, deren generischer Name s ist. VDM fordert, daß jede Komponente einen einzigartigen, mit s_ beginnenden Namen haben muß, der dann identisch mit dem erweiterten Namen der Selektions-Funktion ist).

Erweiterung der Operationsnamen

Die Namenserweiterung muß noch etwas genauer beschrieben werden: dem generischen Namen wird zunächst ein *Unterstrich* nachgestellt, dann folgt in der Regel der ADT-Name, wobei der Großbuchstabe am Anfang in eine Kleinbuchstaben konvertiert wird. Beispiele: Für die oben erwähnten Mengen Intset und Boolset heißen die Konstruktoren einer leeren Menge mk_empty_intset und mk_empty_boolset. Da ein Tree mehrere Komponenten enthält und ein Union-Objekt seinen Datentyp ändern kann, werden die komponentenbezogenen Operationen von Tree und Union mit dem jeweiligen Komponentennamen erweitert, eventuell auch zusätzlich mit dem Namen des ADT. Die beiden führenden Zeichen „s_“ eines Komponentennamens werden dabei weggelassen ! Beispiel:

Uni = Intg | Str
⇒ Die Konstruktoren heißen mk_intg_uni und mk_str_uni

Tree :: s_first : Intg s_last : Str
⇒ Die Selektoren heißen s_first und s_last

3.2.2 Die Domain Specification Language

Ihre Modellspezifikation ist fertig, aber nicht „maschinenlesbar“. Sie wissen, welche VDM-Basistypen Sie benötigen, Sie haben Ihre semantischen Funktionen konstruktiv spezifiziert und wissen auch, welche grundlegenden Operationen der VDM-Basistypen Sie benötigen. Wie geht es weiter ?

[3] Der generische Name beschreibt die grundsätzliche Funktion der Operation — z.B. testet is_subset, ob $\mathcal{A} \subset \mathcal{B}$ ist

Sie „übersetzen“ Ihre Spezifikation in eine für das VDM-System verständliche Sprache, die **Domain Specification Language** (abgekürzt DSL). Keine Panik, das ist einfach: der erste Teil einer DSL-Spezifikation enthält die *Domain-Gleichungen*, die Sie bereits erstellt haben und die Sie nur geringfügig zu verändern brauchen. Der zweite Teil enthält für jeden angegebenen ADT die gewünschte *Implementation* (Sie wissen ja, es gibt zu jedem VDM-Basistyp mehrere Möglichkeiten) und die von Ihren semantischen Funktionen benutzten *Operationen* — natürlich nur ihre generischen Namen, die auch **Enrichments** genannt werden, also „Anreicherungen“ des ADT.

Domain-Gleichungen

Implementation

Dateiname

Der Dateiname dieser DSL-Spezifikation sollte die Endung **.vdm** haben.

Die Regeln, nach der Sie die Domain-Gleichungen „übersetzen“, finden Sie im Abschnitt 3.3.1.

→ 3.3.1

3.2.3 Der VDM-Compiler

Bei den ersten beiden Punkten haben Sie sich vielleicht die folgenden Fragen gestellt:

- „Wer oder was generiert denn das Quellmodul, das meiner Anwendung die benötigten Operationen mit den entsprechend angepaßten Namen, Parameterlisten und Resultattypen bereitstellt ?“
- „Wer oder was kann denn etwas mit der DSL-Spezifikation anfangen ?“

Die Antwort auf beide Fragen lautet: der **VDM-Compiler**. Er liest Ihre DSL-Spezifikation (und moniert eventuelle Fehler), holt aus der VDM-Bibliothek die gewünschten Module heraus und ändert das, was geändert werden muß. Daraus entsteht eine *C*-Deklarationsdatei (eine sogenannte *Header-Datei*) und ein ADT-Implementations-Modul in *C*.

Die Header-Datei enthält die Deklarationen der Funktionen, die die Operationen realisieren, und die Definitionen der Typnamen, die Sie in der DSL-Spezifikation vergeben haben und mit deren Hilfe Sie in Ihrer Applikation Objekte der ADT erzeugen können.

Das ADT-Implementations-Modul enthält den Quellcode von Operationen und Funktionen für die dynamische Speicherverwaltung der Objekte, die von den Operationen benötigt werden, über die aber keine Informationen in der Header-Datei stehen (weil sie in anderen Modulen nicht bekannt zu sein brauchen). Kenntnisse über den Inhalt des ADT-Implementations-Moduls benötigen Sie für die Programmierung Ihrer Applikation *nicht* !

3.2.4 Informationen mit *vdminfo*

Eine weitere Frage drängt sich Ihnen vielleicht noch auf:

„Wenn die VDM-Bibliothek so viele Implementationen der einzelnen Basistypen enthält und so viele Operationen — wie kann ich mir denn da Auskünfte holen, z.B. über neu hinzugekommene Implementationen oder über die Syntax der Operationen ?“

Dokumentation

Auch daran wurde gedacht. Die Dokumentation der VDM-Bibliothek ist ob ihrer Größe natürlich sehr wichtig; also wurde die Möglichkeit vorgesehen, die Dokumentation *in* der Bibliothek abzuspeichern. Bei Änderungen (durch befugtes Personal) muß die Dokumentation aktualisiert werden, so daß sie lückenlos den kompletten Inhalt beschreibt. Gleichzeitig wurde ein Programm namens **vdminfo** erstellt, um diese Dokumentation auch nutzen zu können.

→ 8

Das Kapitel 8 befaßt sich ausführlich mit diesem Programm. Dort ist auch die Struktur der VDM-Bibliothek erläutert (für die Programmierung mit VDM benötigen Sie *keinerlei* Wissen über die interne Struktur — nur der VDM-Compiler greift auf die Bibliothek zu).

3.3 Programmentwicklung mit VDM

Nachdem Sie den Aufbau des VDM-Systems, wie es in der FH Wedel installiert ist, kennengelernt haben, interessiert Sie sicherlich eine zusammenhängende Darstellung der Programmentwicklung mit diesem VDM-System. In diesem Abschnitt werden die einzelnen Phasen schematisch dargestellt und erläutert. Ein Hinweis: Die Dateinamen, die zur Illustration des Ablaufs benutzt werden, sind natürlich nur Beispielsnamen.

Bildelemente

Zuvor eine Erläuterung der Bildelemente in den folgenden Schemata:

- Gestrichelte Vierecke: Phasen der Programmentwicklung
- Dünn umrahmte Vierecke: Dokumente, die innerhalb von Phasen entstehen (und nur dort benötigt werden)
- Dick umrahmte Vierecke: Dokumente, die als Ergebnis von Phasen entstehen (jede Phase bearbeitet Ergebnisse früherer Phasen)
- Quer-Ovale: Arbeitsabläufe innerhalb von Phasen (Teilphasen)
- Rauten: Programme, die Phasenergebnisse verarbeiten
- Pfeile: Sie symbolisieren den Weg der Dokumente
- Hoch-Ovale: Modul-Bibliotheken

3.3.1 Design-Phase – Detailentwurf

Abb. 3.1 stellt die **Design-Phase** dar, in der der Detailentwurf der Applikation erfolgt. Als Eingabe dient das Ergebnis der vorherigen Phase, der

Festlegung des Systementwurfs. Der erste Schritt ist die Erstellung der entsprechenden VDM-Spezifikation in META-IV. Durch die Konkretisierung mit Hilfe der schrittweisen Verfeinerung entsteht eine implementierbare Spezifikation (die Korrektheit jedes Verfeinerungsschrittes kann — wie schon erwähnt — bewiesen werden). Als Ergebnis entstehen die Domain-Gleichungen für die benötigten ADT und die Spezifikation der semantischen Funktionen mit deren Struktur, d.h. dem Kontrollfluß der Applikation.

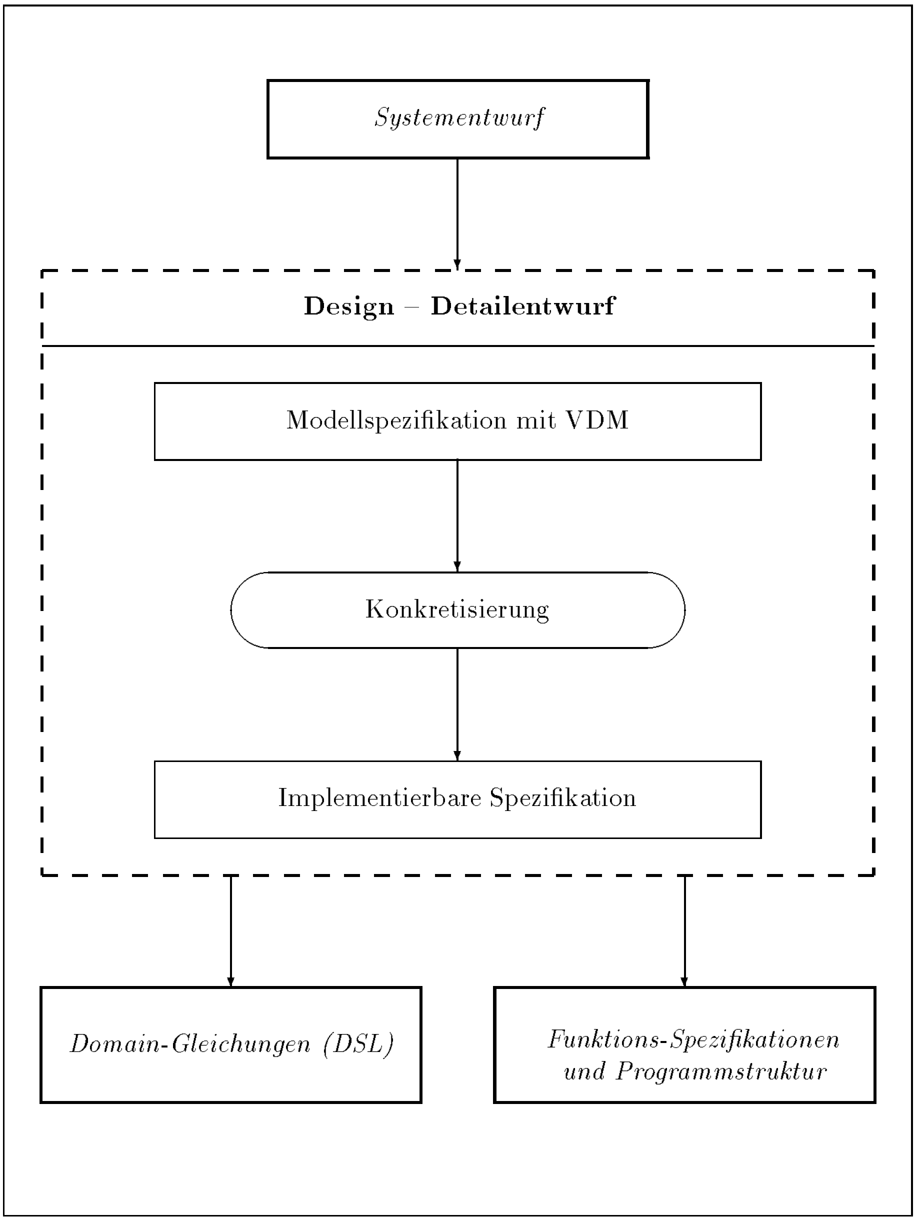

Abbildung 3.1: VDM in der Design-Phase

! → Anmerkung: Nicht immer ist es möglich oder sinnvoll, die gesamte Applikation mit Hilfe von VDM zu spezifizieren — Trivialfunktionen sollten Sie wie gewohnt „direkt“ in C entwerfen und implementieren. Stellen Sie aber sicher, daß die Schnittstellen zwischen den verschiedenen Teilen des Programms konsistent sind.

Übersetzung der Domain-Gleichungen

Die „Papier-Notation“ der Domain-Gleichungen ist leider nicht maschinenlesbar. Mit geringem Aufwand können Sie sie aber konvertieren — so entsteht der erste Teil der DSL-Spezifikation mit dem Namen **My_ADT.vdm** .

Regeln zur Konvertierung

- Komponenten von zusammengesetzten ADT dürfen nicht ebenfalls zusammengesetzt sein — Verschachtelungen müssen Sie jetzt in einzelne ADT zerlegen.
- Jeder ADT-Name hat mindestens zwei Zeichen; das erste ist ein Großbuchstabe, die weiteren sind Kleinbuchstaben, Ziffern oder Unterstriche.
- Anonyme Trees sind nicht erlaubt, denn jeder ADT muß einen Namen haben.
- Die Komponenten eines Tree müssen Selektornamen[4] erhalten. Grund: Die Selektion der einzelnen Datenelemente wird durch eine Anzahl von C-Funktionen realisiert, die der Anzahl an Datenelementen entspricht. Die Erweiterung des generischen Namens der Selektions-Operation nur um den Typ des Datenelementes reicht daher nicht aus, weil zwei Datenelemente den gleichen Typ haben könnten.
  Beispiel: Atree :: s_e1 : E1   s_e2 : E2   s_e3 : E1 ...
- Die Selektoren von Trees müssen mit `s_` anfangen; sie tragen damit den gleichen Namen wie die zugehörigen Selektions-Operationen. Die weiteren Zeichen dürfen Kleinbuchstaben, Ziffern und Unterstriche sein.
- Alle Selektoren in allen Trees einer abstrakten Syntax müssen einzigartig sein, weil ihr Name jeweils ohne weitere Änderung zu einem C-Funktionsnamen wird.
- Auch die Unions können Sie mit expliziten Selektoren definieren. Da eine Union mit zwei identischen Elementtypen allerdings nicht sinnvoll ist (da sie ja immer nur *ein* Objekt enthält), dürfen Sie die Selektoren auch weglassen; in diesem Fall generiert der VDM-Compiler aus den Namen der Elementtypen die Namen der Selektions-Operationen (durch Voranstellen von „`s_`“).
- Als Ersatzsymbol für das Map-Zuordnungszeichen $\xrightarrow[m]{}$ verwenden Sie `-m->`
- Tuples definieren Sie wahlweise mit **Atuple** $\star$ oder **Atuple** $+$

[4] Bezeichner der Datenelemente

3.3.2 Die Codier-Phase

Abb. 3.2 beschreibt die Codier-Phase, die auf die Design-Phase folgt. Die entstandene Struktur- und Funktions-Spezifikation wird in der Zielsprache — in diesem Fall C — implementiert. Die Programmiererin legt nach einer Analyse der Programmanforderungen für jeden ADT die geeignetste Implementation aus dem zur Verfügung stehenden Vorrat fest und trägt ihren Namen zusammen mit den generischen Namen der Operationen, die sie für die Implementation der semantischen Funktionen benutzt, in den zweiten Teil der DSL-Spezifikation ein, die Implementations-Beschreibungen. Die Codierung der Applikation ist noch unvollständig, denn sie ruft die ADT-Operationen auf, ohne daß sie bereits implementiert sind — ein Mangel, der aber im nächsten Schritt vom VDM-Compiler beseitigt wird !

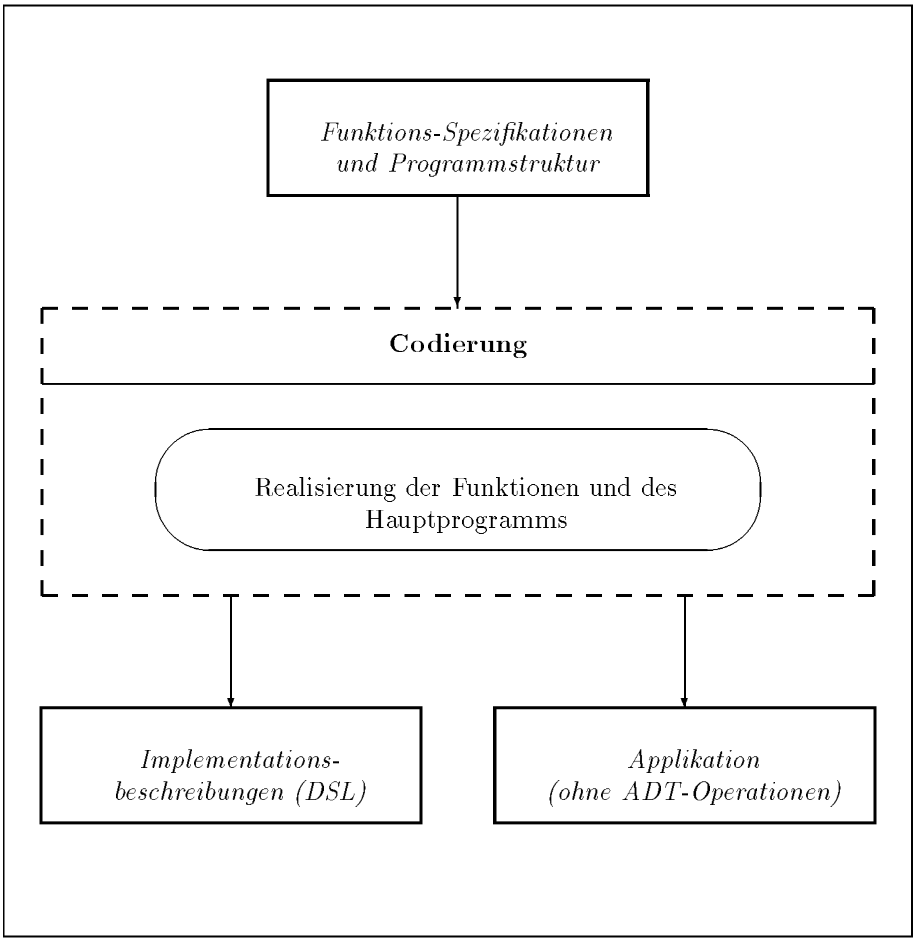

Abbildung 3.2: VDM in Detailentwurf und Codierung

Dateien Bei der Codierung der Applikation erstellen Sie in der Regel mehrere C-Module **My_Prog1.c , . . . , My_Prog_n.c** . Außerdem vervollständigen Sie die DSL-Spezifikation **My_ADT.vdm** .

3.3.3 Übersetzung

VDM-Compiler

Abb. 3.3 zeigt die Arbeitsweise des **VDM-Compilers**. Die Domain-Gleichungen werden mit den Implementations-Beschreibungen zur DSL-Spezifikation zusammengefügt, mit deren Hilfe der VDM-Compiler die benötigten Operationen aus der VDM-Bibliothek liest und sie zum ADT-Implementations-Modul zusammenfügt; als weiteres Ergebnis entsteht eine Header-Datei, die die Operationen in anderen Modulen bekannt macht.

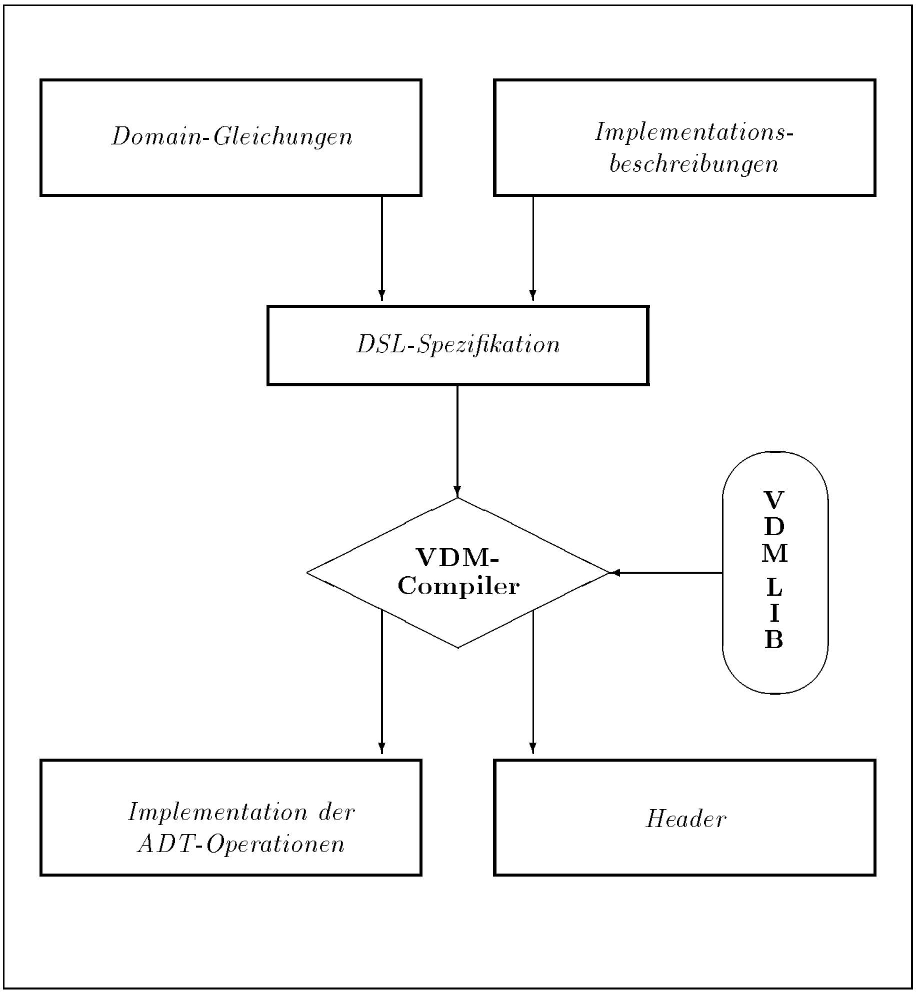

Abbildung 3.3: VDM: Tätigkeit des VDM-Compilers

Die DSL-Spezifikation braucht nicht in einer Datei vorzuliegen; Sie können sie auf mehrere Dateien verteilen und sie dann gemeinsam übersetzen, indem Sie dem VDM-Compiler alle Dateien gleichzeitig als Parameter mitgeben (siehe auch Anhang D).

! $\rightarrow$ Das Schema vereinfacht den Ablauf etwas: der VDM-Compiler kann nicht in *einem* Durchlauf beide Dateien erstellen, sondern braucht für jede Datei einen Durchlauf.

Dateien Das ADT-Implementations-Modul erhält den Namen **My_ADT.c**, die Header-Datei heißt **My_ADT.h** .

Optionen des VDM–Compilers

Sie müssen dem VDM-Compiler unter anderem mitteilen, welche Datei er erzeugen soll; er kennt daher *Optionen*, die seine Arbeit steuern.

Option –**c** Der VDM-Compiler erstellt aus der angegebenen Datei das ADT-Implementations-Modul.

Option –**h** Der VDM-Compiler erstellt aus der angegebenen Datei die Header-Datei.

Option –**e** Der VDM-Compiler erstellt aus der angegebenen Datei die **Enrichment-Datei**. Was ist das ? Die Operationen sind zum Teil nicht in der Lage, allein ihre Aufgabe zu erfüllen, sie benötigen andere Operationen dafür. Damit *Sie* nicht gezwungen sind, diese Beziehungen beim Schreiben der DSL-Spezifikation zu berücksichtigen, erzeugt der VDM-Compiler temporär eine erweiterte, „angereicherte“ Zwischendatei, in der wirklich *alle* benötigten Operationen stehen (er erfährt diese internen Beziehungen aus der VDM-Bibliothek). Das muß er aber bei *jeder* Bearbeitung der DSL-Spezifikation tun, also mindestens zweimal — für Header-Datei und ADT-Implementations-Modul! Dieser zeitaufwendige Vorgang kann verkürzt werden, wenn der VDM-Compiler diese Zwischendatei dauerhaft erzeugt, als sogenannte Enrichment-Datei — sie erhält die Standard-Endung **.enr** . Zur Erzeugung der beiden anderen Dateien geben Sie dann die Enrichment-Datei
Option –**n** als Eingabedatei des VDM-Compilers an mit der Option **–n**; damit sorgen Sie dafür, daß der VDM-Compiler die angegebene Enrichment-Datei nur liest, sie aber *nicht* neu erzeugt. Diese Option spart Zeit, zeigt aber erst bei größeren DSL-Spezifikationen echte Vorteile. Sie sollten Sie bevorzugt im Rahmen einer Projektverwaltung mit MAKE verwenden — wie, sehen Sie dann im
→ 3.3.4 Abschnitt 3.3.4 .

Option –**o** Der VDM-Compiler gibt der erstellten Datei den Hauptnamen, den Sie mit der Option **–o Name** festlegen. Die Dateiendung entspricht jeweils dem gewünschten Datei-Typ (Header, ADT-Implementations-Modul oder Enrichment-Datei).

Option –**v** Der VDM-Compiler zeigt auf dem Bildschirm an, was er gerade tut.

Für alle Optionen finden Sie hier und im Rahmen der Erläuterung von MAKE Beispiele.

Aufruf Sie rufen den VDM-Compiler **vdm** mit folgendem Befehl auf:

vdm [Optionen] Spezifikations-Datei(en)

Unser Beispiel:

Header-Datei My_ADT.h : **vdm** -h My_ADT.vdm -o My_ADT

Implementations-Datei My_ADT.c : **vdm** -c My_ADT.vdm -o My_ADT

Mit einer Enrichment-Datei:

1. **vdm** -e My_ADT.vdm -o My_ADT
2. **vdm** -c -n My_ADT.enr -o My_ADT
3. **vdm** -h -n My_ADT.enr -o My_ADT

! → Sie müssen in jedes Modul der Applikation, in dem eine Operation aufgerufen wird (also wahrscheinlich in alle), die Header-Datei einbinden, um die Operationen und die Namen der von Ihnen definierten ADT im Modul bekannt zu machen. Dazu ist folgender Befehl nötig:

#include "My_ADT.h"

C-Compiler

Abb. 3.4 schließlich zeigt, wie der unter *UNIX* verfügbare *C*-Compiler namens **cc** aus den entstandenen Modulen und seiner Standard-Laufzeit-Bibliothek die fertige, lauffähige Applikation erstellt. Aus Vereinfachungsgründen ist im Schema nur *ein* Quellmodul für die Applikation dargestellt.

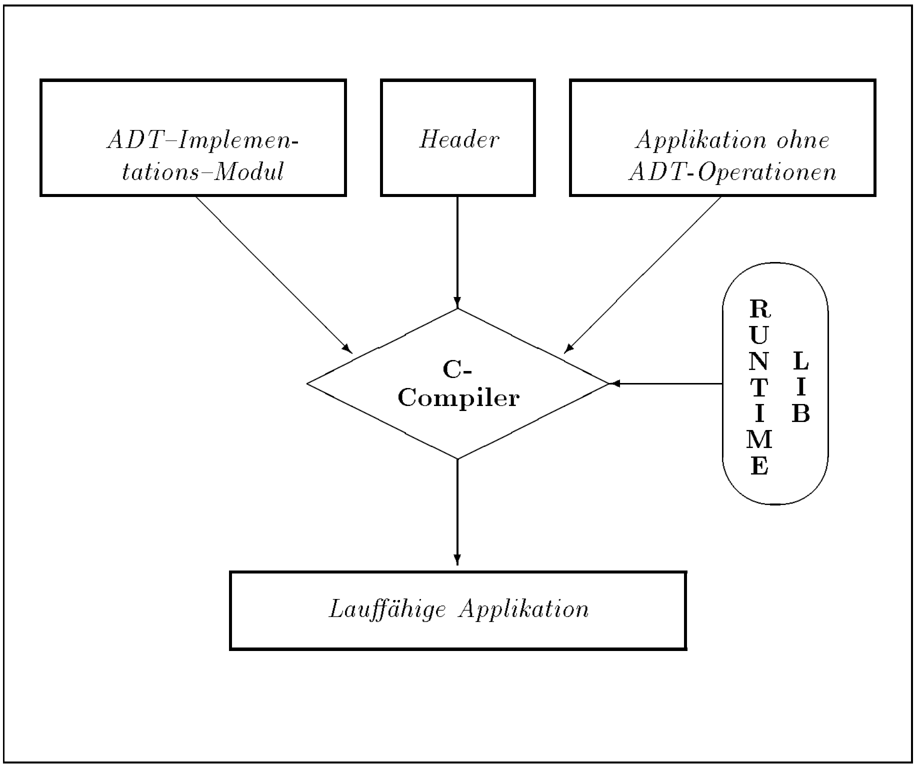

Abbildung 3.4: VDM: Tätigkeit des C-Compilers

Dateien

Der C-Compiler bearbeitet also folgende Dateien: **My_Prog1.c** , ... , **My_Prog_n.c** , **My_ADT.c** und **My_ADT.h** sowie eventuelle applikationsspezifische Header-Dateien.

cc übersetzt die gewünschten Quellmodule einzeln in Objekt-Module, die ausführbaren Code enthalten und den gleichen Hauptnamen wie die Quellmodule haben, aber die Endung **.o** . Sie sind allein allerdings nicht ausführbar — in einem zweiten Schritt bindet der C-Compiler sie zu einem ausführbaren Programm zusammen, das im Normalfall den Namen a.out erhält.

Aufruf des C-Compilers

Aufruf des C-Compilers:

cc [Optionen] Datei-1 Datei-2 ...

Für unser Beispiel:

cc My_Prog1.c My_Prog2.c ... My_Prog_n.c My_ADT.c -o My_Prog

Option –o

Mit der Option **–o Programmname** gibt der C-Compiler dem Programm einen von Ihnen angegebenen Namen statt des normalen **a.out**, die Option hat hier also einen ähnlichen Zweck wie beim VDM-Compiler.

Nur Binden ?

Der C-Compiler kann noch mehr: er kann auch fertige Objekt-Module zusammenbinden (ohne vorher zu übersetzen). Dazu geben Sie einfach nur die Namen der Objekt-Module statt der Quellmodule an.

Beispiel:

Sie haben Ihr Quellmodul **My_Prog1.c** geändert und wollen das Programm neu erstellen. Nur dieses Quellmodul muß neu übersetzt werden, die anderen sind unverändert geblieben, also brauchen nur deren Objekt-Module eingebunden zu werden; dazu geben Sie die Namen der Objekt-Module ein.

cc My_Prog1.c My_Prog2.o ... My_Prog_n.o My_ADT.o -o My_Prog

Option –c

Eine zweite nützliche Option ist **–c**, die den C-Compiler veranlaßt, die übersetzten Module *nicht* zu binden. Das ist hilfreich, wenn Sie eine Änderung in einem Quellmodul Ihres Programms vorgenommen haben und es nur ins Objektmodul übersetzen wollen, ohne Binden. Wenn Sie das Programm MAKE zur Projektverwaltung verwenden wollen (und das ist zu empfehlen), werden Sie diese Option benötigen.

Die übrigen Optionen des C-Compilers können Sie ebenfalls verwenden, sie sind allerdings im Rahmen der Programmierung mit VDM nicht notwendig. Für weitere Informationen sei daher auf das Handbuch zum C-Compiler verwiesen.

3.3.4 Projektverwaltung mit *make*

Sie wissen jetzt, wie man die einzelnen Module erzeugt und übersetzt. Auch kleinere Änderungen in *einem* Quellmodul machen Ihnen keine Sorgen. Aber was ist, wenn eine Änderung z.B. in der DSL-Spezifikation nötig wird ?

„Kein Problem, ich rufe einfach nochmal den VDM-Compiler auf (natürlich mehrmals) und übersetze alle Module, die **My_ADT.h** verwenden, nochmal !“

Sehr gut, aber auch sehr aufwendig — stellen Sie sich vor, Sie machen die dritte kleine Änderung an der DSL-Spezifikation und übersetzen *alles* per Hand zum dritten Mal neu...

„Das könnte man ja durch ein Shell-Script automatisieren !“

Auch richtig — und in diesem Falle auch sinnvoll, da ja tatsächlich alles neu übersetzt werden muß. Wenn Sie aber in einem Quellmodul eine kleine neue Funktion definieren, die in anderen Quellmodulen benutzt werden soll, dann ändern sich zwar einige Quellmodule und eine anwendungsspezifische Header-Datei, die diese neue Funktion nach „außen“ bekannt macht — vielleicht müssen aber nicht einmal alle Quellmodule neu übersetzt werden, und die DSL-Spezifikation bleibt davon auch unberührt. Natürlich wollen Sie nur die Quellmodule neu übersetzen, die auch wirklich von der Änderung betroffen sind; dazu müssen Sie sich in Ihren Modulen sehr gut auskennen (und Lust haben, die entsprechenden Aufrufe der Compiler jedesmal einzugeben), oder Sie benutzen ein Programm, dem Sie *einmal* mitteilen, wie die einzelnen Module verknüpft sind und das dann automatisch mit geringstem Aufwand eine neue Version Ihrer Applikation erstellt.

Dieses Programm heißt *make*. Die Verknüpfungen der Module beschreiben Sie in einer Datei, der **Make-Datei**, und *make* kann erkennen, welche Module neu übersetzt werden müssen, um die neue Version zu erzeugen.

Make-Datei

Eine Make-Datei besteht aus sogenannten **Produktionen**, die für jede beteiligte Datei angeben, von welchen anderen Dateien sie *unmittelbar* abhängt und wie sie aus diesen Dateien erzeugt wird — zunächst also das Hauptprogramm, dann die Module, von denen das Hauptprogramm direkt abhängt, dann die von diesen Modulen wiederum direkt abhängigen Module usw.

Format einer Produktion

Das allgemeine Format einer Produktion sieht so aus:

$\mathsf{Zieldatei} \quad : \quad \mathsf{Datei}_1\ \mathsf{Datei}_2 \ldots \mathsf{Datei}_n \quad ; \quad \mathsf{Kommando}$

Die Zieldatei hängt direkt von Datei_1, Datei_2 usw. ab; das Kommando erzeugt die Zieldatei aus diesen Dateien (und eventuell anderen) neu. Die Produktion braucht nicht in einer Zeile zu stehen, wenn am Ende jeder Zeile (außer der letzten) das Fortsetzungszeichen „\“ steht (der *Backslash*).

Das klingt kompliziert, ist es aber nicht. Die Beispiele in Abb. 3.5 und Abb. 3.6 sollen es Ihnen verdeutlichen; im zweiten wird mit Hilfe einer Enrichment-Datei gearbeitet.

Beispiel 1

Das eigentliche Programm besteht aus zwei Quellmodulen (z.B. Implementa-

```
My_Prog      :  My_Prog1.o My_Prog2.o My_ADT.o \
             ;  cc -o My_Prog My_Prog1.o My_Prog2.o My_ADT.o
My_Prog1.o   :  My_Prog1.c My_Prog.h My_ADT.h \
             ;  cc -c My_Prog1.c
My_Prog2.o   :  My_Prog2.c My_Prog.h My_ADT.h \
             ;  cc -c My_Prog2.c
My_ADT.o     :  My_ADT.c My_ADT.h \
             ;  cc -c My_ADT.c
My_ADT.c     :  My_ADT.vdm \
             ;  vdm -v -c My_ADT.vdm -o My_ADT
My_ADT.h     :  My_ADT.vdm \
             ;  vdm -v -h My_ADT.vdm -o My_ADT
```

Abbildung 3.5: Make-Datei für VDM

tion der semantischen Funktionen und Steuerprogramm) und einer Header-Datei (z.B. zur Deklaration der semantischen Funktionen). Das Hauptprogramm My_Prog hängt direkt nur von den einzelnen Objekt-Modulen ab; um es zu erzeugen, wird der C-Compiler zum Binden der Objekt-Module aufgerufen.

Die Objekt-Module My_Prog1.o und My_Prog2.o hängen jeweils von ihrem Quellmodul, der programmspezifischen Header-Datei My_Prog.h und der aus der DSL-Spezifikation entstandenen Header-Datei My_ADT.h ab; sie werden mit dem C-Compiler nur neu übersetzt (ohne Binden, daher die Option **-c**).

Das Objektmodul My_ADT.o hängt vom ADT-Implementations-Modul und von der zugehörigen Header-Datei My_ADT.h ab, die beide aus der DSL-Spezifikation erzeugt werden. Es wird ebenfalls mit dem C-Compiler neu übersetzt.

Das ADT-Implementations-Modul und die zugehörige Header-Datei hängen ausschließlich von der DSL-Spezifikation My_ADT.vdm ab. Sie werden jeweils durch einen Lauf des VDM-Compilers erzeugt.

make, Aufruf

Nachdem Sie diese Make-Datei geschrieben haben, können Sie sicher sein, daß Sie nach jeder Änderung in einer zur Applikation gehörenden Datei durch den Aufruf

make -f Make-Datei

die aktuellste Version der Applikation erzeugen. Eine Erleichterung: wenn Sie der Make-Datei den Dateinamen makefile geben, genügt der Aufruf von *make* ohne Parameter!

```
Test          : Test.o Test_ADT.o \
              ; cc -o Test Test.o Test_ADT.o
Test.o        : Test.c Test_ADT.h \
              ; cc -c Test.c
Test_ADT.o    : Test_ADT.c Test_ADT.h \
              ; cc -c Test_ADT.c
Test_ADT.c    : Test_ADT.enr \
              ; vdm -v -c -n Test_ADT.enr -o Test_ADT
Test_ADT.h    : Test_ADT.enr \
              ; vdm -v -h -n Test_ADT.enr -o Test_ADT
Test_ADT.enr  : Test_ADT.vdm \
              ; vdm -v -e Test_ADT.vdm -o Test_ADT
```

Abbildung 3.6: Make-Datei mit Erzeugung einer Enrichment-Datei

3.4 Vor- und Nachteile von VDM

Fangen wir mit den Vorteilen an:

- Die VDM-Basistypen decken in der Praxis fast alle Anforderungen ab; obwohl ihre Anzahl relativ gering ist, können problemlos auch große Systeme spezifiziert werden, z.B. die formale Definition von Programmiersprachen als Grundlage für Compiler — ein Auslöser für die Entwicklung von VDM.
- Die ADT sind sehr gut wiederverwendbar, weil sie einer spezifischen Problemstellung durch Auswahl der geeigneten Implementation (mit weiteren Möglichkeiten zur Adjustierung) angepaßt werden können.
- Der schnelle Austausch der spezifischen Implementation eines ADT ist durch Änderung *einer* Zeile in der DSL-Spezifikation möglich.
- Die Implementationen in der Bibliothek sind intensiv getestet, ihre Korrektheit kann daher angenommen werden. Die Operationen der ADT brauchen Sie also nicht mehr zu testen.
- Da Sie die Speicherverwaltung der dynamischen Datenstrukturen, die die ADT repräsentieren, nicht selbst zu programmieren brauchen, sinken die Entwicklungszeit und die Fehler bei der Arbeit mit Pointern drastisch.
- Neue, raffinierte Algorithmen und Datenstrukturen zur Implementation von ADT oder auch neue universelle ADT können leicht in die Bibliothek aufgenommen werden.

- Die Realisierung von komplexen Datenorganisationsformen, die auf rekursiven Strukturen beruhen, wird wesentlich erleichtert, da die Anwenderin mit den normalerweise dafür nötigen Pointern nichts zu tun hat. Der Test ist einfacher und die Wiederverwendbarkeit aufgrund der abstrakten Realisierung hoch — und so allgemeine und effiziente Strukturen wie Binärbäume, B-Bäume usw. werden sehr häufig benötigt.

- Die entstehenden Programme sind fast gezwungenermaßen gut modularisiert und dadurch sehr übersichtlich. Spätere Änderungen sind leicht möglich.

- Die im Entwicklungsprozeß entstehende Dokumentation ist ausreichend exakt zur Beschreibung des Systems; zusätzliche informale Dokumentation ist zwar durchaus sinnvoll, aber nicht unbedingt nötig.

VDM hat auch einige Nachteile — sie entstehen allerdings meist durch die Einbindung in C:

- Die Modellierung von Performance-Anforderungen ist nicht möglich. Erst nach Fertigstellung eines lauffähigen Programms können Sie ermitteln, ob das geforderte Laufzeit-Verhalten erreicht wird. Durch die Auswahl einer anderen, zeitlich günstigeren Implementation für einen ADT kann das Problem aber meist gelöst werden.

- Die Realisierung der Operationen eines ADT durch Funktionen bringt wegen der vielen Unterprogrammaufrufe einen recht hohen Verwaltungsaufwand mit sich.

- Die Erweiterung der Operationsnamen um den Namen des ADT führt in der Regel zu sehr langen Funktionsnamen.

4. Domain Specification Language (DSL)

Die DSL wurde geschaffen, um dem VDM-Compiler die ADT-Definitionen in kompakter Form zur Verfügung zu stellen. Eine DSL-Spezifikation besteht aus zwei Teilen: die *domain equations* (Domain-Gleichungen) definieren die abstrakte Syntax der Applikation, die *implementation descriptions* (Implementations-Beschreibungen) geben für jeden mit einer Domain-Gleichung spezifizierten Datentyp die gewünschte Implementation und den Satz benötigter Operationen an.

Da Sie die von VDM bereitgestellten abstrakten Datentypen — die VDM-Basistypen — noch nicht im einzelnen kennen (die genaue Beschreibung finden Sie in Kapitel 5), werden Sie dieses Kapitel vielleicht nicht beim ersten Durchlesen verstehen. Da aber bei der Beschreibung der VDM-Basistypen von der DSL bereits Gebrauch gemacht wird, sollten Sie sich vorher ein wenig damit vertraut gemacht haben. Wenn Ihnen alle ADT geläufig sind, können Sie dieses Kapitel ja ein zweitesmal durcharbeiten.

4.1 Die Gesamtstruktur

Die DSL-Spezifikation wird in einer separaten Datei gespeichert und hat folgendes Format:

DOMAINS

Domain-Gleichungen

IMPLEMENTATIONS

Implementations-Beschreibungen

Nomenklatur

Da diese Datei die Eingabedatei des VDM-Compilers ist, sollte ihr Name die Standard-Erweiterung **.vdm** haben. Der Hauptname der Datei ist natürlich beliebig; wenn Sie aber einen einzelnen, universell zu verwendenden ADT losgelöst von einer Anwendung definieren, sollte der Hauptname der Datei dem Namen dieses ADT entsprechen !

Beispiel:

Für die DSL-Spezifikation des universellen Datentyps *Btree* sollte folgender Name gewählt werden:

Btree.vdm

Sie können die Spezifikation auch auf mehrere Dateien verteilen, z.B. um einen von Ihnen geschaffenen, universellen Typgenerator separat von seinen verschiedenen Elementtypen zu vereinbaren; Sie brauchen dann für eine neue Anwendung nicht die „alte“ DSL-Spezifikation zu ändern, sondern erstellen nur eine neue Spezifikation für den Elementtyp und übersetzen beide gemeinsam. Näheres dazu finden Sie auch im Anhang D.

4.2 Die Domain-Gleichungen

Für jeden benötigten ADT muß dessen interne Struktur mit Hilfe der VDM-Basistypen, die ja Typgeneratoren darstellen, angegeben werden — dazu dienen die Domain-Gleichungen.

Nomenklatur

Die Nomenklatur empfiehlt für Namen von Domains normale C-Bezeichner mit einigen Einschränkungen: der erste Buchstabe muß groß sein, alle weiteren klein, der Unterstrich darf ebenfalls vorkommen. Die Beispielsnamen wurden entsprechend dieser Regel möglichst „sprechend“ gewählt.

Hier sind nun die Formate der Domain-Gleichungen für die VDM-Basistypen, jedoch nicht als Syntax-Definition[1], sondern in Form von Beispielen. Die Bedeutung der ADT wird dabei kurz beschrieben. Die ADT sind Typgeneratoren, weil die Datentypen ihrer Elemente frei gewählt werden können.

Set

Der Set ist die Klasse von Mengen (im mathematischen Sinn) von Elementen.

Set = Element –**set**

Tuple

Der Tuple ist die Klasse von Sequenzen von gleichartigen Elementen.

Tuple = Element $\star$ *oder*

Tuple = Element +

Map

Die Map ist die Klasse von endlichen Mengen von Paaren von Elementen (partielle, funktionale Zuordnung).

Map = Argumentwert `-m->` Map-Wert

Tree

Der Tree ist die Klasse von endlichen geordneten Sammlungen einzeln benannter Komponenten eventuell unterschiedlichen Typs.

Selektornamen

Alle Komponenten müssen benannt sein. In der VDM-Spezifikation ist das nicht gefordert; Sie müssen dies jetzt nachholen. Ein Komponentenname (Selektorname) beginnt *immer* mit der Zeichenfolge „s_“, darauf folgt zunächst ein Kleinbuchstabe, dann können noch weitere Kleinbuchstaben, Ziffern oder Unterstriche folgen.

[1] Im Anhang befindet sich eine Beschreibung der DSL-Syntax in Backus-Naur-Form.

Tree :: s_eins : Element1
s_zwei : Element2
⋮
s_n : Elementn

Union Die Union ist die Klasse von Objekten, die mehrere Datentypen haben können; zu jeder Zeit enthält ein Union-Objekt genau *ein* Objekt mit einem der möglichen Datentypen.

Union1 = s_eins : Typ1 |
s_zwei : Typ2 |
⋮
s_n : Typn

Bei Unions ist es erlaubt, die Elemente anonym zu lassen; der VDM-Compiler generiert in diesem Fall Namen, indem er vor den (in Kleinbuchstaben umgewandelten) Elementnamen die Zeichen „s_" stellt. Das ist natürlich nur möglich, wenn nicht eine andere anonyme Union die gleichen Datentypen verwendet, da sonst doppelte Bezeichner auftreten!

Union2 = Element1 |
Element2 |
⋮
Elementn

Optional Das Optional ist eine Klasse von Objekten, die zu einem bestimmten Zeitpunkt entweder ein Objekt des Elementtyps oder den speziellen Wert **nil** enthalten.

Optional = [Element]

Elementare Elementare Datentypen (Elementary) gibt es in VDM auch: es sind Zeichen, Zahlen (mit verschiedenen Wertebereichen) und logische Wahrheitswerte. Außerdem gibt es einen vordefinierten Datentyp „String", der als Tuple von Zeichen vereinbart ist; er ist zwar nicht elementar, steht aber immer zur Verfügung.

Nat1	Natürliche Zahlen ab 1
Nat0	Natürliche Zahlen ab 0
Intg	Ganze Zahlen
Bool	**true** und **false**
Char	ASCII-Zeichen
Str	Zeichenketten

Token Ein Token ist ein Typ, für den nur Konstruktoren und Tests auf Gleichheit / Ungleichheit existieren. Eine Repräsentation ist nicht definiert; Sie können damit z.B. Pointer oder Unterbereichstypen definieren (Anwendungen finden Sie im Abschnitt 5.3.4).

Token = ELEMENTARY

! → Es gibt eine Restriktion bezüglich der Verschachtelung von Domains innerhalb einer Domain-Gleichung: Komponenten von zusammengesetzten Domains dürfen nicht ebenfalls zusammengesetzt sein. Dies müssen Sie bei der Umsetzung Ihrer VDM-Spezifikation beachten; dort ist die Verschachtelung von ADT ja beliebig möglich. Die Zerlegung sollten Sie trotzdem erst während der Erstellung der DSL-Spezifikation vornehmen, da die Struktur des von Ihnen geschaffenen ADT klarer ist, wenn sie „komplett“ in einer Domain-Gleichung vorliegt! Warum müssen Verschachtelungen zerlegt werden ? Weil Sie für jeden beteiligten VDM-Basistyp festlegen müssen, wie er implementiert werden soll — dafür muß aber jeder VDM-Basistyp auch einen Namen haben!

Beispiel: Aus der Definition

A = [Nat0 | Intg] ⋆

müssen Sie die folgenden drei Definitionen machen:

A = B ⋆
B = [C]
C = Nat0 | Intg

4.3 Die Implementationsbeschreibungen

Für jeden definierten ADT müssen Sie eine Beschreibung geben, wie dieser realisiert werden soll und welche Operationen sie benötigen — dazu dienen die Implementationsbeschreibungen.

Struktur Eine Implementationsbeschreibung hat die folgende Struktur:

Domain = Implementation + Enrichment + ...

Implementation Die erste Angabe ist der Name der gewünschten **Implementation** des zugrundeliegenden ADT, möglicherweise gefolgt von einer Liste mit Parametern, die bestimmte Aspekte der Repräsentation noch „adjustieren“ können.

Für jeden ADT gibt es einen individuellen Satz von Implementationen. Zur Information über die vorhandenen Implementationen können Sie auch das Online-Hilfesystem *vdminfo* (siehe dazu Kapitel 8) heranziehen.

→ Kapitel 8

Mit der Angabe der Implementation definieren Sie nur die grundlegenden Datenstrukturen mit den nötigen Datenelementen zu ihrer Verwaltung sowie interne Kontrollfunktionen, die von den Operationen benötigt werden, nicht aber die Operationen selbst; aus Gründen der Speicherplatzersparnis sollten Sie nämlich nur diejenigen Operationen in Ihr Programm einbinden, die Sie auch wirklich benutzen!

Enrichments

Daher folgt nun die Angabe der benötigten Operationen, den sogenannten **Enrichments**, also „Anreicherungen" des ADT. Sie brauchen nur den generischen Namen angeben, also den Grundnamen ohne Spezialisierung für den spezifischen, von Ihnen definierten Domain-Namen (Sie wissen doch: der Quelltext der Operation wird vom VDM-Compiler auf den von Ihnen definierten Domain zugeschnitten, was die Parameter und den Funktionswert betrifft; dabei wird der generische Name der Operation durch Erweiterung mit dem Domain-Namen spezialisiert — siehe dazu Abschnitt 3.2.1). Auch Enrichments sind zum Teil mit Parametern noch spezifisch steuerbar.

→ 3.2.1

Zusätzliche Enrichments

Sie können übrigens für einen ADT mehrere Implementationsbeschreibungen angeben — eine häufige Anwendung dafür ist die Erweiterung eines allgemein verwendbaren ADT um einige Enrichments, die Sie nur in *einer* Applikation benötigen; die Änderung der allgemeinen Beschreibung des ADT ist nicht nötig, Sie „erweitern" sie einfach um eine weitere Beschreibung. Da sie die Implementation nicht ändern dürfen (sonst beklagt sich der VDM-Compiler über eine mehrdeutige Implementationsbeschreibung), können Sie sie auch weglassen; Ihre Beschreibung beginnt dann mit einem Plus-Zeichen. Beispiel: Sie möchten den Datentyp Str um die Operationen puts und fwrite erweitern. Sie schreiben dann

Str = + puts + fwrite

Parameter

Parameter müssen Sie innerhalb von runden Klammern angeben. Die meisten Parameter sind optional und haben Default-Werte; daher brauchen Sie sie für eine „Standard-Konfiguration" nicht anzugeben. Welche Parameter angegeben werden müssen, hängt natürlich von der gewählten Implementation ab und ist daher im Anhang C angegeben. Zur Definition von Parametern gibt es fünf Möglichkeiten:

→ Anhang C

1. PARAMETER = " String "

   Dieser String kann beliebige Zeichen außer Anführungsstrichen (") und Zeilenwechseln enthalten. Welche Strings sinnvoll sind, hängt natürlich davon ab, um welchen Parameter es sich handelt.

2. PARAMETER = 0

   Damit wird der Parameter „undefiniert", der VDM-Compiler ignoriert ihn; so können Default-Werte ungültig gemacht oder globale Definitionen per Kommandozeilen-Option des VDM-Compilers für eine spezielle Implementation ignoriert werden.

3. PARAMETER = Zahl

   Diese Zahl ist eine nicht-negative, ganze Zahl. Beachten Sie: Wollen Sie den Wert Null zuweisen, geben Sie bitte «00» ein, eine einfache «0» sorgt ja dafür, daß der Parameter undefiniert ist!

4. PARAMETER

   Das ist eine Kurzform für die Zuweisung PARAMETER = 1.

5. PARAMETER = Name

   Ein String, der nur aus Buchstaben und Ziffern besteht, heißt auch „Name". Solche Strings können auch ohne Anführungsstriche zugewiesen werden.

Der VDM-Compiler prüft, ob ein angegebener Parameter zugelassen ist; wenn nicht, erhalten Sie eine Fehlermeldung.

Beispiel: Zum Abschluß noch ein Beispiel für eine komplette DSL-Spezifikation. Wenn Sie mit den ADT noch nicht vertraut sind, werden Sie es vielleicht nicht verstehen; es soll auch nur dazu dienen, einen Eindruck vom Aussehen einer solchen Spezifikation zu bekommen.

DOMAINS

```
Tuple  =  Nat0 *
Map    =  Nat0  -m->  Tuple
```

IMPLEMENTATIONS

```
Bool   =  + and + not + true + false
Tuple  =  array
          + is_eq + len
          + del1 + copy1
          + mk_empty + mk_one + conc + sel + hd + tl
Map    =  linklist(ALLOC="malloc(SAVE)")
          + is_eq + is_in + is_in_dom
          + card + sel + del + copy
          + mk_empty + mk_one
```

Abbildung 4.1: Beispiel für eine DSL-Spezifikation

5. Die VDM–Basistypen

Vorbemerkungen

Reihenfolge

Die FH-Installation stellt sieben VDM-Basistypen zur Verfügung, die in folgender Reihenfolge in den Abschnitten dieses Kapitels dargestellt sind:

- ELEMENTARY (Char, Nat1, Nat0, Intg, Bool)
- Set
- Tuple
- Map
- Optional
- Tree
- Union

Aufbau

Die Abschnitte über Sets, Tuples, Maps, Optionals, Trees und Unions sind folgendermaßen aufgebaut:

- Kurze Beschreibung, Definition eines Domains in der DSL und Anwendungsmöglichkeiten des ADT
- Implementationen
- Operationen: Konstruktoren, Kombinatoren, Prädikate, Attribute,...

Operationen

Konstruktoren erzeugen neue Objekte. **Kombinatoren** erstellen neue Objekte aus bestehenden Objekten bzw. Objekten und Elementen. **Prädikate** geben zweiwertige Eigenschaften von Objekten (wahr oder falsch) wieder. **Attribute** umfassen alle mehrwertigen Eigenschaften eines Objekts. Weitere Operationen sind von Typ zu Typ unterschiedlich und bilden je nach Bedarf weitere Kategorien.

! →

Die Operations-Namen werden hier in der generischen Form vorgestellt — vergessen Sie nicht, sie in Ihrer Anwendung in der erweiterten Form zu benutzen !

Zu den VDM-Basistypen Set, Tuple, Map und Tree finden Sie außerdem größere Anwendungsbeispiele mit Lösungen in *C*. Sie umfassen jeweils die DSL-Spezifikation und die Codierung der semantischen Funktionen. Die Operations-Namen sind dann natürlich schon erweitert.

Zunächst erfahren Sie allerdings etwas über die grundlegenden Mechanismen der Speicherverwaltung, da die strukturierten ADT auf dynamischen Datenstrukturen basieren und Sie häufig selbst dafür sorgen müssen, daß Ihre Objekte zerstört bzw. nicht zerstört werden.

Danach werden die wenigen Operationen, die für alle VDM-Basistypen definiert sind, kurz zusammengefaßt.

5.1 Speicherverwaltung

Pointer

Die Verwaltung dynamischer Datenstrukturen geschieht mit Hilfe von *Pointern*, das sind „Zeiger“ auf die Objekte in Form von Adressen. In *C* haben Sie die Möglichkeit, viel mit Pointern zu arbeiten; aber Pointer-Operationen sind sehr fehlerträchtig und erfordern einen sorgfältigen Test. Das VDM-System erleichtert Ihnen die Programmierung ungemein, da die Verwaltungsfunktionen für dynamische Datenstrukturen in der VDM-Bibliothek enthalten sind. Sie vereinbaren eine Variable des gewünschten Typs und brauchen sich nicht mit den Pointern zu beschäftigen — VDM macht das für Sie. Die Funktionen sind sorgfältig getestet, so daß Sie sich auf deren Korrektheit verlassen können.

Problem !

Pointer haben jedoch einen schwer zu umgehenden Nachteil: eine Operation kann, wenn Sie einen Pointer auf ein Objekt als Argument erhält, dieses Objekt *verändern*. Das ist in vielen Fällen aber nicht erwünscht; meist soll die Operation ein neues, verändertes Objekt liefern, *ohne* daß das alte Objekt dauerhaft geändert wird.

Lösung !?

Ein möglicher Ausweg: in jeder Operation der VDM-Bibliothek wird eine vollständige Kopie des Objekts erzeugt und bearbeitet, während das „Original“ unberührt bleibt. Am Ende der Bearbeitung wird diese Kopie, wenn nötig, wieder gelöscht. Das ist einfach, hat aber einen großen Nachteil: der Zeitaufwand jeder kleinen Operation, die schreibend auf ein Objekt zugreift, wird unverhältnismäßig hoch, die Performance des gesamten Programms wird unnötig schlecht; denn *manchmal* darf das Objekt ruhig verändert werden, weil Sie das Original später nie wieder benötigen — in diesem Falle sollte das Objekt vollständig gelöscht werden, um keinen kostbaren Speicher zu verschwenden.

Lösung !!

Die Operationen, die ihr Argument verändern, sorgen dafür, daß das Original dabei „zerstört“ wird, d.h. daß es danach keinen definierten Inhalt hat und auch keinen Speicherplatz mehr belegt. Auch diese Lösung hat einen Nachteil:

wenn Sie das Objekt nach dem Aufruf einer solchen Operation noch weiter benötigen, sind *Sie* gezwungen, vorher eine Kopie davon anzufertigen. VDM stellt dafür zwei Operationen zur Verfügung:

- **copy** erstellt eine *vollständige* Kopie des Objekts, d.h. mit allen Elementen; sind diese ebenfalls strukturiert, werden auch deren Sub-Strukturen kopiert usw.

- **copy1** erstellt eine Kopie des Objekts selbst; die Sub-Strukturen werden nicht mitkopiert. In der Praxis wird dieser Befehl meist ausreichen; er ist natürlich nicht so zeitaufwendig wie copy.

Die Abb. 5.1 stellt die Wirkung der beiden Kopierbefehle grafisch dar.

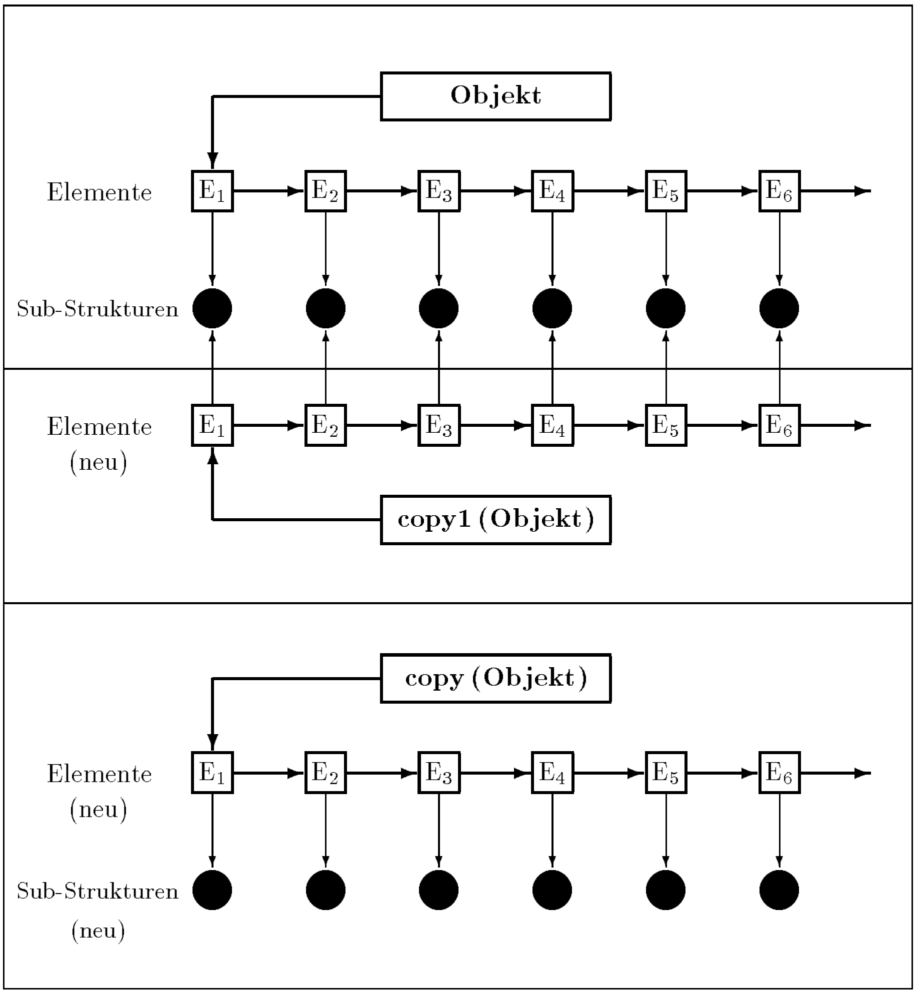

Abbildung 5.1: Wirkung der Operationen **copy1** und **copy**

Zerstörung ? Wenn eine Operation ein Objekt „zerstört“ (in der Befehlsreferenz im Anhang B ist das jeweils angegeben), stellt sie sicher, daß dessen Speicherplatz vorher freigegeben wird. Ein Zugriff nach der „Zerstörung“ führt allerdings zu unvorhersagbaren Ergebnissen — meist erzeugt er einen Laufzeitfehler.

Löschen

Nachdem Sie eine Kopie eines Objekts angelegt und eine solche „zerstörende“ Operation (vorzugsweise natürlich mit der Kopie!) verwendet haben, brauchen Sie sich danach nicht mehr um diese Kopie zu kümmern — sie belegt dann ja keinen Speicher mehr. Schreiben Sie aber selbst eine solche zerstörende Funktion, müssen Sie auch dafür sorgen, daß das Objekt seinen Speicherplatz wirklich *komplett* wieder freigibt (z.B. häufig erforderlich bei rekursiven Funktionen, wie im Beispiel auf S. 5.4.4 im Abschnitt über Sets zu sehen). Dafür gibt es komplementäre Operationen:

- **del** löscht ein Objekt inklusive aller Sub-Strukturen; del ist in der Regel nur nach einem vorangegangenen copy sinnvoll.
- **del1** löscht dementsprechend nur das Objekt selbst, nicht aber dessen Sub-Strukturen; del1 ist damit das Pendant zu copy1.

! → Versuchen Sie nie, ein durch eine Operation zerstörtes Objekt „sicherheitshalber“ noch mit del oder del1 zu löschen!

Operationen mit impliziten Löschaktionen

Es gibt Operationen für strukturierte Objekte, bei denen Elemente aus den Objekten entfernt werden, z.B. die Tuple-Operation tl, die das Ende eines Tuple ohne das erste Element liefert. Alle diese Operationen zerstören das Objekt, das sie manipulieren. Ein zusätzliches Problem ist dabei aber, daß Element-Objekte (Sub-Strukturen) danach eventuell keine Zuordnung mehr zu einem strukturierten Objekt haben, z.B. das erste Tuple-Element nach einer tl-Operation. Häufig wird vor einer solchen Operation das von der Löschung betroffene Element referenziert, d.h. es wird ein Objekt vom Elementtyp eingerichtet, das dieses Element bezeichnet; z.B. wird durch die Tuple-Operation hd ein Verweis auf das erste Element eingerichtet. Wenn ein solches Objekt allerdings *nicht* mehr anderweitig referenziert ist, sollte es zerstört werden, um keinen Speicherplatz zu verschwenden.

Daher gibt es zu allen diesen Operationen ein Pendant, das die ausgegliederten Elemente implizit löscht; ihr Name entspricht der „normalen“ Variante, nur der Buchstabe „d“ wird angefügt — z.B. wird aus tl dann tld.

Beispiel: Sie möchten in einem Tuple der Reihe nach das jeweils erste Element-Tuple löschen, bis es weniger als vier Elemente (deren Typ hier egal ist) enthält. Da Sie die gelöschten Elemente nicht mehr benötigen, verwenden Sie die Operation tld:

```
Tup = Elem ⋆
   ⋮
Tup t;
   ⋮
while (len_tup (t) > 3)
   t = tld_tup (t);
```

Auch Operationen, die auf den ersten Blick keine Elemente aus strukturierten Objekten entfernen, können von dieser Regelung betroffen sein: bei der Vereinigung von Sets z.B. wird ein neues strukturiertes Objekt geschaffen — da aber Elemente, die in *beiden* Sets vorkommen, nur einmal in die Vereinigung aufgenommen werden, wird jeweils eines dabei überflüssig und kann implizit gelöscht werden.

Ausnahmen

Wann dürfen Sie das implizite Löschen nicht verwenden ? Wenn Sie selbst eine Funktion schreiben, die ihr Argument zerstört, und Sie die (strukturierten) Objekte, auf die Sie diese Funktion anwenden wollen, mit copy1 kopiert haben, aber nicht dessen Sub-Strukturen, würden Sie mit implizitem Löschen die Sub-Strukturen des *Originals* vernichten — das Original wäre inkonsistent.

Faustregel

Gibt es in Ihrem Programm irgendwo noch ein Objekt, das auf von einem impliziten Löschen betroffene Elemente verweist, sollten Sie die „Standardversion“ ohne implizites Löschen verwenden.

Diese Operationen sind betroffen:

- Sets: diffd, intersd, sub1d, union1d, uniond
- Tuples: ovwrt1d, rmv1d, tld
- Maps: intersd, ovwrt1d, ovwrtd, sub1d, subd, union1d, uniond

5.2 Generelle Operationen

Neben den im vorigen Abschnitt bereits angesprochenen Kopier- und Lösch-Operationen zur Speicherverwaltung gibt es noch einige andere universell verfügbare Operationen, die Sie hier an zentraler Stelle zusammengefaßt finden. Auch in der Befehlsreferenz im Anhang B sind diese Operationen ausgegliedert.

- **pr** und **print** sorgen für die Ausgabe eines beliebigen Objekts, pr formatiert zusätzlich. Sie können diese Operationen während des Programmtests sinnvoll einsetzen; in der fertigen Applikation sollten sie aber nicht mehr vorkommen, denn sie kosten viel Speicherplatz und reichen für eine wirklich „schöne“ Ausgabe nicht.

- **hash** wird intern verwendet, um strukturierte Objekte schnell und speicherplatzsparend abzuspeichern. Sie können zwar auf der Basis dieser Operation eigene Speicherverwaltungsfunktionen schreiben; die VDM-Operationen sind allerdings schon sehr gut und vor allem stabil. So ist der Einsatz von hash nur selten sinnvoll.

Die übrigen generellen Operationen werden Sie in der Regel nicht benötigen, denn sie dienen nur als Hilfsfunktionen für andere, wichtigere Operationen.

5.3 Elementare Datentypen

Die elementaren Datentypen entsprechen den elementaren Zahlentypen in C. Die VDM-Datentypen sind jedoch (etwas abstrakter) an die mathematischen Bezeichnungen angelehnt. Auch in VDM gibt es die durch Zahlen repräsentierten Datentypen wie das Zeichen und den logischen Wahrheitswert (Bool). Wie sie durch C-Datentypen repräsentiert werden und welche speziellen Anwendungen die verschiedenen Implementationen bieten, erfahren Sie in diesem Abschnitt.

Normalerweise brauchen Sie sich um die Implementation der vordefinierten Datentypen nicht zu kümmern, da VDM sie in jede Applikation einbindet. Sie können daraus aber eigene Datentypen ableiten; wie, das erfahren Sie hier ebenfalls.

5.3.1 Die vordefinierten Zahlentypen

Vorweg sei gesagt, daß Sie mit den Zahlentypen wie gewohnt rechnen können — alle mathematischen Operationen können mit den VDM-Zahlentypen umgehen und auch die logischen Funktionen haben die erwarteten Resultate (bei Bedingungen in den verschiedenen Schleifenstrukturen usw.).

Der Vollständigkeit halber gibt es drei Vergleichsoperationen für die Zahlentypen:

- **is_eq** $(x_1 = x_2\ ?)$
- **is_ge** $(x_1 \geq x_2\ ?)$
- **is_gr** $(x_1 > x_2\ ?)$

Diese Operationen stehen zum Teil auch für die strukturierten Datentypen zur Verfügung und basieren dann auf dem Vergleich der Elemente mit Hilfe der entsprechenden Operationen für den Elementtyp.

Der Datentyp *Char*

VDM bietet für den Umgang mit Zeichen — wie auch *C* — einen speziellen, recht kleinen Zahlentyp an, der gerade für den Zeichensatz des Computers ausreicht, also 256 Zeichen umfaßt. Char steht in der Implementation „unsigned“ zur Verfügung und wird mit dem Datentyp „unsigned char“ dargestellt. Natürlich können Sie auch Zahlen in einem Char abspeichern, wenn Sie Speicherplatz sparen möchten; aus Gründen der Verständlichkeit sollten Sie aber davon Abstand nehmen und Zahlen tatsächlich durch Zahlentypen repräsentieren.

Der Datentyp *Nat1*

In der Zahlentheorie, einem Zweig der Mathematik, werden die verschiedenen Zahlenarten in einige Mengen unterteilt. Die anschaulichste ist die Menge $\mathcal{N}_1$ der natürlichen Zahlen (ohne die Null), denn sie repräsentiert alles, was abzählbar ist — die Objekte, die Null-mal da sind, können ja nicht gezählt werden.

VDM stellt diese Menge als Datentyp bereit, er heißt **Nat1**. Sie können ihn mit verschiedenen Implementationen darstellen, am sinnvollsten jedoch ist die Implementation „unsigned“, die den größtmöglichen Wertebereich für Nat1 zur Verfügung stellt — leider kann die theoretisch unendliche Menge der natürlichen Zahlen per Computer nicht abgebildet werden. Standardmäßig wird Nat1 durch den *C*-Datentyp „unsigned long“ repräsentiert.

Der Datentyp *Nat0*

Die nächstgrößere zahlentheoretische Menge entsteht durch die Hinzunahme der Null: $\mathcal{N}_0$ enthält also alle nicht-negativen ganzen Zahlen.

In VDM wird diese Menge durch den Datentyp **Nat0** repräsentiert, der wie **Nat1** intern durch „unsigned long“-Werte dargestellt wird.

Der Datentyp *Intg*

Die ganzen Zahlen bilden die nächstgrößere Zahlenmenge $\mathcal{Z}$. Hinzu kommen also negative Zahlen.

VDM stellt dafür den Datentyp **Intg** zur Verfügung. Die Implementation „unsigned“ ist dafür natürlich ungeeignet; stattdessen wird die Implementation „scalar“ verwendet, die Intg mit dem Datentyp „signed long“ darstellt.

5.3.2 Der logische Datentyp *Bool*

Prädikate und Vergleichsoperationen liefern entweder „wahr“ oder „falsch“. In *C* werden alle „falschen“ Resultate durch die Zahl Null dargestellt, während ein beliebiger Wert ungleich Null „wahr“ bedeutet.

VDM bietet Ihnen für logische Daten den Datentyp **Bool** an, der sich genauso verhält wie die entsprechenden Hilfskonstruktionen in *C* und demzufolge auch mit diesen kompatibel ist. Häufig wird in *C* folgendes vereinbart:

```
typedef BOOL unsigned char;
#define TRUE (1 == 1)
#define FALSE (0 == 1)
```

Der VDM-Datentyp Bool sollte an die Stelle von „BOOL“ treten; *TRUE* und *FALSE* und auch die in *C* vorhandenen logischen Operatoren können Sie aber auch weiterhin so verwenden. VDM stellt aber auch entsprechende Operationen zur Verfügung, die z.B. in mit Funktionalen erzeugten Funktionen mit einem Bool-Resultat eingesetzt werden können; eine Anwendung dafür finden Sie bei den Sets im Abschnitt 5.4.

- **true ()** liefert den logischen Wert „wahr“
- **false ()** liefert den logischen Wert „falsch“
- **not (x)** invertiert einen logischen Wert; in *C* kann dies mit dem Operator „**!**“ erreicht werden.
- **and (x,y)** verknüpft zwei logische Werte mit UND, ist also nur dann „wahr“, wenn x UND y „wahr“ sind. In *C* kann dies mit dem Operator „**&&**“ erreicht werden.
- **or (x,y)** verknüpft zwei logische Werte mit ODER, ist also dann „wahr“, wenn x ODER y ODER beide „wahr“ sind. In *C* kann dies mit dem Operator „**| |**“ erreicht werden.
- **xor (x,y)** verknüpft zwei logische Werte mit EXKLUSIV–ODER, ist also dann „wahr“, wenn genau einer der beiden Werte x ODER y „wahr“ ist. In *C* kann dies mit dem Test auf Ungleichheit („**!=**“) erreicht werden.

Bool-Werte werden durch den kleinstmöglichen Datentyp („unsigned char“) in der Implementation „bool“ dargestellt, der speziell an logische Werte angepaßt ist (z.B. was die Ausgabe von Bool-Werten betrifft).

5.3.3 Der vordefinierte Datentyp *Str*

Ein strukturierter Datentyp ist im VDM-System immer definiert: **Str**, die Klasse der Zeichenketten. Sie sind als Tuples implementiert und nutzen einige Besonderheiten der Tuple-Implementation „array“ aus; da sie sich von gewöhnlichen *C*-Strings unterscheiden, werden sie zur Abgrenzung als **VDM-Strings** bezeichnet. Es stehen Operationen zur Verfügung, um bequem mit diesen VDM-Strings zu arbeiten; außerdem existieren Schnittstellen zu den normalen *C*-Strings.

→ 5.5 Im Abschnitt 5.5 über Tuples wird näher auf VDM-Strings und ihre Besonderheiten eingegangen.

5.3.4 Selbstdefinierte elementare Datentypen

Unterbereiche von Zahlenmengen

Wenn Sie nur einen bestimmten Unterbereich einer Zahlenmenge benötigen, können Sie ihn selbst definieren. Sie geben ihm einen Namen, tragen in der Domain-Gleichung den Datentyp „ELEMENTARY" ein und wählen eine der Implementationen „unsigned" (bei einem positiven Unterbereich) oder „scalar" (wenn auch negative Werte im Unterbereich enthalten sind). Zusätzlich setzen Sie die Implementationsparameter „LB" und „UB", die Unter- und die Obergrenze des gewünschten Bereichs. Der VDM-Compiler wählt automatisch den kleinstmöglichen Datentyp aus, mit dem der Unterbereich darstellbar ist.

Beispiel: Sie benötigen den Zahlenbereich von eins bis einunddreißig, um Monatslängen zu verwalten.

```
DOMAINS
Days = ELEMENTARY
IMPLEMENTATIONS
Days = unsigned(LB=1,UB=31)
```

Der Datentyp „Days" wird dann mit Hilfe von „unsigned char"-Variablen repräsentiert.

Token

Ein interessanter Sonderfall ist das **Token** — es gibt keine Definition, wie es repräsentiert wird, es ist nur sichergestellt, daß zwei Token auf Gleichheit getestet werden können.

Geeignete Implementationen für Token sind „unsigned" und „scalar", denn sie enthalten eine Operation new, mit der ein neuer, vorher noch nie verwendeter Wert erzeugt wird.

Damit können Sie einen Aufzählungstyp definieren, wie er in *Pascal* existiert: Sie erzeugen in der Applikation die Anzahl benötigter Token in entsprechenden Variablennamen und ordnen auf unterster Ebene der Applikation den darzustellenden Zuständen einen der Token-Namen zu. In den höheren Verarbeitungsfunktionen können Sie dann die Token-Namen statt nichtssagender Zahlencodes benutzen.

Beispiel: Das klingt sehr abstrakt; am Beispiel eines Tischrechners, der die vier Grundrechenarten beherrscht, soll dies verdeutlicht werden. Die Operatoren sollen im Programm durch die Namen PLUS, MINUS, MULT, DIV angesprochen werden. Die Eingabefunktion wertet die eingegebenen Operatoren aus und ordnet ihnen das entsprechende Token zu.

```
DOMAINS
Token = ELEMENTARY
IMPLEMENTATIONS
Token = unsigned + new
⋮
In der Applikation:
Token PLUS,MINUS,MULT,DIV;
PLUS = new_token ();
MINUS = new_token ();
MULT = new_token ();
DIV = new_token ();
⋮
Token input_operator (void)
{
  Einlesen des Operatorsymbols und Abspeichern in op
  switch (op) {
      '+': return PLUS;
      '−': return MINUS;
      '×': return MULT;
      '÷': return DIV;
  }
}
```

Heap

Eine andere Anwendung für Token ist die Modellierung der *Heap-Speicherverwaltung* mit einer Map, die ebenso schnell ist wie die „direkte" Verwaltung in C (mit malloc und free). Sie bietet allerdings die Möglichkeit, durch Austausch der Map-Implementation wesentlich mächtigere Operationen und mehr Sicherheit beim Umgang mit Pointern zu erreichen, wenn die Performance-Anforderungen das erlauben. Im Abschnitt 5.6 über Maps erfahren Sie mehr darüber. Da die Token in diesem Fall Adressen von dynamischen Objekten enthalten, sollten Sie die Implementation „ptr" wählen. Den Datentyp, auf den die Pointer weisen, legen Sie mit dem Implementationsparameter „ELEM" fest.

Selbstdefinierte C-Strukturen

Wenn Sie eine bestehende Applikation mit VDM unterstützen möchten, werden Sie bereits einen Satz an Datenstrukturen mit den Mitteln von C definiert haben. Um diese auch von VDM-Operationen verarbeiten lassen zu können, müssen Sie sie „importieren". Manchmal ist das auch erforderlich, wenn Sie Manipulationen durchführen wollen, für die VDM keine Unterstützung bietet; die Datei-Operationen von VDM sind beispielsweise nur bedingt verwendbar.

Hier brauchen Sie eine Schnittstelle, um VDM-Daten an *C*-Funktionen zu übergeben, die diese Manipulationen durchführen.

Eine Möglichkeit ist die Definition eines elementaren Datentyps in VDM, der die gewünschte Struktur aufnehmen kann, für alle VDM-Operationen aber unteilbar ist. Dies erreichen Sie mit Hilfe der Implementation „unsigned“ und dem Parameter „CSTRUCT“, der die Typdefinition enthält. Ein Beispiel: Sie möchten eine sequentielle Datei von Datensätzen bearbeiten, die Sie komplett in den Speicher laden und dort in einer Baumstruktur halten, um eine Sortierung der Datensätze zu ermöglichen. Diese Baumstruktur erzeugen Sie mit den VDM-Basistypen *Tree* und *Optional*, aber alle Dateioperationen führen Sie mit normalen *C*-Strukturen aus. Eine DSL-Spezifikation dafür sieht so aus:

```
#typedef struct { ...} RECORD;
DOMAINS
Btree  =  [ Node ] ...
Node   ::  s_lt : Btree   s_data : Data   s_rt : Btree
Data   =  ELEMENTARY
IMPLEMENTATIONS
Btree  =  ptelem + ...
Node   =  ptstruct + ...
Data   =  unsigned (CSTRUCT="RECORD")
```

Der Btree enthält außer seinen Verwaltungsinformationen Datenknoten vom Typ Node. Die in einem Node existierenden Daten vom Typ Data sind für die VDM-Operationen unteilbar. Seine interne Struktur ist daher uninteressant, wichtig ist nur die Größe eines Objekts. Für *C*-Funktionen aber ist ein Data-Objekt eine Struktur und kann mit den üblichen Methoden manipuliert werden.

5.4 Sets

5.4.1 Was ist ein Set ?

Definition Ein **Set** ist eine endliche, ungeordnete Sammlung von unterschiedlichen, aber dennoch gleichartigen Objekten. Etwas ausführlicher:

- Die Objekte gehören der gleichen Kategorie, also dem gleichen Datentyp an.
- Ein Set enthält immer nur eine *begrenzte* Anzahl von Objekten.
- Nie sind zwei gleiche Objekte in einem Set enthalten.

- Innerhalb eines Set ist kein Ordnungskriterium für die Objekte definiert; dadurch ist es nicht möglich, z.B. gezielt auf das siebte Objekt zuzugreifen.
- Es gibt auch Sets, die *keine* Objekte enthalten.

Mit diesen Eigenschaften ähnelt ein Set einer mathematischen Menge; diese jedoch kann auch unendlich groß sein (wie z.B. die Zahlenmengen). Die Beschränkung auf endliche Sets ist jedoch im Rahmen der Anwendung in der Programmierung vernünftig. Mathematische Mengen sind grundsätzlich ebenfalls ungeordnet, auch wenn die Zahlenmengen ein Ordnungprinzip aufzuweisen scheinen; dies ist jedoch den Objekten (Zahlen) immanent — Punktmengen sind zum Beispiel ungeordnet.

DSL Um in der DSL-Spezifikation einen Set $Aset$ von Objekten eines Domains $\mathcal{A}$ zu definieren, schreiben Sie

$$Aset \quad = \quad \mathcal{A} - set$$

Anwendung Sie können Sets also einsetzen, wenn Sie Sammlungen von Objekten modellieren möchten, die ungeordnet sind und deren Elemente alle verschieden sind; außerdem sollte die Manipulation eines Sets auf dem Hinzufügen, Entfernen und (zufälligem) Auswählen von Objekten basieren.

Beispiel: Ein kleines Beispiel soll den Einsatz von Sets verdeutlichen: eine Schule besteht aus mehreren verschiedenen Klassen und in jeder Klasse sind verschiedene Schüler. Eine Klasse bildet demzufolge einen Set von Schülern, eine Schule ist ein Set von Klassen. Mehr noch: natürlich bildet auch die Schule einen Set von Schülern, der die gleiche Größe und die gleichen Elemente hat wie die Vereinigung aller Klassen. Die Anzahl der Schüler ist immer endlich und sie unterliegen keinem immanenten Ordnungssystem, also ist die Repräsentation durch Sets gerechtfertigt. Sie können also folgendes modellieren:

$$\begin{array}{lcl} \mathsf{School} & = & \mathsf{Class\ -set} \\ \mathsf{Class} & = & \mathsf{Student\ -set} \end{array}$$

Die Tatsache, daß auch Schulen Sets von Schülern sind, wird hiermit noch nicht wiedergegeben. Dazu könnten Sie einen weiteren Set Schoolofstudents einführen: um eine Schule als Set von Schülern zu betrachten, benötigen Sie dann eine Funktion, die ein Objekt School in ein äquivalentes Objekt des Typs Schoolofstudents überführt.

Während der Erläuterung der Operationen werden Sie diesen Domains immer wieder begegnen.

5.4.2 Set-Implementation

Zur Zeit gibt es für Sets nur die Implementation **linklist**. Sie wird auch für andere VDM-Basistypen verwendet; ihre Charakteristika finden Sie dort oder im Anhang C.

5.4.3 Set-Operationen

Zur Veranschaulichung der Operationen dienen folgende Variablen:

```
School  s1, s2;
Class   c1, c2;
Student st1, st2;
```

Set-Konstruktoren

VDM bietet Ihnen vier Operationen zur Erzeugung von Set-Objekten:

mk_empty Mit **mk_empty** erzeugen Sie einen neuen, leeren Set.
Beispiel — Gründung einer Schule:

```
s1 = mk_empty_school ();
```

mk_one Mit **mk_one** erzeugen Sie einen neuen, ein-elementigen Set.
Beispiel — Eröffnung einer Klasse mit folgender Gründung einer Schule, die diese eine Klasse umfaßt:

```
c1 = mk_empty_class ();
s2 = mk_one_school (c1);
```

copy / copy1 **copy** und **copy1** haben Sie ja bereits kennengelernt — diese Operationen stehen für alle strukturierten Typen zur Verfügung.

Set-Kombinatoren

union Die Vereinigung zweier Sets (s1 ∪ s2) heißt **union** und enthält alle Elemente, die in einem der beiden Sets enthalten sind. Es gibt häufig den Wunsch, dem Set genau ein Objekt hinzuzufügen, also einen ein-elementigen Set. Um nun nicht den Umweg über die Erzeugung eines solchen Set gehen zu müssen,
union1 gibt es dafür die Operation **union1**.
Beispiel — das Hinzufügen einer Klasse zu einer Schule läßt sich auf zwei Weisen darstellen:

```
union1_school (s1,c1) ≡ union_school (s1,mk_one_school (c1))
```

Zusätzlich gibt es die Versionen **uniond** und **union1d** mit implizitem Löschen.

inters Der Durchschnitt zweier Sets (s1 ∩ s2) heißt **inters** und enthält die Ele-

mente, die in beiden Sets enthalten sind. **intersd** löscht implizit.
Beispiel — der Durchschnitt zweier Schulen ist natürlich leer, da keine Klasse zu zwei Schulen gehören kann:

is_empty_school (inters_school (s1,s2)) == TRUE

diff Die Set-Subtraktion (s1 \ s2) heißt **diff** und enthält alle Elemente, die im ersten, aber nicht im zweiten Set enthalten sind. Auch hier gibt es eine Vereinfachung, wenn Sie nur *ein* Element aus einem Set entfernen möchten:
sub1 **sub1**. Hinzu kommen **diffd** und **sub1d**, die implizit löschen.
Beispiel — eine Klasse in einer Schule wird geschlossen:

s1 = sub1_school (s1,c1); oder
s1 = diff_school (s1,mk_one_school (c1));

Set-Prädikate

is_eq Mit **is_eq** testen Sie die Gleichheit zweier Sets, d.h. ob sie die gleiche Kardinalität (Element-Anzahl) haben und die gleichen Elemente enthalten.
Beispiel — zwei Schulen haben nie die gleichen Klassen:

is_eq_school (s1,s2) == FALSE

is_empty Mit **is_empty** testen Sie, ob ein Set gerade keine Elemente enthält.
Beispiel — eine neugegründete Schule hat zunächst keine Klassen:

is_empty_school (mk_empty_school ()) == TRUE

is_in Mit **is_in** können Sie testen, ob ein bestimmtes Element in einem Set enthalten ist ($c1 \in s1$).
Beispiel — ist ein bestimmter Schüler in einer bestimmten Klasse ?

if (is_in_class (st1,c1)) ...

is_subset Mit **is_subset** können Sie testen, ob alle Elemente eines Set auch in einem anderen Set des gleichen Typs enthalten sind (Teilmengen-Beziehung, $c1 \subseteq c2$).
Beispiel — die Schüler einer Klasse sind *nie* vollständig auch in einer anderen Klasse:

is_subset_class (c1,c2) == FALSE

Der leere Set ist in jedem anderen Set enthalten:

is_subset_xxx (mk_empty_xxx (),xxx) == TRUE

Set-Attribute

card Die Kardinalität eines Set, d.h. die aktuelle Anzahl von Elementen, liefert Ihnen **card**. Der leere Set hat die Kardinalität 0.
Beispiel — Ermittlung der Schülerzahl in einer Klasse:

printf ("%d",card_class (c1));

Selektion in einem Set

sel

Mit **sel** wählen Sie zufällig ein Element eines nicht-leeren Set aus. Beispiel — Auslosung des Klassensprechers in einer Schulklasse:

```
st1 = sel_class (c1);
```

Set-Funktionale

Zur Erzeugung neuer, komplexer Funktionen mit Hilfe der vorhandenen gibt es zwei **Funktionale**, das sind Funktionen, die wieder Funktionen zurückliefern. In der DSL-Spezifikation geben Sie dazu den Namen des Funktionals, gefolgt von einem Punkt und dem Namen der zu erzeugenden Funktion an. Wie üblich folgen dann in Klammern die Parameter des Funktionals.

binary

Mit **binary** können binäre Funktionen, die aus einem Set und einem Element einen neuen Set erzeugen, verallgemeinert werden zu Funktionen, die aus zwei Sets einen neuen Set erzeugen und dabei ihre Argumente zerstören. Eine mit binary erzeugte Funktion arbeitet wie folgt:

```
Aset newfct_aset (Aset s1, Aset s2)
{
  Elem e;
  if (is_empty_aset (s2))
      return s1;
  else {
      e = sel_aset (s2);
      return newfct_aset (foldfct_aset (s1,e), sub1_aset (s2,e)) ;
  }
}
```

Parameter

foldfct ist die Funktion, die binary als Parameter übergeben wurde. Sie hat zwei Argumente, einen Set und ein Element, und liefert wieder einen Set; Sie legen sie in der DSL-Spezifikation mit dem Parameter „**FOLDFCT**“ fest. Für die Funktion newfct_aset sieht das so aus:

```
IMPLEMENTATIONS
...+ binary.newfct (FOLDFCT="foldfct")
```

Nomenklatur

Beachten Sie, daß sowohl der Name der erzeugten Funktion (newfct) als auch der Name der Parameterfunktion (foldfct) als generische Namen aufgefaßt werden; sie werden mit dem zugehörigen Set-Namen erweitert. Falls Sie also eine selbstdefinierte Funktion als Parameterfunktion einsetzen möchten, geben Sie ihr den richtigen Namen! Im ADT-Implementations-Modul wird dann eine Funktion ähnlich der obenstehenden erzeugt; damit es auch compiliert werden kann, wird in der Header-Datei für die Parameterfunktion eine Deklaration eingetragen.

Beispiel — die Operation union kann mit Hilfe von binary und union1 definiert werden:

```
IMPLEMENTATIONS
...+ binary.union (FOLDFCT="union1")
```

fold Das Funktional **fold** ist wesentlich komplexer: es hat drei Parameter-Funktionen und erzeugt eine Funktion, die einen Set auf einen durch die Parameter-Funktionen festgelegten Resultattyp abbildet. Damit lassen sich sehr gut komplexe Manipulationen *aller* Elemente einer Menge realisieren. Die Parameter-Funktionen haben folgende Bedeutung:

Parameter

1. Die *Konvertierungsfunktion* (**CVFCT**) bildet ein Element des Set auf den Resultattyp ab (ist das Resultat vom Elementtyp, kann diese Funktion wegfallen).

2. Die *Nullfunktion* (**NULLFCT**) hat kein Argument und liefert konstant den Resultat-Wert für den leeren Set.

3. Die *Fold-Funktion* (**FOLDFCT**) bildet zwei Werte des Resultattyps auf den Resultattyp ab.

RESULT

In Klammern ist der jeweilige Parametername für die DSL-Spezifikation angegeben; hinzu kommt noch der optionale Parameter „**RESULT**“, der den Resultattyp angibt für den Fall, daß es *nicht* der Elementtyp ist.

Die erzeugte Funktion sieht von der Struktur her wie folgt aus:

```
Result newfct_aset (Aset s)
{
  Elem e;
  if (is_empty_aset (s))
      return nullfct_result ();
  else {
      e = sel_aset (s);
      return foldfct_result (cvfct_elem (e),
                             newfct_aset (sub1_aset (s,e)));
  }
}
```

Die Verarbeitung ähnelt also strukturell dem Funktional binary, nur die Resultate werden auf komplexere Weise erzeugt.

Nomenklatur Die Namen der drei Parameterfunktionen und der erzeugten Funktion werden erweitert, wie Sie in der obigen Funktion sehen können:

- Die Konvertierungsfunktion mit dem Namen des Elementtyps
- Die Nullfunktion mit dem Namen des Resultattyps
- Die Fold-Funktion ebenfalls mit dem Namen des Resultattyps

- Die erzeugte Funktion mit dem Namen des Set

Beispiel 1 Eine häufige Anwendung von fold ist die distributive Vereinigung aller Sets in einem Set von Sets, also z.B. die Zusammenlegung aller Klassen einer Schule zu einer neuen Klasse:

IMPLEMENTATIONS
... + fold.d_union (NULLFCT="mk_empty", FOLDFCT="union")
erzeugt
Class d_union_school (School s)

Da der Resultattyp hier identisch mit dem Elementtyp des Set school ist, braucht keine Konvertierungsfunktion angegeben zu werden, ebensowenig wie der Parameter „RESULT".

Beispiel 2 Gelegentlich möchten Sie testen, ob entweder für alle Elemente eines Set ein bestimmtes Prädikat erfüllt ist oder ob es mindestens ein solches Element gibt. Dafür haben sich die folgenden Schreibweisen eingebürgert:

$(\forall\ e \in\ Set)\ (Predicate\,(e))$
$(\exists\ e \in\ Set)\ (Predicate\,(e))$

Mit fold können Sie beide Tests generieren; Sie benötigen dazu Operationen für den Datentyp Bool. „forall" ist für den leeren Set erfüllt, ansonsten wird das Prädikat für jedes Element geprüft und die Resultate AND-verknüpft. „exists" ist für den leeren Set nicht erfüllt, ansonsten werden die Resultate des Prädikats OR-verknüpft.

```
fold.forall  (RESULT="Bool",
              CVFCT="Predicate",
              NULLFCT="true",
              FOLDFCT="and")
fold.exists  (RESULT="Bool",
              CVFCT="Predicate",
              NULLFCT="false",
              FOLDFCT="or")
```

! → Der VDM-Compiler erkennt übrigens nicht, welche Standard-Operationen für irgendwelche mit binary oder fold definierte Funktionen notwendigerweise eingebunden werden müssen ! Sie müssen sie also in Ihrer DSL-Spezifikation explizit anfordern, sonst erhalten Sie einen Fehler beim Binden, z.B. „undefined function mk_empty_class".

Zusätzliche Parameter Manchmal benötigen die Parameterfunktionen ihrerseits noch weitere Argumente; vielleicht ist die Konvertierung eines Elements nur möglich, wenn Zusatzinformationen vorliegen. Die Erzeugung der Deklaration in der Header-Datei ist natürlich nur möglich, wenn der Typ dieser zusätzlichen Argumente bekannt ist; Sie geben dazu dem Funktional fold weitere Parameter:

- Die Parameter **X, XX** und **XXX** enthalten die Datentypen der Zusatzargumente für die Konvertierungsfunktion.
- Die Nullfunktion ist konstant und benötigt daher keine Argumente.
- Die Parameter **Y, YY, YYY, Z, ZZ** und **ZZZ** enthalten die Datentypen der Zusatzargumente für die Fold-Funktion.

Beim Aufruf der erzeugten Funktion geben Sie die zusätzlichen Argumente in der entsprechenden Reihenfolge an, zuerst die mit den **X**..., danach die mit **Y**... und am Ende die mit **Z**... vereinbarten Argumente. Die Funktionsdeklaration sieht dann wie folgt aus:

```
Result newfct_aset (Aset as,
                    X x1,
                    XX x2, ...,
                    Y y1, ...,
                    ZZZ z3)
```

→ 5.6 Eine Anwendung dafür finden Sie im Abschnitt 5.6 über Maps.

Set: Diverse Operationen

forall Mit **forall** wird für alle Elemente eines Set eine angegebene Prozedur ausgeführt, die das aktuelle Element als Argument erhält.
Beispiel — die klassenweise Ausgabe aller Schüler einer Schule mit der Ausgabe-Operation pr:

```
forall_school (s1,pr_class);
```

forsome Mit **forsome** wird mit allen Elementen eines Set, die ein angegebenes Prädikat erfüllen (in Form einer Funktion mit bool-Resultat), eine Prozedur ausgeführt.
Beispiel — die klassenweise Ausgabe aller Schüler einer Schule, wobei leere Klasssen automatisch unterdrückt werden:

```
forsome_school (s1,not_empty_class,pr_class);
```

Definition von not_empty_class:

```
Bool not_empty_class (class c)
{ return !is_empty_class (c) }
```

5.4.4 Beispiel: „Coarsest Partitioning“

Wir haben beliebige, nicht-leere Sets gegeben, die *nicht* überschneidungsfrei sind. Daraus soll nun die „coarsest partitioning“ erzeugt werden — die „gröbste Zerlegung“ dieser Sets, die aus nicht-leeren, aber überschneidungsfreien Sets besteht, bei minimaler Anzahl an Element-Sets. Sie sehen: die

„coarsest partitioning“ ist kein selbständig definierter Begriff, sondern nur relativ zu einem Ausgangszustand definiert. Graphisch veranschaulicht diesen Vorgang Abb. 5.2.

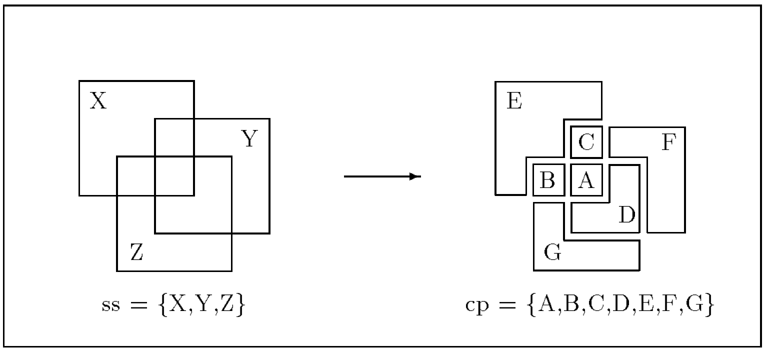

Abbildung 5.2: Beispiel für Sets: das „coarsest partitioning“-Problem

DSL Wir können bereits die DSL-Spezifikation für dieses Problem festlegen: ein Domain **Aset** mit beliebigen Elementen (in unserem Beispiel natürliche Zahlen), ein Domain **Sset** mit Elementen vom Typ **Aset**, also ein Set von Sets. In der Abbildung sind die Sets **ss** und **cp** vom Typ **Sset**, alle übrigen vom Typ **Aset**.

DOMAINS
Aset = Nat0 –set
Sset = Aset –set
IMPLEMENTATIONS
Aset = linklist + ...
Sset = linklist + ...

Der Ablauf einer Zerlegung sieht folgendermaßen aus: solange noch Sets im Gesamt-Set vorhanden sind, deren Durchschnitt nicht-leer ist, wählen wir ein Paar solcher Sets (x,y) aus, entfernen sie aus dem Gesamt-Set $(s \setminus \{x,y\})$ und fügen diesem dann drei neue, nicht-leere Sets hinzu: $(x \setminus y)$, $(y \setminus x)$ und $(x \cap y)$ — falls einer dieser Sets leer ist, wird er ignoriert. Mit diesem neukonstruierten Gesamt-Set verfahren wir ebenso, bis er schließlich kein Paar sich überschneidender Sets mehr enthält. Der dabei entstehende Gesamt-Set ist die „coarsest partitioning“ des Ausgangszustands.

Spezifikation mit VDM Hier sehen Sie die VDM-Funktions-Spezifikation für eine semantische Funktion **decompose** zur Ermittlung der „coarsest partitioning“ unter Verwendung der VDM-Notation.

```
TYPE decompose (Ss):   Sset   ⟶   Sset
decompose =
  if (∃ x,y ∈ Ss) ((x ≠ y) ∧ (x ∩ y) ≠ {} ))
      then (let {x,y} ⊆ Ss be s.t. ((x ≠ y) ∧ (x ∩ y) ≠ {} )) in
          decompose (({x\y , y\x , x∩y} ∪ Ss \ {x,y}) \ {} )
      else Ss
```

Sie lesen die VDM-Spezifikation folgendermaßen: „Für den Fall, daß es noch irgend zwei verschiedene Sets im Gesamt-Set gibt, die sich überschneiden, so wähle x und y so, daß sie zwei sich überschneidende Sets im Gesamt-Set darstellen und führe damit den Aufruf von decompose aus; sonst liefere den Gesamt-Set zurück“. Das Argument des rekursiven decompose–Aufrufs wird wie oben angegeben ermittelt — beachten Sie: eine der beiden Set-Differenzen könnte leer sein, daher wird vom neukonstruierten Set Ss sicherheitshalber der leere Set subtrahiert.

Die semantische Funktion decompose_sset, die ein wohlgeformtes Objekt vom Typ Sset (das nicht-leer ist und nur nicht-leere Element-Sets enthält) in ihre „coarsest partitioning“ umwandelt, codieren wir nun in *C*:

```
#define TRUE 1
#define FALSE 0
Sset decompose_sset (Sset ss)
{
  Aset a1,a2,sav1,sav2;
  Sset ss1,ss2;
  Bool found = FALSE;
  ss1 = copy1_sset (ss);
  while (!found && !is_empty_sset (ss1)) {
      a1 = sel_sset (ss1);
      ss1 = sub1_sset (ss1,a1);
      ss2 = copy1_sset (ss1);
      a2 = find_partner (ss2,a1);
      found = !is_empty_aset (a2);
  }
  del1_sset (ss1);
  if (!found)
      return (ss);
```

```
  else {
      ss = sub1_sset (ss,a1);
      ss = sub1_sset (ss,a2);
      sav1 = copy1_aset (a1);
      sav2 = copy1_aset (a2);
      ss = union1_sset (ss,inters_aset (sav1,sav2));
      sav1 = copy1_aset (a1);
      a1 = diff_aset (a1,a2);
      a2 = diff_aset (a2,sav1);
      del1_aset (sav1);
      if (!is_empty_aset (a1))
          ss = union1_sset (ss,a1);
      if (!is_empty_aset (a2))
          ss = union1_sset (ss,a2);
      return (decompose_sset (ss));
  }
}
```

decompose_sset Zunächst müssen zwei Sets gesucht werden, die sich überschneiden. Dazu wird zunächst eine Kopie des Gesamt-Set angelegt (copy1_sset), danach solange ein Set dem Gesamt-Set entnommen (sel_sset und sub1_sset), bis mit Hilfe der Funktion find_partner (die ihr Argument zerstört, daher die Kopie in ss2), ein solches Paar gefunden wurde (found = TRUE) oder der Gesamt-Set leer ist (is_empty_sset). Im letzteren Fall ist man fertig.

Nachdem die while–Schleife verlassen wurde, ist ss2 zerstört, ss1 aber nicht. Der Aufruf von del1_sset sorgt dafür, daß kein Speicherplatz vergeudet wird. Wie Sie sehen, reicht die Kopie der Haupt-Struktur (mit copy1) aus, da die Sub-Strukturen nicht verändert werden.

Für den Fall, daß ein Paar sich überschneidender Sets existiert, ist es in den Variablen a1 und a2 gespeichert. Nun werden diese beiden Sets — entsprechend dem oben geschilderten Ablauf — aus dem Gesamt-Set entfernt (mit sub1_sset). Danach wird der garantiert nicht-leere Durchschnitt der beiden Sets, der mit inters_aset ermittelt wurde, eingefügt (mit union1_sset). Die beiden Differenz-Sets werden mit Hilfe von diff_aset erzeugt und in denselben Variablen abgelegt. Weil dabei der leere Set entstehen kann, wird mit is_empty_aset überprüft, ob eine Aufnahme in den Gesamt-Set erforderlich ist. Schließlich wird decompose_sset mit diesem veränderten Gesamt-Set rekursiv aufgerufen.

find_partner Die Hilfsfunktion find_partner erledigt die Suche nach einem Set, der mit dem Argument einen nicht-leeren Durchschnitt hat. Sie liefert bei Mißerfolg den leeren Set zurück; das ist sinnvoll, weil die Element-Sets per Definition nichtleer sind !

```
Aset find_partner (Sset ss,Aset as)
{
  Aset sav1,sav2;
  Aset ret;
  if (is_empty_sset (ss))
      return (mk_empty_aset ());
  else {
      ret = sel_sset (ss);
      ss = sub1_sset (ss,ret);
      sav1 = copy1_aset (as);
      sav2 = copy1_aset (ret);
      if (! is_empty_aset (inters_aset (sav1,sav2))) {
          del1_sset (ss);
          return (ret);
      }
      else
          return (find_partner (ss,as));
  }
}
```

Auch find_partner ist eine rekursive Funktion: mit jedem Aufruf wird die Anzahl der Element-Sets im Gesamt-Set um eins kleiner — ist kein Set mehr übrig, war die Suche erfolglos.

Bei jedem Aufruf wird dem Gesamt-Set ein Element-Set entnommen (mit sel_sset und sub1_sset) und überprüft, ob der Durchschnitt mit dem Argument-Set leer ist (mit is_empty_aset und inters_aset) — wenn nicht, ist ein Paar gefunden und der Gesamt-Set muß „per Hand“ zerstört werden (explizit mit del1); wenn doch, wird find_partner rekursiv aufgerufen.

5.5 Tuples

5.5.1 Was ist ein Tuple ?

Definition

Ein **Tuple** ist eine endliche, geordnete, aber unsortierte Sammlung von gleichartigen Objekten, die auch mehrfach auftreten dürfen. Etwas ausführlicher:

- Die Objekte haben den gleichen Datentyp.
- Ein Tuple enthält immer nur eine *begrenzte* Anzahl von Objekten.
- Ein Objekt kann mehrfach an verschiedenen Positionen im Tuple auftreten.
- Ein Objekt hat eine definierte, unveränderliche Position innerhalb eines Tuple. Diese Position wird durch einen numerischen *Index* repräsentiert.

- Die Objekte in einem Tuple sind nicht automatisch sortiert.
- Es gibt auch Tuples, die *keine* Objekte enthalten.

Eine Zweiergruppe von Objekten heißt Paar, eine Dreiergruppe „Tripel“, eine Vierergruppe „Quadrupel“ und eine Gruppe von n Objekten heißt „n-Tupel“; daraus leitet sich der allgemeine Begriff „Tuple“ ab. Andere, passende Bezeichnungen sind z.B. „Liste“, „Sequenz“ oder „eindimensionale Matrix“. Die Objekte sind aufsteigend durchnumeriert und jedem Objekt ist seine laufende Nummer als Index zugeordnet — daraus folgt aber nicht, daß auch die Objekte in irgendeiner Weise sortiert sind !

Sprechweisen

Aus der Numerierung ergeben sich „Namen“ bzw. Sprechweisen für bestimmte Objekte oder Teile des Tuple: das Objekt an der Position 1 heißt auch *head* und ist das „linke“ Ende des Tuple, der Rest des Tuple (ohne den *head*) heißt *tail*, das letzte Objekt heißt dementsprechend *last* und ist das „rechte“ Ende des Tuple. Die Anzahl der Objekte heißt auch „Länge“ des Tuple. Die Nomenklatur der Operationen ist diesen Sprechweisen angepaßt.

Array ?

Tuples erinnern Sie sicherlich an Arrays (Tabellen) in C; tatsächlich können Sie Tuples auch mit deren Hilfe implementieren. Der Unterschied besteht darin, daß Tuples beliebig viele, beliebig komplexe Objekte enthalten können !

DSL

Um in der DSL-Spezifikation ein Tuple *Atuple* von Objekten eines Domains $\mathcal{A}$ zu definieren, schreiben Sie

$$Atuple \quad = \quad \mathcal{A}\star \qquad \text{oder} \qquad Atuple \quad = \quad \mathcal{A}+$$

Anwendung

Wenn Sie Sammlungen von Objekten modellieren, die in einer wohldefinierten Reihenfolge repräsentiert werden sollten, weil sich Bezeichnungen wie das „erste“ Objekt, das „letzte“ Objekt oder auch das „n.te“ Objekt aufdrängen und wenn Sie Manipulationen wie das Aneinanderhängen von solchen Objektsammlungen, den indizierten Zugriff auf ein einzelnes Objekt und Verarbeitung der Objekte in der gegebenen Reihenfolge durchführen möchten, dann ist die Verwendung von Tuples angezeigt.

Beispiel:

Dazu ein Beispiel: Um Texte als definierte Folgen von Wörtern und Zwischenräumen, die wiederum definierte Folgen von Zeichen sind, zu modellieren, können Sie Wörter und Zwischenräume als Tuple von Zeichen, Texte als Tuple von Wörtern und Zwischenräumen auffassen. Hier wollen wir Wörter und Zwischenräume nicht trennen, sie werden also beide als Objekte vom Typ „Wort“ aufgefaßt.

$$\begin{aligned} \text{Text} &= \text{Word} \star \\ \text{Word} &= \text{Char} \star \end{aligned}$$

Natürlich ist ein Text damit selbst eine Sequenz von Zeichen. Sets sind nicht zur Modellierung von Texten geeignet, da Zeichen ja durchaus mehrfach auftreten können und zudem erst ihre Reihenfolge den Wörtern und Texten einen Sinn gibt!

VDM-Strings Dieses Beispiel zeigt Ihnen schon, daß die Verarbeitung von Zeichenketten (Strings), die in vielen Applikationen benötigt wird, am besten mit Tuples erfolgt. Folgerichtig stellt VDM einen vordefinierten Datentyp **Str** bereit, der dem Word im obigen Beispiel entspricht. Die für Strings adäquate Implementation „array" kennt einige Operationen, die den Umgang mit diesen **VDM-Strings** allein und in Verbindung mit normalen *C*-Strings erleichtern; sie sind das Thema eines gesonderten Abschnitts innerhalb der Darstellung der Operationen.

5.5.2 Tuple-Implementationen

Für Tuples stehen zwei Implementationen zur Verfügung:

- **array** basiert auf zusammenhängenden Arrays der Elemente, die auf dem Heap angelegt werden und daher beliebig groß werden können. Die Speicherallokation erfolgt standardmäßig in „Portionen" von Zweierpotenzen, um bei Änderungen der Element-Anzahl in kleinerem Maße nicht ständig die Array-Größe ändern zu müssen — dann wäre der Speicherverwaltungs-Aufwand zu groß. Der Zugriff auf einzelne Elemente erfolgt über eine einfache Adreßrechnung und damit *sehr* schnell. Allerdings muß bei Einfüge- und Löschoperationen in der Mitte des Tuple jeweils ein größerer Bereich umkopiert werden. Die Operationen für VDM-Strings sind nur in dieser Implementation vorhanden. Der Parameter „SIMPLEELEMS" signalisiert, daß die Elemente statisch sind, also keine Pointer auf dynamische Strukturen; der Parameter „ISCHAR" sagt, daß die Elemente Zeichen sind.
- **linklist** ist eine Standard-Implementation auf Basis einer verketteten Liste der Elemente, wie sie auch für andere VDM-Basistypen existiert. Zwar wird der benötigte Speicherplatz immer optimal ausgenutzt, für die Listenverwaltung ist allerdings zusätzlicher Speicherplatz für jedes Element nötig. Zudem ist für den Zugriff auf einzelne Elemente ein Durchgang durch die Liste erforderlich; je weiter hinten das Element ist und je länger das Tuple ist, desto größer ist die Zugriffszeit. Allerdings sind Einfüge- und Löschoperationen sehr schnell, da nur einige Pointer umgesetzt werden müssen.

5.5.3 Tuple-Operationen

Zur Veranschaulichung der Operationen dienen folgende Variablen:

```
Text    t1, t2;
Word    w1, w2;
Char    c1, c2;
```

Tuple-Konstruktoren

mk_empty Mit **mk_empty** erzeugen Sie ein leeres Tuple, das also keine Elemente enthält.
Beispiel — ein neuer, noch leerer Text wird erzeugt:

```
t1 = mk_empty_text ();
```

mk_one Mit **mk_one** erzeugen Sie ein neues Tuple, das bereits ein Element enthält.
Beispiel — Wörter müssen aus mindestens einem Zeichen bestehen, um in einen Text aufgenommen zu werden:

```
c1 = ´A´;
w1 = mk_one_word (c1);
```

mk2 Mit **mk2** können Sie mit einem Aufruf gleich ein zwei-elementiges Tuple erzeugen.
Beispiel — ein Text besteht zu Beginn mindestens aus einem Wort und einem Zwischenraum, hier in Form der Klammer hinter dem Buchstaben *A*:

```
c2 = ´)´;
w2 = mk_one_word (c2);
t1 = mk2_text (w1, w2);
```

$\Longrightarrow$ t1 enthält den Text „A)“

mk3 Mit **mk3** können Sie auch drei-elementige Tuples auf einmal initialisieren.

```
w1 = mk3_word (´V´,´D´,´M´);
```

mkn Mit **mkn** wird Ihnen die Aufgabe erleichtert, ein Tuple mit n-mal dem gleichen Element zu erzeugen.
Beispiel — Trennlinie aus 80 Bindestrichen:

```
w1 = mkn_word (´-´,80);
```

copy / copy1 Auch für Tuples gibt es natürlich diese beiden Operationen zum Kopieren eines anderen Tuple.

mkc / mklc VDM-Strings können Sie mit **mkc** und **mklc** aus vorhandenen *C*-Strings erzeugen, die wie üblich durch ihre Startadresse charakterisiert sind; mklc wandelt den String bei der Konvertierung in Kleinbuchstaben um. Näheres über diese beiden Operationen finden Sie im Abschnitt über String-Manipulation auf S. 77.

Tuple-Kombinatoren

Kombination mehrerer Tuples

conc Mit **conc** verketten Sie zwei Tuples miteinander (Konkatenation), d.h. das zweite wird an das erste angehängt.
Beispiel — der Hauptteil t1 und der Anhang t2 bilden zusammen *ein* Buch:

t1 = conc_text (t1,t2);

conc_path Mit **conc_path** erzeugen Sie einen Pfadnamen aus zwei Verzeichnisnamen, indem bei der Verkettung ein Pfad-Delimiter des jeweiligen Betriebssystems eingefügt wird. Dieser Pfad-Delimiter ist unter UNIX der *Slash* „/“, unter MS-DOS der *Backslash* „\“. Sie merken schon, es handelt sich um einen Befehl für Strings.
Beispiel — das Include-Verzeichnis mit den Standard-Header-Dateien von *C* konstruieren Sie aus den einzelnen Verzeichnisnamen wie folgt:

w1 enthält „/usr“ (Wurzel der Benutzer-Verzeichnisse)
w2 enthält „include“
w1 = conc_path_word (w1,w2);
⟹ w1 enthält jetzt “/usr/include“ !

Kombination eines Tuples mit einem Element

appl Mit **appl** hängen Sie am „linken“ Ende eines Tuple ein Element an („append left“); alle Elemente werden also um eine Position zum Ende des Tuple geschoben und das neue Element ist danach das erste Tuple-Element, also der „head“. Die Reihenfolge der Argumente ist übrigens entsprechend: erst das Element, dann das Tuple!
Beispiel — Sie stellen Ihrer Autobiographie das Wort „Erinnerungen“ als Titel voran:

w1 enthält „Erinnerungen“
t1 enthält Ihre Autobiographie
t1 = appl_text (w1,t1);
⟹ t1 enthält jetzt das komplette Werk!

appr Mit **appr** hängen Sie ein Element am „rechten“ Ende eines Tuple an („append right“), wodurch es zum neuen “last“ wird. Die Argument-Reihenfolge spiegelt diesen Vorgang wieder: erst das Tuple, dann das Element!
Beispiel — an einen Geschäftsbrief fügen Sie das Wort „Anlagen“ an:

w1 enthält „Anlagen“
t1 enthält Ihren Brief
t1 = appr_text (t1,w1);
⟹ t1 enthält jetzt den fertigen Brief!

ovwrt1 Mit **ovwrt1** ersetzen Sie das Element an einer bestimmten Position im

Tuple durch ein anderes, vorausgesetzt, das Tuple ist nicht kürzer. **owvrt1d** ist die zugehörige Version mit implizitem Löschen.
Beispiel — Ihre Autobiographie soll "Lebenserinnerungen" heißen, den alten Titel benötigen Sie nicht weiter:

w1 enthält „Lebenserinnerungen"
t1 enthält Ihre Autobiographie
t1 = ovwrt1d_text (t1,1,w1);
⟹ t1 enthält jetzt das berichtigte Werk !

Tuple-Prädikate

is_eq Mit **is_eq** testen Sie zwei Tuples auf Gleichheit; sie sind dann und nur dann gleich, wenn sie die gleiche Länge haben und die Elemente in der vorgegebenen Reihenfolge paarweise gleich sind (deren Gleichheit wird mit der entsprechenden Operation für den Elementtyp ermittelt).

is_ge Mit **is_ge** testen Sie, ob ein Tuple größer oder gleich einem anderen ist. Ein Tuple ist größer oder gleich, wenn keines seiner Elemente kleiner als das entsprechende Element des anderen Tuple ist und es bei Gleichheit aller Elementpaare nicht kürzer ist als das andere Tuple. Dazu muß für die Elemente natürlich eine entsprechende Relation existieren; der VDM-Compiler bemerkt einen etwaigen Fehler.

is_gr Mit **is_gr** schließlich testen Sie, ob ein Tuple größer als ein anderes ist. Es ist größer, wenn mindestens eines seiner Elemente größer ist als das entsprechende Element des anderen Tuple oder wenn es, bei Gleichheit aller Elementpaare, mehr Elemente hat.

Die folgende Tabelle veranschaulicht die Zusammenhänge mit Hilfe von VDM-Strings:

Wort 1	Wort 2	is_eq	is_ge	is_gr
Test	Test	J	J	N
Teile	Test	N	N	N
Test	Teile	N	J	J
Teile	Teil	N	J	J
Teil	Teile	N	N	N

is_empty Mit **is_empty** testen Sie, ob das Tuple gerade leer ist, also keine Elemente enthält und damit die Länge 0 hat.

Tuple-Attribute

len Mit **len** ermitteln Sie die „Länge" eines Tuple, also die Anzahl seiner Elemente.
Beispiel — die Länge eines VDM-Strings ist identisch mit der Länge des ent-

sprechenden C-Strings, denn dessen Endemarkierung wird von der Funktion strlen nicht mitgezählt!

w1 enthält „Suelldorf“
w2 enthält „!“
len_word (w1) == 9
len_text (mk2_text (w1,w2)) == 2

Die nächsten beiden Attribute haben mehrere Argumente, liefern aber dennoch eine Eigenschaft des Tuple!

search Mit **search** ermitteln Sie das am weitesten “links“ im Tuple stehende Element, das ein angegebenes Prädikat (in Form einer Funktion mit einem Bool-Resultat) erfüllt; gibt es kein solches Element im Tuple, ist das Ergebnis 0.
Beispiel — Suche nach dem ersten Großbuchstaben in einem Wort:

```
printf ("%d",search_word (w1,isupper));
```

index Mit **index** ermitteln Sie die erste Position eines bestimmten Elements im Tuple; ist das Element nicht enthalten, ist das Ergebnis 0.
Beispiel — Suche nach einem Bindestrich in einem Wort:

```
printf ("%d",index_word (w1,'-'));
```

Selektion in einem Tuple

Selektion eines Elements

hd Mit **hd** greifen Sie das erste Element, den *head*, aus einem Tuple heraus. Die Operation wird häufig zusammen mit tl zur kompletten Bearbeitung eines Tuple mit einer rekursiven Funktion verwendet; ein Beispiel finden Sie etwas weiter unten bei der Darstellung von tl.

last Mit **last** greifen Sie das letzte Element eines Tuple heraus, also das Element mit dem höchsten Index, vorausgesetzt, das Tuple ist nicht leer.
Beispiel — bevor Sie das Wort „Anlagen“ an Ihren Geschäftsbrief anfügen, möchten Sie prüfen, ob es bereits dort steht:

w1 enthält „Anlagen“
t1 enthält Ihren Brief

```
if (! is_eq_word (last_text (t1),w1))
   t1 = appr_text (t1,w1);
```

$\Longrightarrow$ t1 enthält den Brief, „Anlagen“ ist genau einmal vorhanden!

sel Mit **sel** greifen Sie ein beliebiges Element über dessen Index aus dem Tuple heraus — wenn es soviele Elemente enthält. sel ist damit eine allgemeine Operation, mit deren Hilfe hd und last realisiert werden können:

```
hd_xxx (obj) == sel_xxx (obj,1);
last_xxx (obj) == sel_xxx (obj,len_xxx (obj));
```

Selektion eines Teil-Tuple

tl Mit **tl** selektieren Sie den *tail* eines Tuple, entfernen also das erste Element, den *head*. Zusätzlich existiert die Version **tld** mit implizitem Löschen. Diese Operation wird häufig benutzt, um rekursiv ein komplettes Tuple abzuarbeiten, indem der *head* verarbeitet wird und die Funktion mit dem *tail* erneut aufgerufen wird.
Beispiel — Ermittlung der Zeichen-Summe (Summe der Zeichen-Codes) eines Wortes:

```
Nat0 charsum (Word w)
{
  if (is_empty_word (w))
      return 0;
  else
      return hd_word (w) + charsum (tl_word (w));
}
```

sub_tup Mit **sub_tup** selektieren Sie einen zusammenhängenden Teil des Tuple durch Angabe des unteren und oberen Index; ist der obere Index kleiner als der untere, wird das leere Tuple geliefert, ebenso, wenn der untere Index größer ist als die Länge des Tuple. Ist hingegen der obere Index größer als die Länge des Tuple, erhalten Sie das Ende des Tuple ab dem unteren Index.
Beispiel — Sie wollen den Anhang Ihrer Autobiographie herausgreifen, von dem Sie zufällig wissen, daß er beim 14965. Wort beginnt:

t1 enthält Ihr Buch
```
t2 = copy1_text (t1);
t2 = sub_tup_text (t2,14965,len_text (t2));
```
t1 enthält immer noch das gesamte Buch; die Operation sub_tup zerstört ihr Argument, daher wurde vorher mit copy1 eine Kopie des Buches erzeugt. Beide Texte verwenden aber die gleichen Wörter, da die Sub-Strukturen *nicht* mitkopiert wurden!

rmv1 Mit **rmv1** entfernen Sie ein Element aus einem Tuple, indem Sie dessen Position angeben. Sie erhalten also ein Tuple, in dem alle Elemente an höheren Positionen um eine Position nach „links“ geschoben wurden — vorausgesetzt, das Tuple enthielt ein Element an der gewünschten Position. Zusätzlich gibt es die Version **rmv1d** mit implizitem Löschen des entfernten Elements. rmv1 ist die allgemeine Form des Entfernens von Elementen aus einem Tuple; tl und sub_tup können damit realisiert werden:

```
tl_xxx (obj) == rmv1_xxx (obj,1)

Xxx sub_tup_xxx (Xxx obj,Nat1 lb,Nat0 ub)
{
  int i;
  for (i = 1; i < lb−1 && len_xxx (obj) > 0; i++)
        obj = rmv1_xxx (obj,1);
  ub − = lb−1;
  while (ub < len_xxx (obj))
        obj = rmv1_xxx (obj,ub+1);
  return obj;
}
```

Tuple: Datei-Operationen

Sie können mit Tuples sequentielle Datei-Operationen durchführen, denn ein Tuple kann als das Speicher-Abbild einer sequentiellen Datei (also eine Sequenz von Datensätzen) angesehen werden. Die üblichen Operationen sind das Einlesen der Datei, das Schreiben aller Datensätze im Tuple in eine Datei und das Anhängen der Datensätze im Tuple an eine bestehende Datei. Mit den drei Operationen fread, fwrite und fapp können Sie diese Aktionen durchführen, ohne sich um die Dateiverwaltung (Öffnen, Schließen usw.) ! → kümmern zu müssen. Allerdings wird angenommen, daß die Elemente des Tuple einfache, unstrukturierte Objekte sind.

Die Dateinamen werden übrigens bei allen drei Operationen in Form von VDM-Strings erwartet.

fread Mit **fread** lesen Sie die angegebene Datei *vollständig* in ein Tuple ein. Tritt in einer Datei-Operation ein Fehler auf, wird das leere Tuple geliefert. Beispiel — Sie können eine Textdatei in ein Tuple vom Typ word einlesen:

```
w2 enthält den Dateinamen „/usr/include/stdio.h“
w1 = fread_word (w2);
if (is_empty_word (w1))
  fileerror ();
fileerror ist eine selbstdefinierte Fehlerroutine !
```

fwrite Mit **fwrite** schreiben Sie ein Tuple vollständig in die angegebene Datei, deren vorheriger Inhalt (wenn sie schon existierte) gelöscht wird. Tritt bei einer Datei-Operation ein Fehler auf, wird *FALSE* geliefert.
Beispiel — Sie schreiben das soeben gelesene Tuple nach einigen Manipulationen wieder zurück:

```
w1 enthält die Datensätze
w2 enthält den Dateinamen „/usr/include/stdio.h“
if (! fwrite_word (w1,w2))
  fileerror ();
```

fapp Mit **fapp** hängen Sie ein Tuple an eine existierende Datei an (wenn sie

noch nicht existiert, wird sie erzeugt, die Wirkung entspricht dann fwrite). Tritt bei einer Datei-Operation ein Fehler auf, wird *FALSE* geliefert.
Beispiel — Sie haben einige zusätzliche Zeilen für Ihre persönliche *C*-Header-Datei in Form eines Wortes in einem Tuple vom Typ word gespeichert und hängen sie nun an diese Textdatei an:

w1 enthält die neuen Datensätze
w2 enthält den Dateinamen „/usr/mydir/my_decl.h"
```
if (!fapp_word (w1,w2))
   fileerror ();
```

Operationen für VDM-Strings

Wie schon kurz angedeutet, gibt es den vordefinierten Typ „Str" als Tuple von Zeichen, damit bei der Verwendung von VDM auch Zeichenketten bequem gehandhabt werden können; Sie haben bei den Datei-Operationen auch schon Einsatzmöglichkeiten kennengelernt. Außerdem waren alle bisherigen Operationen mit Hilfe von VDM-Strings illustriert. Es gibt auch einige spezielle VDM-String-Operationen, die allerdings *nur* in der Implementation „array" existieren, da eine Repräsentation von Strings nur in dieser Implementation sinnvoll ist. Diese Implementation hat zwei Parameter, die für VDM-Strings definiert sein sollten, nämlich „SIMPLEELEMS" (einfache Elemente ohne dynamische Speicheroperationen) und „ISCHAR", der anzeigt, daß es sich um Elemente vom Typ Char handelt.

Vorteil Der größte Vorteil von VDM-Strings: Da die Größe eines Tuple an die Anforderungen angepaßt wird, sind Fehler wegen Überschreitung des Speicherplatzes nicht möglich — bei *C*-Strings kommen sie gelegentlich vor und haben meist schwerwiegende Folgen (und sind zudem sehr schwer zu finden).

Die ersten drei Operationen sorgen für die Konvertierung zwischen VDM-Strings und normalen *C*-Strings.

cpc Mit **cpc** konvertieren Sie einen VDM-String in einen *C*-String, der wie üblich durch seine Anfangsadresse definiert wird. Sie müssen allerdings selbst dafür sorgen, daß der Speicherplatz für den String ausreicht. Die übliche Ende-Kennung für *C*-Strings, das Null-Zeichen, wird automatisch angefügt.
Beispiel — eine Konvertierungsfunktion, die den verfügbaren Speicher für den *C*-String als weiteres Argument erhält und bei zuwenig Speicher *FALSE* zurückliefert:

```
Bool to_c_str (Str s,char⋆ c,Nat0 l)
{
  if (len_str (s) >= l)
       return FALSE;
  else {
       cpc_str (s,c);
       return TRUE;
  }
}
```

Da zusätzlich eine Endekennung angehängt wird, darf die Länge des Strings nicht genauso groß sein wie der verfügbare Speicherplatz !

mkc Mit **mkc** konvertieren Sie einen *C*-String, den Sie mit seiner Anfangsadresse definieren, in einen VDM-String; das abschließende Null-Zeichen wird *nicht* mitkopiert.

```
char ⋆ c;
⋮
c = "Suelldorf !";
w1 = mkc_str (c);
```

mklc **mklc** macht dasselbe wie mkc, wandelt dabei aber alle Großbuchstaben in Kleinbuchstaben um.

Damit nicht alle weiteren Dinge, die man mit Strings machen kann, nur mit *C*-Strings möglich sind und eine Konvertierung nötig machen, bietet VDM eine Operation zur Ausgabe eines VDM-Strings in eine angegebene Datei; das kann natürlich auch *stdout* sein, so daß eine Bildschirmausgabe direkt möglich ist !

puts Mit **puts** schreiben Sie einen VDM-String zeichenweise mit Hilfe der *C*-Standardfunktion fputc in eine durch einen File-Pointer definierte *C*-Datei; die Hauptanwendung ist die Bildschirmausgabe in die Systemdateien stdout bzw. stderr. So ersparen Sie sich die vorherige Konvertierung in einen *C*-String.
Beispiel — Ausgabe einer Fehlermeldung, die im Tuple w1 steht:

```
puts_word (w1,stderr);
```

Tuple-Funktionale

Zur Erzeugung neuer, komplexer Funktionen mit Hilfe der vorhandenen gibt es drei **Funktionale**. Da Funktionale bereits bei den Sets ausführlich behandelt werden, ist die Beschreibung hier etwas kürzer; wenn Sie damit Schwierigkeiten haben, sollten Sie die entsprechenden Passagen ab S. 61 erneut lesen.

→ S. 61

binary Mit **binary** können binäre Funktionen die aus einem Tuple und einem Element ein neue Tuple erzeugen, verallgemeinert werden zu Funktionen, die aus zwei Tuples ein neues Tuple erzeugen und dabei ihre Argumente zerstören. Eine mit binary erzeugte Funktion arbeitet wie folgt:

```
Atup newfct_atup (Atup t1, Atup t2)
{
  Elem e;
  if (is_empty_atup (t2))
      return t1;
  else {
      e = hd_atup (t2);
      return newfct_atup (foldfct_atup (t1,e), tl_atup (t2,e)) ;
  }
}
```

foldfct hat zwei Argumente, ein Tuple und ein Element, und liefert wieder ein Tuple; Sie legen sie in der DSL-Spezifikation mit dem Parameter „FOLDFCT“ fest.

Ein Anwendungsbeispiel: die Operation conc kann mit Hilfe von binary und appr definiert werden.

```
IMPLEMENTATIONS
...+ binary.conc (FOLDFCT="appr")
```

fold Das Funktional **fold** hat drei Parameter-Funktionen und erzeugt eine Funktion, die ein Tuple auf einen durch die Parameter-Funktionen festgelegten Resultattyp abbildet. Damit lassen sich sehr gut komplexe Manipulationen *aller* Elemente eines Tuple realisieren. Die Parameter-Funktionen haben die gleiche Bedeutung wie bei Sets (hier natürlich auf Tuples bezogen). Die erzeugte Funktion sieht von der Struktur her wie folgt aus:

```
Retult newfct_atup (Atup t)
{
  Elem e;
  if (is_empty_atup (t))
      return nullfct_result ();
  else {
      e = hd_atup (t);
      return foldfct_result (cvfct_elem (e),
                             newfct_atup (tl_atup (t,e)));
  }
}
```

Nomenklatur Die Namen der drei Parameterfunktionen und der erzeugten Funktion werden erweitert: die Konvertierungsfunktion mit dem Namen des Elementtyps, die Nullfunktion mit dem Namen des Resultattyps, die Fold-Funktion mit dem Namen des Resultattyps und die erzeugte Funktion mit dem Namen des Tuple.

Beispiel: Eine häufige Anwendung von fold ist die distributive Konkatenation aller Tuples in einem Tuple von Tuples, also z.B. die Erzeugung eines Wortes, in dem alle Worte eines Textes aneinandergehängt sind:

```
IMPLEMENTATIONS
...+ fold.d_conc (NULLFCT="mk_empty", FOLDFCT="conc")
```

erzeugt

```
Word d_conc_text (Text t)
```

Weitere Informationen (z.B. über zusätzliche Parameter) und Beispiele über fold finden Sie im entsprechenden Abschnitt über fold für Sets bzw. Maps.

foldix **foldix** ist ein weiteres Funktional, das allerdings fold sehr ähnelt. Der einzige Unterschied betrifft die Konvertierungsfunktion eines Elements in den Resultattyp, die ein weiteres Argument erhält (an zweiter Stelle, direkt nach dem Element und noch vor den zusätzlich vereinbarten Argumenten): den *Index* des aktuellen Elements im Tuple. Damit ist eine index-abhängige Konvertierung der Elemente möglich.

Beispiel: Sie möchten die distributive Konkatenation (wie oben erläutert) nur für jedes zweite Wort des Textes durchführen.

```
IMPLEMENTATIONS

...+ foldix.d2_conc (CVFCT="second",
                     NULLFCT="mk_empty",
                     FOLDFCT="conc")
```

erzeugt

```
Word d2_conc_text (Text t)
```

Konvertierungsfunktion:

```
Word second_word (Word w, Nat0 i)
{ return (i & 1) ? mk_empty_word () : w ; }
```

Tuple: Diverse Operationen

forall Mit **forall** führen Sie für alle Elemente eines Tuple (in aufsteigender Reihenfolge) eine angegebene Prozedur aus, die ein Argument des Elementtyps erwartet.
Beispiel — die Ausgabe aller Worte eines Textes auf dem Bildschirm:

```
void disp_word (Word w)
{ puts_word (w, stdout); }
:
forall_text (t1, disp_word);
```

forindex Mit **forindex** führen Sie für alle Elemente eines Tuple (in aufsteigender Reihenfolge) eine angegebene Prozedur aus, die als Argumente ein Element

UND eine natürliche Zahl erwartet; diese Prozedur erhält zu jedem Element auch dessen Index im Tuple.
Beispiel — zeilenweise Ausgabe aller Worte eines Textes mit ihrer Ordnungszahl auf dem Bildschirm:

```
void disp2_word (Word w, Nat1 n)
{
  printf ("\n%d", n);
  puts_word (w, stdout);
}
:
forindex_text (t1, disp2_word);
```

forrange Mit **forrange** führen Sie für die Elemente eines Tuple, die innerhalb eines angegebenen Index-Bereichs liegen, eine Prozedur mit einem Argument vom Elementtyp aus.
Beispiel (nicht sehr sinnvoll) — Ausgabe eines bestimmten Ausschnitts eines Textes, dessen Grenzen vielleicht vom Benutzer eingegeben werden können:

```
void disp_word (Word w)
{ puts_word (w, stdout); }
:
forrange_text (t1, disp_word, firstword, lastword);
```

forsome Mit **forsome** führen Sie für einige Elemente eines Tuple, die ein angegebenes Prädikat erfüllen, eine Prozedur mit einem Argument vom Elementtyp aus.
Beispiel — Ausgabe aller Wörter eines Textes, die mit einem Großbuchstaben anfangen und mehr als vier Zeichen haben:

```
void disp_word (Word w)
{ puts_word (w, stdout); }
Bool main_word (Word w)
{ return isupper (hd_word (w)) && len_word (w) > 4; }
:
forsome_text (t1, main_word, disp_word);
```

5.5.4 Beispiel: „Pascal-Dreieck“

Wir wollen das berühmte Pascal-Dreieck bis zu einer beliebigen Ordnung berechnen. Zur Erinnerung: das Pascal-Dreieck stellt in der n. Zeile der Reihe nach folgende Werte dar (wobei die oberste Zeile als „nullte“ Zeile nur die eins enthält):

$$\binom{n}{0}\binom{n}{1}\cdots\binom{n}{n-1}\binom{n}{n}$$

Das Pascal-Dreieck beginnt also wie in Abb. 5.3 dargestellt.

							1							
						1		1						
					1		2		1					
				1		3		3		1				
			1		4		6		4		1			
		1		5		10		10		5		1		
	1		6		15		20		15		6		1	
1		7		21		35		35		21		7		1

Abbildung 5.3: Beispiel für Tuples: das Pascal-Dreieck

Wie wir dieser Abbildung entnehmen können, gibt es einen weiteren, einfacheren Algorithmus zur Erzeugung des Dreiecks: ein Element ergibt sich (von den Rand-Elementen einmal abgesehen, die immer 1 sind) aus der Summe der beiden Elemente, die diagonal rechts und links über ihm stehen! Diesen Algorithmus wollen wir nun umsetzen.

Zur Erzeugung der n.ten Zeile benötigen wir auch alle vorherigen Zeilen. Wir wollen uns auf die Fälle mit $n > 1$ beschränken, weil die ersten beiden Zeilen (Ordnungen 0 und 1) trivial sind.

Jede Zeile modellieren wir als Tuple von Zahlen, das ganze Dreieck als Tuple von Zeilen. Die DSL-Spezifikation muß also folgendermaßen aussehen:

```
DOMAINS
Tri  =  Row ⋆
Row  =  Nat1 ⋆
IMPLEMENTATIONS
Tri  =  array + ...
Row  =  array + ...
```

Zunächst entwerfen wir das Hauptprogramm, das die trivialen ersten beiden Zeilen im Dreieck abspeichert (und damit ein „wohlgeformtes Dreieck“ für die Berechnung größerer Dreiecke erzeugt) und danach, abhängig von der gewünschten Ordnung, die semantische Funktion mk_pascal aufruft.

```
void main (int argc,char ⋆ argv[])
{
  Tri tri;
  Row row;
  Nat0 n = atoi (argv[1]);
```

```
    row = mk_one_row (1);
    tri = mk_one_tri (row);
    if (n == 0) {
        pr_tri (tri);
        return;
    }
    row = mk2_row (1, 1);
    tri = appr_tri (tri, row);
    if (n == 1) {
        pr_tri (tri);
        return;
    }
    tri = mk_pascal (tri,n);
    pr_tri (tri);
}
```

Hauptprogramm

Zunächst wird die gewünschte Zeilenzahl aus der Kommandozeile gelesen; die erste Zeile wird mit dem Element 1 erzeugt (mit mk_one_row) und in das Dreieck übertragen (mit mk_one_tri). War die gewünschte Ordnung 0, so ist man bereits fertig. Sonst wird die nächste Zeile erzeugt (mit mk2_row) und an das Dreieck angehängt (mit appr_tri). War die gewünschte Ordnung 1, so ist man jetzt fertig. Für höhere Ordnungen wird die Funktion mk_pascal zur Erzeugung des gewünschten Dreiecks aufgerufen. Die Ausgabe des fertigen Dreiecks erfolgt in allen drei Fällen mit der Operation pr_tri, die natürlich nicht sehr schön ist; hier sollte man eine bessere Funktion schreiben.

! → Eine Falle bezüglich der Speicherverwaltung wurde hier umgangen: man könnte ja denken, daß es für die Erzeugung der Zeile mit der Ordnung 1 (also der zweiten Zeile) ausreicht, an die existierende Variable row eine weitere 1 anzuhängen — das wäre aber verhängnisvoll, da durch das Eintragen von row in tri ein Verweis auf den aktuellen Zustand von row entstanden ist. Hängen wir nun ein weiteres Element an, taucht es wie von Geisterhand auch in der ersten Zeile des Dreiecks auf, wo es fehl am Platze ist! Aus diesem Grunde müssen wir den bisherigen Inhalt von row aufgeben und die zweite Zeile neu aufbauen; dies hätten wir z.B. auch durch einen Aufruf von copy1_row und Anfügen des zweiten Elements mit appr_row erreichen können.

Es fehlt nun noch die semantische Funktion **mk_pascal**, die ein wohlgeformtes Pascal-Dreieck (mit der Ordnung 1) und die gewünschte Ordnung als Argumente erwartet und daraus das gewünschte Pascal-Dreieck erzeugt.

```
Tri mk_pascal (Tri tri, Nat1 n)
{
  Row row, last;
  int i, len;
  if (n < 2)
      return tri;
  else {
      row = mk_one_row (1);
      len = len_tri (tri);
      last = last_tri (tri);
      for (i = 1; i < len; i++)
          row = appr_row (row, sel_row (last,i) + sel_row (last,i+1));
      row = appr_row (row, 1);
      tri = appr_tri (tri, row);
      return mk_pascal (tri, n-1);
  }
}
```

mk_pascal Die Funktion prüft zunächst, ob das Dreieck bereits fertig ist — dazu muß die Ordnung kleiner als zwei sein. Wenn ja, wird das bisher erstellte Dreieck zurückgeliefert. Im anderen Fall wird nun das erste Element der neuen Zeile (das ja immer 1 ist) eingetragen (mit mk_one_row). Die Zeilenzahl des bisherigen Dreiecks (also seine Ordnung plus eins) wird ermittelt (mit len_tri), um die Anzahl der Additionen, die für die neue Zeile erforderlich sind, zu bestimmen. Schließlich wird die Zeile, aus der die Elemente für die Addition entnommen werden (die bisher letzte Zeile des Dreiecks) mit last_tri herausgegriffen.

In der for-Schleife werden für die Zeile der Ordnung n, die ja $n+1$ Elemente enthält, $n-1$ Additionen durchgeführt, denn die beiden äußeren Elemente sind immer gleich 1. Das i.te Element der neuen Zeile (das wegen des bereits vorhandenen ersten Elements innerhalb der Zeile die Position $i+1$ einnimmt) ergibt sich als Summe der Elemente i und $i+1$ der vorigen Zeile, die jeweils mit sel_row ermittelt werden. Nach Verlassen der for-Schleife wird das letzte Element der Zeile, die rechte 1, angehängt und die fertige Zeile in das Dreieck eingetragen; danach wird mk_pascal mit um 1 verringerter Ordnung rekursiv aufgerufen.

5.6 Maps

5.6.1 Was ist eine Map ?

Definition Eine **Map** ist eine partielle Funktion, die Objekten aus einer Argumentmenge andere Objekte aus einer Zielmenge zuordnet. Die Argumentmenge ist endlich; ihre Elemente sind ungeordnet, so daß die Paare nicht in einer Reihenfolge stehen.

Nomenklatur

Die Argumentmenge heißt auch *Argument-Domain*. Die Zielmenge wird auch *Co-Domain* genannt. Die Elemente der Argumentmenge heißen *Argumentwerte*, die ihnen zugeordneten Werte aus der Zielmenge heißen *Map-Werte*.

Häufig kann keine allgemeine Regel zur Erzeugung der Map-Werte angegeben werden. Das entscheidende bei einer Map ist allerdings folgendes: die Paare sind durch die Argumentwerte *eindeutig* definiert, denn kein Argumentwert kann mehr als einen Map-Wert haben! Also dient der Argumentwert als „Schlüssel" zum Map-Wert; um auf ein Paar zuzugreifen, benötigt man nur Kenntnisse über den Argumentwert.

Zusammenfassung

- Die Argumentwerte sind einzigartig, sie kommen nur einmal in der Argumentmenge vor.
- Die Argumentwerte gehören dem gleichen Datentyp an.
- Die Map-Werte gehören ebenfalls dem gleichen Datentyp an.
- Argumentmengentyp und Zielmengentyp stimmen nicht notwendigerweise überein; in der Regel sind sie verschieden.
- Die Argumentwerte sind nicht geordnet, obwohl sie natürlich einem Ordnungsprinzip unterliegen können (wie z.B. die natürlichen Zahlen).
- Die Anzahl der Argumentwerte ist endlich.
- Durch den Argumentwert ist das Paar eindeutig bestimmt. Er kann als Schlüssel zu seinem Map-Wert dienen.

Maps und Sets

Aus diesen Angaben kann man ersehen, daß die Argumentmenge ein Set im Sinne des Abschnitts 5.4 ist, auf den die Set-Operationen angewendet werden können. Für Maps gibt es dementsprechend viele ähnliche Operationen.

DSL

Um in einer DSL-Spezifikation eine Map zu definieren, die Objekte vom Typ $\mathcal{A}$ auf Objekte vom Typ $\mathcal{B}$ abbildet, schreiben Sie

$$Amap \quad = \quad \mathcal{A} \quad -m-> \quad \mathcal{B}$$

Dies ist die „maschinenlesbare" Schreibweise von $\mathcal{A} \overrightarrow{m} \mathcal{B}$

Anwendung

Die Map ist der vielseitigste VDM-Basistyp. Ihr Einsatz ist immer dann angezeigt, wenn Sie eine ungeordnete Sammlung verschiedener, zusammengesetzter Objekte modellieren sollen, die möglichst schon durch einen ihrer Teile genau bestimmt sind und auf die Operationen wie Suche nach Objekten, oder Hinzufügen, Löschen und Ersetzen von Objekten angewendet werden sollen. Die letztgenannten Operationen geben Ihnen vielleicht einen Hinweis: die

Manipulation von indizierten Datensätzen ist mit Hilfe von Maps gut durchzuführen — der Index (Primärschlüssel) kann als Argumentwert dienen.

Beispiele

Sie können auch ein Tuple mit einer Map darstellen, wenn Sie auf die Reihenfolge der Indizes verzichten; der Typ der Argumentwerte ist dann der Zahlentyp Nat1 . Sie können aber, wenn Sie auch negative Indizes benötigen, ebensogut den Typ Intg verwenden — Sie sind also flexibler, was den Typ der Indizes angeht. Die Implementation „scaldom“ unterstützt übrigens genau diese Anwendung von Maps. Häufig spricht aber die Ordnung der Elemente und — vor allem — der schnellere Zugriff auf die Elemente für die Wahl des Tuple statt der Map.

DOMAINS
Elem $=$ ...
Amap $=$ Nat0 $\overrightarrow{m}$ Elem
Atup $=$ Elem $\star$

Eine weitere Anwendung ist die Zuordnung von Dateinamen des Datei-Systems zu ihren Inhalten. Wenn es sich um sequentielle Dateien mit Daten des gleichen, elementaren Typs handelt, kommt als Zielmenge idealerweise ein Tuple in Betracht. Auch diese Anwendung wird durch eine Implementation besonders unterstützt, nämlich durch „dir“.

DOMAINS
Elem $=$ ...
Atup $=$ Elem $\star$
Amap $=$ Str $\overrightarrow{m}$ Atup

Wenn Sie in C mit dynamischen Objekten umgehen, vereinbaren Sie für den Zugriff einige statische Pointer. Sie benötigen in der Regel keine Operationen für die Bearbeitung der Gesamtheit dieser Pointer, sondern nur den Zugriff auf das Heap-Objekt sowie Erzeugen und Löschen solcher Objekte. Die Adresse eines Objekts auf dem Heap ist dabei ein Charakteristikum dieses Objekts; Sie können also auch Pointer und Objekte zu einer Map zusammenfassen und die Objekte mit Hilfe der Map-Operationen manipulieren. Wenn Probleme und Fehler beim Umgang mit diesen Objekten auftauchen, können Sie die Implementation der Map einfach austauschen — Sie haben dann zwar mehr Verwaltungsaufwand, aber auch eine „sichere“ Basis für die Manipulation der Heap-Objekte. Im Abschnitt 5.6.5 wird diese Anwendung ausführlicher beschrieben.

Nach diesen sehr speziellen Anwendungen nun noch eine „gewöhnliche“, die im weiteren auch zur Erläuterung der Operationen eingesetzt wird.

Ein Datei-System enthält Verzeichnisse, die Dateien enthalten. Jede Datei ist durch ihren Namen eindeutig gekennzeichnet. Ein Verzeichnis kann also als Map von Dateinamen auf Dateien modelliert werden, während ein Datei-

System eine Map von Verzeichnisnamen auf Verzeichnisse ist. Die Dateien sind einfach Tuples von irgendwelchen, nicht näher definierten Datensätzen.

DOMAINS
Filesys = Str $\overrightarrow{m}$ Dir
Dir = Str $\overrightarrow{m}$ File
File = Rec $\star$

5.6.2 Map-Implementationen

Maps können durch fünf Implementationen repräsentiert werden:

- **linklist** ist die allgemeinste Implementation, sie ist für alle Kombinationen von Argument-Domains und Co-Domains verfügbar. Naturgemäß ist sie wegen ihrer Universalität nicht sehr schnell.
- **scaldom** ist einsetzbar, wenn der Argument-Domain ein einfacher Datentyp mit fester Untergrenze ist; alle Zahlentypen und auch der Zeichentyp Char erfüllen dieses Kriterium, bei Intg allerdings sollte man die Untergrenze auf jeden Fall an die in der Praxis vorkommenden Werte anpassen. Die Modellierung nicht vorhandener Paare erfolgt durch die Einführung eines speziellen Wertes **nil** für diesen Zustand, also ist der Co-Domain ein *Optional* des Zielmengentyps[1].
- **dir** dient wie erwähnt zur Modellierung der Map-Werte als Dateien von Tuples, die sequentielle Datensätze enthalten können oder als Unterverzeichnisse, die gleichartige Maps sind.
- **heap** stellt die schon erwähnte Möglichkeit bereit, praktisch direkt auf den Heap-Speicher zuzugreifen. Durch die fehlende Aggregierung der Argumentwerte (sie sind auf einzelne Pointer ohne weitere zusammenfassende Informationen verteilt) sind alle Operationen, die mit der gesamten Map umgehen, nicht realisierbar; die vorhandenen Operationen schaffen aber eine geeignete Basis für die Manipulation des Heap. Sie sollten „heap" allerdings nicht während der Testphase Ihrer Applikation verwenden, da diese Operationen einige Sicherheitsprüfungen *nicht* durchführen! Testen Sie also z.B. mit „heapmap" und wechseln Sie erst nach erfolgreichem Test auf „heap", wenn die Performance-Anforderungen das *wirklich* nötig machen. Die folgenden Operationen sind vorhanden:
  mk_empty, ovwrt1, union1, sel, sub1, ext
- **heapmap** ist die „sichere" Variante der Implementation „heap" — die Argumentwerte und Map-Werte sind genau wie bei „heap" Pointer und

[1]Optionals werden im nächsten Abschnitt erläutert; hier sei gesagt, daß sie außer allen Werten des Elementtyps noch einen zusätzlichen, speziellen Wert nil bereitstellen.

Heap-Objekte, allerdings werden die Argumentwerte in einem Set gehalten, so daß unter „heapmap“ *alle* Operationen verfügbar sind. Natürlich ist sie dadurch etwas langsamer als die „reine“ Heap-Version, macht dies allerdings durch erhöhte Sicherheit und wesentlich mehr Komfort wieder wett.

5.6.3 Map-Operationen

Vorbemerkung Die Operationen für Sets und Tuples sind entweder Prozeduren oder „echte“ Funktionen, die genau ein Resultat haben. Für Maps hat es sich als praktikabel erwiesen, Operationen mit mehreren Resultaten bereitzustellen; dazu erhalten diese Funktionen Referenzparameter, die (wie in *C* üblich) in Form ihrer Adresse übergeben werden.

„heap“ Die Implementation „heap“ stellt einen Sonderfall einer Map dar, daher sind nur sehr wenige Operationen verfügbar, auf die im folgenden jeweils explizit hingewiesen wird. Fehlt dieser Hinweis also, gibt es die Operation unter „heap“ nicht.

Zur Erläuterung der Operationen dienen folgende Variablen:

```
Filesys  fs;
Dir      d1, d2, d3, d4;
File     f1, f2;
Str      fname, dname;
```

Map-Konstruktoren

mk_empty Mit **mk_empty** erzeugen Sie eine leere Map. mk_empty ist in der Implementation „heap“ verfügbar.
Beispiel — Einrichtung eines Datei-Systems auf einem neuen Rechner:

```
fs = mk_empty_filesys ();
```

mk_one Mi **mk_one** erzeugen Sie eine ein-elementige Map. Natürlich müssen Sie hier *zwei* Argumente angeben, die die beiden Teile des ersten Paares in der Map angeben.
Beispiel — Erzeugung eines Verzeichnisses mit einer Datei:

```
d1 = mk_one_dir (fname, f1);
```

copy / copy1 Natürlich gibt es auch für Maps die beiden Kopier-Operationen **copy** und **copy1**.

Map-Kombinatoren

Die Kombinatoren sind zum größten Teil von den Sets her geläufig, denn sie beziehen sich auf den Argument-Domain einer Map; die Map-Werte werden dabei meist nur als „Zubehör“ der Argumentwerte betrachtet.

Die einzige Ausnahme ext ist daher auch nur in den Implementationen „heap" und „heapmap" verfügbar, die ja, was ihre Argumentwerte angeht, einen Sonderstatus einnehmen: es sind die Pointer auf Daten auf dem Heap.

ext Mit **ext** tragen Sie einen neuen Map-Wert in die Map ein, die hier Referenzparameter ist. Der Argumentwert, dem der neue Map-Wert zugeordnet wird, also ein Pointer auf den Map-Wert, wird als Resultat geliefert. Beispiel:

```
DOMAINS
Token = ELEMENTARY
Map = Token  -m->  Str
IMPLEMENTATIONS
Token = ptr + ...
Map = heap + mk_empty + ext + ...

Token t;
Map m;
Str s;
⋮
m = mk_empty_map ();
t = ext_map (& m,s);
```

Der String wird an freier Position auf dem Heap abgespeichert, seine Adresse wird im *Token* abgelegt. Die Operation entspricht also einer normalen Speicherallokation.

Die übrigen Kombinatoren sind für alle Implementationen verfügbar:

union Mit **union** fügen Sie zwei Maps zusammen, vereinigen also ihre Argumentmengen. Eine Vorbedingung muß allerdings erfüllt sein: den Argumentwerten, die in beiden Maps auftreten, muß jeweils der gleiche Map-Wert zugeordnet sein ! Hinzu kommt die Version **uniond** mit implizitem Löschen. Beispiel — Sie fügen zwei Verzeichnisse zu einem zusammen; enthalten sie gleiche Dateien (Name und Inhalt), wird eine der beiden gelöscht:

```
d1 = uniond_dir (d1,d2);
```

union1 Mit **union1** tragen Sie in eine Map ein neues Paar ein, das also noch *nicht* enthalten ist. Erlaubt ist allerdings auch, daß die Map das Paar bereits enthält; in diesem Falle können Sie eines der beiden identischen Paare mit **union1d** implizit löschen. Ein Fehler ist aber das Hinzufügen eines Paares, dessen Argumentwert mit einem anderen Map-Wert in der Map enthalten ist ! union1 ist in der Implementation „heap" verfügbar.
Beispiel — ein neues Verzeichnis wird ins Datei-System eingetragen:

```
fs = union1_filesys (fs,dname,d1);
```

ovwrt Mit **ovwrt** fügen Sie ebenfalls zwei Maps zu einer zusammen, allerdings

braucht die Vorbedingung von union hier nicht erfüllt zu sein: ein Argumentwert darf in beiden Maps vorkommen und jeweils einen unterschiedlichen Map-Wert haben! Die zweite angegebene Map ist dabei dominant. Die Version mit implizitem Löschen heißt **ovwrtd**.
Beispiel — beim Zusammenfügen zweier Verzeichnisse sollen die Dateien des zweiten Vorrang genießen:

d1 = ovwrtd_dir (d1,d2);

ovwrt1 Mit **ovwrt1** überschreiben Sie ein Paar in einer Map durch ein neues (ist es noch nicht vorhanden, wird es ganz normal eingefügt). **ovwrt1d** ist die Version mit implizitem Löschen. ovwrt1 ist in der Implementation „heap" verfügbar.
Beispiel — Sie wollen eine Datei nach einer Änderung wieder zurück ins Verzeichnis schreiben; die alte Version der Datei soll dabei gelöscht werden:

d1 = ovwrt1d_dir (d1,fname,f1);

sub Mit **sub** entfernen Sie alle Paare, deren Argumente in einer Map vorhanden sind, aus einer anderen Map (unabhängig vom Map-Wert). **subd** löscht dabei implizit.
Beispiel — Sie haben zwei Verzeichnisse, die zum Teil identische Dateien enthalten. In einem dieser Verzeichnisse wollen Sie die doppelten Dateien löschen:

d1 = subd_dir (d1,d2);

sub1 Mit **sub1** entfernen Sie das Paar, dessen Argumentwert Sie angeben, aus einer Map (wenn es enthalten ist). **sub1d** löscht dabei das Argument und den Map-Wert implizit. sub1 ist in der Implementation „heap" verfügbar.
Beispiel — Sie wollen ein ganzes Verzeichnis aus Ihrem Datei-System löschen:

fs = sub1d_filesys (fs,dname);

inters Mit **inters** bilden Sie den Durchschnitt zweier Maps, also genau die Paare, die in beiden Maps enthalten sind — die Gleichheit betrifft sowohl den Argumentwert als auch den Map-Wert! **intersd** löscht implizit die Paare, die nicht im Ergebnis vorkommen.
Beispiel — Sie möchten die vollkommen identischen Dateien zweier Verzeichnisse herausfinden, nicht aber die Datei-Paare, die den gleichen Namen haben, aber unterschiedlichen Inhalt; die gemeinsamen Dateien sollen in ein neues Verzeichnis eingetragen werden, die alten Verzeichnisse sollen unverändert bleiben:

```
d3 = copy_dir (d1);
d4 = copy_dir (d2);
d3 = intersd_dir (d3, d4);
```

d4 dient nur als temporärer Speicher und wird von intersd vollständig wieder gelöscht.

Map-Prädikate

is_eq Mit **is_eq** ermitteln Sie, ob zwei Maps vollkommen identisch sind, also die gleiche Anzahl Argumentwerte haben, die jeweils in beiden Maps identisch sind und denen jeweils identische Map-Werte zugeordnet sind.
Beispiel — Test, ob zwei Verzeichnisse identisch sind; wenn ja, kann das eine komplett gelöscht werden:

```
if (is_eq_dir (d1,d2))
  del_dir (d2);
```

Es wird angenommen, daß die Dateien jeweils doppelt existieren, daß die Verzeichnisse also nicht nur Verweise auf Dateien enthalten; sonst wäre das Löschen aller Sub-Strukturen fatal!

is_empty Mit **is_empty** ermitteln Sie, ob eine Map keine Paare enthält.
Beispiel — Test, ob ein einem Verzeichnis Dateien enthalten sind; wenn nicht, kann das Verzeichnis gelöscht werden:

```
if (is_empty_dir (d1))
  del1_dir (d1);
```

Das Löschen mit del1 genügt hier, da es keine Sub-Strukturen gibt, die ebenfalls gelöscht werden können.

is_in Mit **is_in** ermitteln Sie, ob ein Argumentwert in einer Map enthalten ist.
Beispiel — Sie prüfen, ob ein Dateiname in einem bestimmten Verzeichnis enthalten ist; wenn nicht, tragen Sie eine Datei mit diesem Namen ein:

```
if (!is_in_dir (d1,fname))
  d1 = union1_dir (d1,fname,f1);
```

is_submap Mit **is_submap** prüfen Sie, ob eine Map vollständig in einer anderen Map enthalten ist, das heißt, ob alle ihre Paare ein identisches Pendant in der anderen Map haben.
Beispiel — wenn alle Dateien eines Verzeichnisses identisch in einem anderen Verzeichnis vorkommen, kann das Verzeichnis gelöscht werden:

```
if (is_submap_dir (d1,d2))
  del_dir (d1);
```

Es gibt noch vier weitere Prädikate, die aber gleichzeitig ein Objekt selektieren, wenn das Prädikat erfüllt ist. Aus diesem Grund sind sie unter den Selektions-Operationen aufgeführt. Es handelt sich um is_in_dom, is_in_rng, is_in_min und is_in_max.

Map-Attribute

card **card** liefert die Anzahl an Paaren in einer Map.
Beispiel — Anzahl der Dateien in einem Verzeichnis:

```
printf ("%d", card_dir (d1));
```

Map: Selektion

sel — Mit **sel** ermitteln Sie den Map-Wert, der dem angegebenen Argumentwert zugeordnet ist; eine übliche Schreibweise dieser „Anwendung der Map" (*map apply*) ist auch $val = map(arg)$. sel ist in der Implementation „heap" verfügbar.
Beispiel — Sie wollen die Datei mit dem Namen rec.dat verarbeiten:

```
fname enthält "rec.dat"
f1 = sel_dir (d1, fname);
```

sel_dom — Mit **sel_dom** erhalten Sie irgendeinen zufälligen Argumentwert der Map (die Wirkung entspricht also der sel-Operation für Sets).
Beispiel (nicht sehr sinnvoll) — Sie benötigen irgendeinen Dateinamen, der in einem Verzeichnis vorkommt:

```
fname = sel_dom_dir (d1);
```

Selektion mit Prädikaten — Die folgenden Operationen sind Kombinationen von Prädikaten und Selektions-Operationen, denn sie liefern ein Objekt (immer als Referenzparameter), wenn das Prädikat erfüllt ist; ist es nicht erfüllt, ist der Referenzparameter undefiniert. Das Resultat ist jeweils ein Wahrheitswert.

is_in_dom — **is_in_dom** prüft (wie is_in), ob ein bestimmter Argumentwert in einer Map enthalten ist; wenn ja, wird auch der zugeordnete Map-Wert geliefert.
Beispiel — wenn die Datei rec.dat in einem Verzeichnis existiert, soll sie verarbeitet werden:

```
fname enthält "rec.dat"
if (is_in_dom_dir (fname, d1, &f1))
    process_file (f1);
else
    perror ("rec.dat nicht vorhanden");
```

is_in_rng — Mit **is_in_rng** ermitteln Sie, ob ein Map-Wert in einer Map enthalten ist; wenn ja, erhalten Sie den zugehörigen Argumentwert. Ist der Map-Wert mehrfach vorhanden, wird der Argumentwert zufällig ausgewählt.
Beispiel — Sie möchten wissen, ob und wenn ja unter welchem Dateinamen eine bestimmte Datei in einem Verzeichnis enthalten ist:

```
if (is_in_rng_dir (f1,d1,& fname))
  puts_str (fname);
```

is_in_min — Mit **is_in_min** ermitteln Sie den kleinsten Argumentwert in einer Map, der größer oder gleich dem angegebenen ist (wenn es überhaupt einen solchen gibt).
Diese Operation kann z.B. angewendet werden, wenn Sie eine indizierte Datei mit einer Map von Key auf Datensatz modelliert haben und einen Teilschlüssel suchen:

DOMAINS
File = Str $\xrightarrow{m}$ Record
⋮
key ist ein Teilschlüssel und enthält „Schmi"
if (is_in_min_file (key, f, & fullkey))
    process_rec (sel_file (f, fullkey);

Der erste Schlüssel, der größer oder gleich „Schmi" ist (also z.B. „Schmid"), wird gesucht und der zugehörige Datensatz verarbeitet.

is_in_max **is_in_max** sucht den größten Argumentwert in einer Map, der kleiner oder gleich dem angegebenen ist (wenn es einen solchen gibt).
is_in_max ist also der Partner von is_in_min, der z.B. für die Teilschlüsselsuche in einer absteigend sortierten indizierten Datei oder vom Dateiende her verwendet werden kann.

Map: Dateiverwaltung

Es gibt für Maps (wie auch für Tuples) Operationen zum Zugriff auf das Datei-System, allerdings nur Spezialfälle, die dementsprechend deutlichen Einschränkungen unterworfen sind:

- Die Argumente müssen VDM-Strings sein; sie repräsentieren Verzeichnis- bzw. Dateinamen.
- Die Map-Werte müssen Objekte sein, für die die Operationen fread und fwrite definiert sind; mit Maps als Map-Werten werden Unterverzeichnisse repräsentiert, mit Tuples Dateien (siehe dazu die Datei-Operationen für Tuples).

Die Implementationen „heap", „heapmap" und „scaldom" scheiden also bereits aus, da ihre Argumente keine VDM-Strings sein können.

Anwendung Zur Verwaltung einfacher Tuples reicht die „Standardversion" aus:

DOMAINS
Dir = Str $\xrightarrow{m}$ File
File = Rec $\star$

Eleganter ist die rekursive Definition des Co-Domain, entweder als Datei oder als weiteres Verzeichnis; damit kann ein hierarchisches Datei-System wie unter *UNIX* oder *MS–DOS* verwaltet werden. Damit ist eine einfache indizierte Dateiverwaltung möglich, die für jeden Datensatz eine Datei anlegt: sie arbeitet mit Schlüsseln, die in Verzeichnis- und Dateinamen zerlegt werden. Eine beliebig tiefe Strukturierung der Schlüssel ist möglich — der einzige Nachteil ist, daß jeder Datensatz eine Datei bildet und sehr viel Festplattenplatz vergeudet (Verwaltungsinformation für die Verzeichnisse und Datei-Speicherplatz, der in der Regel blockweise vergeben wird). Die Definition benutzt *Unions*, die im Abschnitt 5.9 genauer behandelt werden:

DOMAINS
Direc = Str $\xrightarrow{m}$ Direntry
Direntry = Datfile | Direc

„dir“ Die Implementation „dir“ kann diese beiden Map-Strukturen direkt auf der Festplatte verwalten. Nur bei Bedarf wird ein Teil des Datei-Systems in den Hauptspeicher geladen, wo die „linklist“-Implementation verwendet wird (momentan die einzige Möglichkeit...). Wenn Sie also sehr viele Objekte in Ihrer Applikation verwalten, sollten Sie überlegen, die Implementation „dir“ zu verwenden, um Hauptspeicherplatz zu sparen.

Haben Sie eine solche Map vereinbart, stehen die folgenden beiden Operationen zur Verfügung; beide haben einen VDM-String als Argument, der den Verzeichnisnamen festlegt.

fread Mit **fread** wird ein durch einen VDM-String bezeichnetes Verzeichnis des Datei-Systems in eine Map eingelesen. Unterverzeichnisse (wenn sie vorhanden sind) bilden untergeordnete Maps, Dateien werden in Tuples gelesen. Tritt bei einer Datei-Operation ein Fehler auf, wird die leere Map zurückgeliefert.

```
dirvar = fread_direc (dname);
if (is_empty_direc (dirvar))
    file_error ();
```

fwrite Mit **fwrite** wird umgekehrt eine Map in entsprechende Verzeichnisse bzw. Dateien des Datei-Systems geschrieben. Tritt bei einer Datei-Operation ein Fehler auf, wird *FALSE* zurückgeliefert, sonst *TRUE*.

```
if (!fwrite_direc (dirvar, dname))
    file_error ();
```

Map: Funktionale

Zur Erzeugung neuer, komplexer Funktionen mit Hilfe der vorhandenen gibt es zwei **Funktionale**, die Sie mittlerweile von den Sets und Tuples her kennen. Wie schon gesagt: wenn Sie mit der folgenden Beschreibung Schwierigkeiten haben, sollten Sie die entsprechenden Passagen ab S. 61 erneut lesen.

→ S. 61

binary Mit **binary** können binäre Funktionen die eine Map und eine Paar — Argumentwert und Map-Wert — in eine Map überführen, verallgemeinert werden zu Funktionen, die aus zwei Maps eine neue Map erzeugen. Eine mit binary erzeugte Funktion arbeitet wie folgt:

```
Amap newfct_amap (Amap m1,Amap m2)
{
  Argdom a;
  Codom c;
  if (is_empty_amap (m2))
      return m1;
  else {
      a = sel_dom_amap (m2);
      c = sel_amap (m2, a);
      return newfct_amap (foldfct_amap (m1, a, c),
                          sub1_amap (m2,a));
  }
}
```

foldfct hat bei Maps *drei* Argumente, eine Map, einen Argumentwert und einen Map-Wert, und liefert wieder eine Map. Sie wird mit dem Implementationsparameter „FOLDFCT" definiert.

fold Das Funktional **fold** hat drei Parameter-Funktionen und erzeugt eine Funktion, die eine Map auf einen durch die Parameter-Funktionen festgelegten Resultattyp abbildet. Damit lassen sich sehr gut komplexe Manipulationen *aller* Paare einer Map realisieren. Die Bedeutung der Parameter-Funktionen ist Ihnen ja bereits bekannt (Konvertierungsfunktion, Null-Funktion und Fold-Funktion).

CVFCT Bei Maps ist die Konvertierungsfunktion allerdings *nicht* optional, da sie ein Paar der Map in *einen* Wert vom Resultattyp umwandelt. Ihr Name wird sowohl mit dem Namen des Argument-Domains als auch mit dem Namen des Co-Domains erweitert.

Die erzeugte Funktion sieht von der Struktur her wie folgt aus:

```
Result newfct_amap (Amap m)
{
  Argdom a;
  Codom c;
  if (is_empty_amap (m))
      return nullfct_result ();
  else {
      a = sel_dom_amap (m);
      c = sel_amap (m, a);
      return foldfct_result (cvfct_argdom_codom (a, c),
                          newfct_amap (sub1_amap (m, a)));
  }
}
```

Beispiel: Zwei Standard-Operationen für Maps werden von VDM a priori nicht unterstützt, da der Ergebnistyp nicht in jeder Applikation spezifiziert ist. Mit Hilfe von fold können Sie sie aber leicht generieren, wenn Sie die entsprechenden Datentypen definiert haben. Es handelt sich um den Set aller Argumente (der mit der Standard-Operation **dom** ermittelt wird) und den Set aller Map-Werte (Ergebnis der Standard-Operation **rng**). Folgende Domain-Gleichungen sind notwendig:

```
DOMAINS
Amap   =  Arg  -m->  Rng
Argset =  Arg -set
Rngset =  Rng -set
```

dom Die Operation **dom** ist die Vereinigung (union) aller einelementigen Sets, die aus der Argumentmenge der Map gebildet werden können. Die Nullfunktion erzeugt also den leeren Argset, die Konvertierungsfunktion wandelt ein Paar in einen ein-elementigen Argset um. Diese letzte Funktion müssen Sie selbst schreiben; das ist aber ganz leicht — nur die Ermittlung des Funktionsnamens erfordert kurzes Nachdenken:

```
IMPLEMENTATIONS
Amap  = ...
          + fold.dom (RESULT="Argset",
                      CVFCT="toargset",
                      NULLFCT="mk_empty",
                      FOLDFCT="union")

Argset toargset_arg_rng (Arg a, Rng r)
{ return mk_one_argset (a); }
```

rng Die Operation **rng** wird nach demselben Muster definiert, nur der Resultattyp und die Konvertierungsfunktion sind zu ändern:

```
IMPLEMENTATIONS
Amap  = ...
          + fold.rng (RESULT="Rngset",
                      CVFCT="torngset",
                      NULLFCT="mk_empty",
                      FOLDFCT="union")

Rngset torngset_arg_rng (Arg a, Rng r)
{ return mk_one_rngset (r); }
```

! → Beachten Sie: Hier werden natürlich *andere* Versionen von mk_empty und union ausgewählt als bei der Funktion dom .

Zusätzliche Parameter

→ 98ff

Eine einfache Anwendung für zusätzliche Parameter für eine der Parameterfunktionen finden Sie im abschließenden Beispiel über eine Stücklistenverwaltung mit Maps (S. 98ff). Sollten Sie Probleme beim Verständnis des zusätzlichen Implementations-Parameters haben, schauen Sie doch noch einmal auf S. 63 nach.

Map: Diverse Operationen

forall

Mit **forall** führen Sie mit allen Paaren in einer Map eine Prozedur aus, die zwei Elemente hat: eines aus dem Argument-Domain und eines aus dem Co-Domain der Map.
Beispiel — Ausgabe eines Hexdumps für alle Dateien in einem Verzeichnis:

```
void hexdump (Str name, File f)
{ puts_str (name);   dump_file (f); }
forall_dir (d1, hexdump);
```

forrange

Mit **forrange** führen Sie mit allen Paaren einer Map, deren Argumentwert in einem angegebenen Bereich liegt, eine Prozedur aus, die zwei Elemente hat: eines aus dem Argument-Domain und eines aus dem Co-Domain der Map. Voraussetzung ist, daß für den Argument-Domain die Operation is_ge definiert ist; dies ist z.B. bei Sets nicht gegeben.
Beispiel — Hexdump aller mit „F“ beginnenden Dateien in einem Verzeichnis:

```
Str f1, f2;
f1 = mk_one_str (´F´);
f2 = mk2_str (´F´,´~´);
forrange_dir (d1, hexdump, f1, f2);
```

Hinweis: alle mit „F“ beginnenden Dateinamen sind kleiner als „F~“.

forsome

Mit **forsome** führen Sie mit allen Paaren einer Map, die ein angegebenes Prädikat erfüllen (*beide* Teile des Paares werden dem Prädikat übergeben), eine Prozedur aus, die zwei Elemente hat: eines aus dem Argument-Domain und eines aus dem Co-Domain der Map.
Beispiel — Hexdump aller mit „F“ beginnenden Dateien, die weniger als 100 Datensätze haben:

```
Bool test_file (Str name, File f)
{ return hd_str (name) == ´F´ && len_file (f) < 100 }
forsome_dir (d1, test_file, hexdump);
```

5.6.4 Beispiel: Stücklistenverwaltung

Stückliste

Eine **Stückliste** enthält alle Teile, die für die Konstruktion eines Werkstücks benötigt werden. Für diejenigen Teile, die zusammengesetzt sind, also untergeordnete Bestandteile haben, ist zudem diese Zusammensetzung angegeben; diese Liste von Elementen ist für die nicht weiter zerlegbaren Teile (die „Rohstoffe") natürlich leer.

Es ist zwar sinnvoll, die *Anzahl* der benötigten Teile zur Zusammensetzung einer größeren Einheit ebenfalls in die Stückliste einzutragen. Für unser Beispiel wollen wir uns aber auf eine **qualitative Stückliste** beschränken, die für jedes zusammengesetzte Teil nur die benötigten Elemente enthält, unabhängig von ihrer Stückzahl.

Wir nehmen weiter an, daß ein Teil durch seine *Teilnummer* (part number) eindeutig gekennzeichnet ist.

Modell

Einem zusammengesetzten Teil wollen wir die (ungeordnete) *Menge* der Teilnummern seiner Bestandteile *zuordnen*. Ist das Teil nicht zerlegbar, soll diese Menge leer sein. Aus dieser Beschreibung leiten wir das abstrakte Modell ab: ein Eintrag in der Stückliste ist eine funktionale Zuordnung zwischen einer Teilnummer und dem (möglicherweise leeren) Set der Bestandteile dieses Teils; die Stückliste ist also eine Map von Teilnummern auf Sets von Teilnummern! Die Domain-Gleichungen zur Modellierung einer Stückliste Plist sieht so aus:

DSL

$$\begin{array}{l} \mathsf{DOMAINS} \\ \mathsf{Plist} \ = \mathsf{Pn} \ \overrightarrow{m} \ \mathsf{Pnset} \\ \mathsf{Pnset} = \mathsf{Pn} \text{–}\mathsf{set} \\ \mathsf{Pn} \quad\ = \mathsf{ELEMENTARY} \end{array}$$

Invariante

Wann ist eine Stückliste wohlgeformt ?

1. Wenn in allen Sets der Bestandteile zusammengesetzter Teile nur Teilnummern vorkommen, die auch durch einen Stücklisteneintrag beschrieben sind und
2. wenn kein Teil „rekursiv" spezifiziert ist (bildet man durch Auflösung der Stückliste die Menge *aller* notwendigen Teile zur Konstruktion eines Teils t, so darf t nicht selbst in dieser Menge vorkommen).

Die VDM-Spezifikation der Datentyp-Invarianten für die Stückliste sieht dann so aus:

$$\begin{array}{l} \textbf{inv-Plist}\,(\mathsf{plist}) = \\ \quad (\forall\ ps \in \textbf{rng}\,(plist))\ (ps \subseteq \textbf{dom}\,(plist)) \\ \quad \wedge\ (\forall\ p \in \textbf{dom}\,(plist))\ (p \notin parts\_pn\,(p, plist)) \end{array}$$

Die beiden obigen Bedingungen sind jeweils in einer Zeile der Invariante formalisiert:

1. Für alle Elemente aus der Menge aller Sets im Co-Domain einer Stückliste (mit **rng** ermittelt), die ja Sets von Teilnummern sind, muß gelten: sie sind in der Menge aller Argumentwerte der Stückliste (mit **dom** ermittelt) enthalten.

2. Für alle Teilnummern im Argument-Domain der Stückliste (wiederum mit **dom** ermittelt) muß gelten: sie sind *nicht* in der Menge ihrer Bestandteile (mit *parts_pn* ermittelt) enthalten.

Die Invariante wollen wir *hier* nicht überprüfen. Sie finden eine Anwendung in sehr ähnlicher Form, nämlich für eine quantitative Stückliste, im Kapitel 7, wo ein komplettes Beispiel für die Anwendung von VDM dargestellt wird.

parts_pn Die semantische Funktion **parts_pn** müssen wir selbst definieren; sie liefert den Set aller Bestandteile eines Teils. Dies ist auf jeden Fall eine nützliche Funktion für unsere Stückliste. Wir spezifizieren sie implizit:

TYPE parts_pn (pn, plist) (ps): Pn Plist $\longrightarrow$ Pnset
pre: pn $\in$ **dom** plist
post: ps =
$\{p \mid p \in \textbf{dom}\,(plist) \wedge (p \in\ plist\,(pn) \vee (\exists\ p_2 \in ps)\ (p \in plist\,(p_2)))\}$

postcondition Die *postcondition* verlangt natürlich nach einer Erklärung, zumal sie eine Rekursion enthält. *ps* ist der Set aller Teile *p*, für die gilt: *p* ist in der Stückliste beschrieben (also in der Argumentmenge enthalten) UND es ist entweder ein direktes Bestandteil des Teils *pn* (also in dessen Map-Wert enthalten) ODER es gibt ein Teil p_2 in dem hier gerade zu konstruierenden Set *ps*, dessen direktes Bestandteil *p* ist (sodaß also *p* im Map-Wert von p_2 enthalten ist).

Wegen der Rekursion bezüglich *ps* sollte man diese Funktion auch rekursiv realisieren. Ein Problem haben wir aber: der Typ des gewünschten Resultats ist nicht identisch mit dem Parametertyp! Wir behelfen uns, indem wir eine Hilfsfunktion **parts_pnset** schaffen, die als Argument einen Set *ps* von Teilen hat und einen Set aller Bestandteile der Teile in *ps* liefert. Diese Hilfsfunktion brauchen wir nur noch mit dem Set der direkten Bestandteile von *pn* aufzurufen und erhalten das gewünschte Ergebnis. Damit können wir die semantische Funktion parts_pn bereits implementieren (wir benutzen eine Kopie dieses Set, weil er von parts_pnset zerstört wird):

```
Pnset parts_pn (Plist plist, Pn pn)
{ return parts_pnset (plist, copy1_pnset (sel_plist (plist, pn))); }
```

parts_pnset Nun geht es darum, die Funktion **parts_pnset** zu spezifizieren. In jedem Rekursionsschritt wird zu dem Set *ps*, der als Argument übergeben wird, der Set *ps_sub* der direkten Bestandteile *aller* Elemente von *ps* ermittelt. Enthält dieser Set *ps_sub* ein Teil, das nicht in *ps* enthalten ist, wird die Funktion (mit der Vereinigung von *ps* und *ps_sub* als Argument) erneut aufgerufen; ist *ps_sub* aber vollständig in *ps* enthalten, sind wir fertig und *ps* enthält das Ergebnis. Die konstruktive VDM-Spezifikation von parts_pnset sieht demzufolge so aus:

TYPE parts_pnset (ps, plist): Pnset Plist $\longrightarrow$ Pnset
pre: $ps \subseteq$ **dom** $(plist)$
parts_pnset =
  if subparts_pnset $(ps) \subseteq ps$
    then ps
    else parts_pnset (subparts_pnset $(ps) \cup ps)$

Das können wir gleich in C implementieren:

```
Pnset parts_pnset (Plist plist, Pnset ps)
{
  Pnset ps_sub = copy1_pnset (ps);
  ps_sub = subparts_pnset (ps_sub, plist);
  if (is_subset_pnset (ps_sub, ps)) {
      del1_pnset (ps_sub);
      return ps;
  }
  else {
      ps = union_pnset (ps, ps_sub);
      return parts_pnset (plist, ps);
  }
}
```

Zunächst wird die Variable *ps_sub* erzeugt, indem das Argument, *ps*, kopiert wird (copy1_pnset), weil wir ihn später noch brauchen. Nun wird der Set aller *direkten* Bestandteile aller in ps_sub befindlichen Teile ermittelt (subparts_pnset) und wieder in *ps_sub* gespeichert. Ist *ps_sub* eine Teilmenge des Arguments *ps* (sind also keine neuen Teile hinzugekommen, ermittelt mit is_subset_pnset), braucht nur noch ps_sub wieder gelöscht zu werden (del1_pnset) und die Funktion ist beendet. Wenn nicht, wird das Argument *ps* mit *ps_sub* vereinigt (union_pnset) und parts_pnset erneut aufgerufen.

subparts_pnset

Noch sind wir nicht fertig: die neu eingeführte Hilfsfunktion **subparts_pnset** muß noch realisiert werden. Sie ermittelt die Vereinigung der Bestandteil-Sets *aller* Teile im ihr übergebenen Set. Für solche Operationen, die man mit dem Begriff „Für alle“ charakterisieren kann, ist häufig das Funktional fold geeignet; so auch hier. Die Nullfunktion muß einen leeren Pnset-Set liefern, dafür ist mk_empty_pnset geeignet; die Fold-Funktion vereinigt zwei Sets, dafür taugt union_pnset; die Konvertierungsfunktion wandelt ein Element des Set in den Resultattyp um, ermittelt also zu einem Teil die direkten Bestandteile mit Hilfe der Stückliste. Siehe da: die Konvertierungsfunktion benötigt Zugriff auf die Stückliste! Wir benötigen also für die mit fold erzeugte Funktion subparts_pnset ein zusätzliches Argument, die Stückliste, die der Konvertierungsfunktion übergeben wird. Der Typ dieses Arguments muß in der Implementations-Beschreibung von Pnset in der Parameterliste von fold auftauchen, und zwar mit dem Parameter **X** als erstem zusätzlichen Argument der Konvertierungsfunktion.

```
IMPLEMENTATIONS
Pnset   = ...
            + fold.subparts (RESULT="Pset",
                             CVFCT="getparts",
                             NULLFCT="mk_empty",
                             FOLDFCT="union",
                             X="Plist")
```

getparts Die als Konvertierungsfunktion angegebene Funktion **getparts** müssen wir natürlich auch noch definieren; außerdem brauchen wir ihren „richtigen" Namen, nach der Erweiterung. Er lautet getparts_pn:

```
Pnset getparts_pn (Pn pn, Plist plist)
{ return copy1_pnset (sel_plist (plist, pn)); }
```

Sie liefert eine *Kopie* des Set der Bestandteile des Teils *pn*, weil die Fold-Funktion der Funktion subparts_pnset ihre Argumente zerstört und wir die Stückliste plist später noch brauchen.

Zum Abschluß sehen Sie die fertige, mit fold generierte Funktion subparts_pnset, die Sie nicht selbst zu schreiben brauchen (sie wird ja vom VDM-Compiler generiert):

```
Pnset subparts_pnset (Pnset ps, Plist plist)
{
  Pn pn;
  if (is_empty_pnset (ps))
      return mk_empty_pnset ();
  else {
      pn = sel_pnset (ps);
      return union_pnset (getparts_pn (pn, plist),
                          subparts_pnset (sub1_pnset (ps, pn)));
  }
}
```

5.6.5 Beispiel: Heap-Speicherverwaltung

In diesem Abschnitt wird eine konventionelle Heap-Verwaltung in *C* einer Heap-Verwaltung mit Hilfe einer Map von Pointern (Token) auf Objekte gegenübergestellt. Wir wollen hier der Einfachheit halber nur Zahlen auf dem Heap manipulieren; das Prinzip läßt sich aber auch auf komplexe Datentypen anwenden. Zunächst die DSL-Spezifikation:

DOMAINS
Heap $=$ Token $\overrightarrow{m}$ Nat0
Token $=$ ELEMENTARY
IMPLEMENTATIONS
Heap $=$ heap
$+$ mk_empty $+$ union1 $+$ sub1 $+$ sel $+$ ovwrt1 $+$ ext
Token $=$ ptr(ELEM=Nat0) $+$ del $+ \ldots$

In C:
#typedef Nat0 $\star$ Adr;

Token ist also der Typ der „Pointer auf Nat0"; Heap ordnet diesen Pointern Werte des Typs Nat0 zu. In C wird der Typ Adr als Pointer auf Nat0 vereinbart.

Die Map hat einen Nachteil: die Operationen haben die bei Maps übliche Syntax (benötigen also eine Map als Argument), obwohl das eigentlich unnötig ist: die Paare der Map sind ja nicht irgendwie zusammengefaßt unter einem Namen ansprechbar. Der Grund dafür ist allerdings einleuchtend: wenn Ihnen die Arbeit mit Heap-Objekten und die notwendige Speicherverwaltung einmal zu gefährlich, zu schwierig oder zu unübersichtlich wird, können Sie problemlos eine andere Map-Implementation auswählen, denn Ihr Programm enthält bereits alle Operationsaufrufe mit der korrekten Syntax, Sie haben bereits ein Map-Objekt eingerichtet usw. Die Initialisierung der Map sieht also so aus:

Heap h = mk_empty_heap ();

Beachten Sie, daß dieses Objekt „virtuell" ist, denn es belegt keinen Speicher, stört Sie im Programm also nicht weiter.

Die üblichen Heap-Operationen werden nun einander gegenübergestellt.

Objekt erzeugen

In C erzeugen Sie ein Objekt auf dem Heap, indem Sie Speicher anfordern; dazu rufen Sie die Standard-Funktion malloc mit der Größe des Objekttyps als Parameter auf und wandeln den Ergebnispointer in den Typ Token um. Danach initialisieren Sie das Objekt mit Hilfe einer Dereferenzierung. Bei der Map rufen Sie die Operation ext auf, die genau das gleiche durchführt, was Sie in C „per Hand" machen. In einer anderen Implementation würden Sie die Operation union1 verwenden, da der Argumentwert dann ja von Ihnen bestimmt wird und nicht dynamisch von malloc ermittelt wird.

```
Adr c1;
Token m1;
c1 = (Token) malloc (sizeof (Nat0));   *c1 = 14;
m1 = ext_heap (&h, 14);
```

Objekt löschen

In C löschen Sie ein Objekt auf dem Heap, indem Sie den Speicher wieder freigeben (mit free) und danach den Pointer sicherheitshalber löschen. Bei der

Map entfernen Sie das Objekt mit **sub1**, nachdem Sie es mit **del** gelöscht haben. **sub1** ist nur nötig, wenn Sie den Austausch der Implementation erwägen.

```
free ((void *) c1);    c1 = NULL;
del_token (m1);    h = sub1_heap (h, m1);
```

Auslesen

Zum Auslesen des Heap-Objekts in *C* dereferenzieren Sie den Pointer; bei der Map verwenden Sie die Operation **sel**.

```
printf ("\n%d %d", *c1, sel_heap (h, m1));
```

Überschreiben

In *C* weisen Sie dem dereferenzierten Pointer (der dadurch mit dem Objekt identisch wird) einen neuen Wert zu. Bei der Map verwenden Sie die Operation **ovwrt1** mit einem bereits initialisierten Token und dem neuen Inhalt.

```
*c1 = 9;    h = ovwrt1_heap (h, m1, 9);
```

Sie sehen: der Unterschied ist nicht *sehr* groß und der kleine Mehraufwand sollte für den Fall, daß die Performance nicht die größte Rolle spielt, in Kauf genommen werden. Die Implementation "heapmap" z.B. ist der Implementation „heap" sehr ähnlich — allerdings werden die Argumentwerte in einem Set gehalten, so daß *alle* Map-Operationen zur Verfügung stehen, ohne daß die Performance der obigen Heap-Operationen wesentlich schlechter wird.

5.7 Optionals

5.7.1 Was sind Optionals ?

Definition

Ein **Optional** enthält entweder ein Objekt vom angegebenen Elementtyp oder einen besonderen Wert **nil**, der anzeigt, daß das Optional gerade *kein gültiges Objekt* enthält. Das Optional ist dann auch *nicht vorhanden*. Dies sind die beiden möglichen Interpretationen von **nil**, die in der Praxis benötigt werden.

Zusammenfassung

- Der aktuelle Inhalt eines Optionals kann jederzeit durch entsprechende Operationen geändert werden.
- Der Datentyp der Objekte kann beliebig gewählt werden, es kann also ein beliebig strukturierter Typ sein.
- Der Wert **nil** ist immer unterschiedlich von allen gültigen Zuständen des Objekts, kann also nicht mit einem gültigen Objektzustand verwechselt werden.
- Ein Optional mit dem Wert **nil** kann als „nicht existent" oder als „ungültig" interpretiert werden.

Diese Formulierungen erinnern Sie vielleicht an den NULL-Pointer in *C*, der gerade auf kein Objekt zeigt. Tatsächlich ist **nil** diesem NULL-Pointer sehr ähnlich, kann jedoch für beliebige Elementtypen eingerichtet werden; den NULL-Pointer gibt es eben nur für Pointer.

Abstraktion ! Wenn Sie den ungültigen Zustand eines beliebigen Objekts in einer Applikation modellieren wollen, machen Sie sich Gedanken, welche Werte in der Praxis auftreten können, also „gültige“ Zustände beschreiben; aus der Menge der übrigen Zustände wählen Sie dann einen heraus, der in der gesamten Applikation für die Modellierung eines ungültigen Objekts dienen soll. Das Optional hilft Ihnen dabei, die Konsistenz der Applikation zu wahren, indem Sie abstrahieren vom tatsächlichen Inhalt des Objekts, wenn es ungültig ist: Sie legen *einmal*, bei der Definition des Optionals mit der DSL, den Wert des Objekts fest, der nil repräsentieren soll — in der Applikation können Sie dann die Optional-Operationen verwenden, ohne über den tatsächlichen „Wert“ von nil nachzudenken. Gerade bei vielen verschiedenen nil-Werten für verschiedene Objekttypen werden Sie es zu schätzen wissen, immer die gleichen Operationen zu verwenden. Ein weiterer positiver Nebeneffekt: muß irgendwann und unvorhersehbar der Wert für nil geändert werden (weil der einst gewählte Wert plötzlich doch „gültig“ ist), brauchen Sie nur *eine* Stelle in der DSL-Spezifikation zu ändern und die Applikation neu zu übersetzen — die Wartung wird durch Verwendung von Optionals also einfacher !

DSL Um in der DSL-Spezifikation ein Optional *Anopt* für einen Domain $\mathcal{A}$ zu definieren, schreiben Sie

$$Anopt \quad = \quad [\ A\ ]$$

Anwendungen Wegen der Ähnlichkeit mit NULL-Pointern in C ist deren „klassische“ Anwendungsmöglichkeit durch Optionals realisierbar: die Repräsentation eines nicht vorhandenen Nachfolgers in einer rekursiven Struktur (z.B. das Ende einer verketteten Liste oder die Blatt-Ebene eines Binärbaumes). Eine eher philosophische Anmerkung dazu: Die Sichtweise des Abbruchs einer rekursiven Struktur mit einer (grundsätzlich beliebigen) Ende-Markierung in Form von nil ist wesentlich „natürlicher“ und abstrakter — weil sie mehr an die eigentliche Aufgabenstellung der Modellierung einer rekursiven Struktur angelehnt ist — als wenn Sie nur die aus der *Implementation* erwachsende Möglichkeit der Ende-Markierung mit dem NULL-Pointer in Erwägung ziehen.

Sie können Optionals aber auch z.B. im Compilerbau gut einsetzen, da die Grammatik-Spezifikationen mit der BNF (Backus-Naur-Form) ebenfalls den optionalen Fall kennen (der auch mit eckigen Klammern dargestellt wird). Der zu schreibende Parser für die Syntaxanalayse muß in der Lage sein, fehlende Teile in irgendeiner Form darzustellen; dazu eignen sich Optionals.

Beispiel: Die Grammatik der zweiseitigen Bedingung in C (in Form von **if**...**else**) sieht vor, daß der ELSE-Zweig auch entfallen kann:

IF-ELSE ::= IF (expression) statement ; [ELSE statement ;]

Dieser Abschnitt enthält nur die Beschreibung des VDM-Basistyps „Optional“ und die verfügbaren Implementationen und Operationen; ein Anwendungsbeispiel finden Sie im Abschnitt 5.8 über Trees.

→ 5.8

5.7.2 Optional-Implementationen

Für Optionals stehen zwei Implementationen zur Verfügung:

- **ptelem** repräsentiert nil mit dem NULL-Pointer (wer hätte das gedacht ?), wenn der Elementtyp als Pointer implementiert ist. Das ist z.B. nicht der Fall für die normalen elementaren Datentypen. Als Zusatzbedingung darf der NULL-Pointer natürlich auch keinen gültigen Element-Wert darstellen; damit sind die „linklist“-Implementationen für Sets, Tuples und Maps nicht zulässig, wo der NULL-Pointer den Zustand „leeres Objekt“ darstellt.
- **elemonil** erwartet von Ihnen die *zwingende* Angabe des Objekt-Wertes, der nil repräsentieren soll (im Parameter NILVAL). Sie müssen „elemonil“ verwenden, wenn „ptelem“ nicht verwendet werden kann; für die „linklist“-Implementationen von Sets, Tuples und Maps bietet sich der Wert -1 für nil an, der in der Zweierkomplement-Darstellung den höchstmöglichen Wert eines Pointers ergibt.

5.7.3 Optional-Operationen

Zur Veranschaulichung der Operationen dienen folgende Variablen:

```
DOMAINS
Nopt  = [ Nat0 ]
Sopt  = [ Nset ]
Nset  = Nat0 –set
⋮
Nat0  a, b;
Nopt  o, p;
Nset  s1, s2;
Sopt  so1, so2;
```

Optional-Konstruktoren

mk Mit **mk** erzeugen Sie ein Optional, das einen gültigen Wert enthält. Der Name wird mit dem Namen des Elementtyps *und* dem Optional-Namen erweitert; daher ist der Umgang mit mk recht umständlich. Tip: verwenden Sie lieber die Operation mk_opt, sie hat die gleiche Wirkung.

o = mk_nat0_nopt (14);

mk_opt **mk_opt** ist die „bessere“ Variante zu mk, da der VDM-Basistyp „Optional“ am Namen erkennbar ist und die Nomenklatur leichter zu behalten ist, weil sie den bisher kennengelernten Normen entspricht.

so1 = mk_opt_sopt (mk_one_nset (9));

mk_nil Mit **mk_nil** erzeugen Sie ein Optional-Objekt mit dem Wert nil. Wenn Sie einem Optional, das gerade einen gültigen Inhalt hat, den Wert nil geben möchten, sollten Sie den vorherigen Inhalt mit del1 oder del löschen, wenn er nicht noch von anderer Stelle referenziert wird — Sie verhindern so eine Verschwendung von Speicherplatz.

del1_sopt (so1);
so1 = mk_nil_sopt ();

Optional-Prädikate

is_eq Mit **is_eq** testen Sie, ob zwei Optionals entweder beide das gleiche Element enthalten oder beide gerade nil sind.

is_eq_nopt (mk_opt_nopt (14),mk_nil_nopt ()) == FALSE

is_ge Mit **is_ge** testen Sie, ob ein Optional gerade größer oder gleich einem anderen Optional ist. Der Vergleich ist nur definiert, wenn is_ge auch für den Elementtyp erhältlich ist, also z.B. nicht für den Typ Sopt !

is_gr Mit **is_gr** testen Sie, ob ein Optional gerade größer als ein anderes Optional ist. Der Vergleich ist ebenfalls nur definiert, wenn is_gr auch für den Elementtyp erhältlich ist.

! → Die Operationen is_ge und is_gr werden nur der Vollständigkeit halber bereitgestellt, denn einige Implementationen von anderen VDM-Basistypen erwarten die Existenz dieser Prädikate bei allen ihren Elementtypen, auch bei Optionals; ob diese Prädikate irgendwie sinnvoll sind, ist unwichtig. Sie werden diese Prädikate bei eigenen Optional-Anwendungen in der Regel nicht selbst benötigen.

is_nil **is_nil** liefert *TRUE*, wenn das Optional gerade den Wert nil hat.

is Der Name der Operation **is** wird sowohl mit dem Namen des Elementtyps als auch mit dem Optional-Namen erweitert und liefert *TRUE* genau dann wenn is_nil das Ergebnis *FALSE* hat und umgekehrt.

o = mk_nat0_nopt (14);
⇒ is_nat0_nopt (o) == TRUE
⇒ is_nil_nopt (o) == FALSE

Es gibt ein weiteres Prädikat, das aber gleichzeitig ein Element liefert, wenn es erfüllt ist: is_opt wird daher bei den Selektions-Operationen erläutert.

Optional: Selektion

s s wird (wie auch mk und is) sowohl mit dem Namen des Elementtyps als auch mit dem Optional-Namen erweitert und liefert — wenn das Optional nicht gerade nil enthält — den Element-Wert.

```
if (is_nat0_nopt (o)) { a = s_nat0_nopt (o); process_nat0 (a); }
```

is_opt **is_opt** ist eine Verknüpfung der Prüfung, ob ein Optional gerade einen gültigen Wert hat (mit is) und, im Erfolgsfall, Selektion des Elements (mit s). Das Resultat ist vom Typ Bool, im Falle der Erfüllung dieses Prädikats wird das Element in einem Referenzparameter geliefert.
Das obige Beispiel zu s kann also auch einfacher formuliert werden:

```
if (is_opt_nopt (o,&a)) process_nat0 (a);
```

5.8 Trees

5.8.1 Was ist ein Tree ?

Definition Ein **Tree** ist das kartesische Produkt seiner Komponententypen, d.h. er kann alle möglichen Kombinationen von Werten seiner Komponententypen enthalten. Ein Tree-Objekt ist also eine festgelegte Aggregation *mehrerer* Komponenten eventuell unterschiedlichen Typs. Etwas ausführlicher:

Zusammenfassung

- Jedes Tree-Objekt enthält *alle* in der Tree-Definition festgelegten Komponenten. Nie kommen welche hinzu, nie werden welche entfernt (das ist ein großer Unterschied zu Sets, Tuples und Maps).
- Die Komponenten dürfen verschiedene Datentypen haben. Es dürfen aber auch mehrere Komponenten vom gleichen Datentyp enthalten sein.
- Die Datentypen der Komponenten sind *beliebig*. Insbesondere dürfen es strukturierte Typen sein; auch Trees sind erlaubt.
- Die Komponenten sind entsprechend der angegebenen Reihenfolge bei der Definition angeordnet. Weitere Ordnungsprinzipien gibt es für Tree-Objekte a priori nicht.

Kartesisches Produkt Das kartesische Produkt kennen Sie von Funktionen mit mehreren Argumenten; sie bilden das kartesische Produkt der Argumenttypen auf den Resultattyp ab. Üblicherweise schreibt man das kartesische Produkt wie folgt:

$$\mathcal{KP} \quad = \quad \mathcal{A} \times \mathcal{B} \times \ \ldots \ \times \mathcal{N}$$

Bei der Beschreibung von Trees werden die Kreuz-Operatoren üblicherweise weggelassen — die Komponententypen werden einfach aufgezählt:

$$\mathcal{T}ree \quad = \quad \mathcal{A} \quad \mathcal{B} \quad \ldots \quad \mathcal{N}$$

struct

Natürlich kommt Ihnen das Konzept der Trees bekannt vor: in *C* gibt es den Typgenerator **struct** zur Aggregation mehrerer Komponenten unter einem Namen. Tatsächlich entspricht die praktische Konzeption des Tree-Datentyps im VDM-System dem *struct*, Trees werden mit ihrer Hilfe implementiert.

Abstraktion

Ein (theoretischer) Unterschied besteht nur darin, daß ein Tree von seiner Repräsentation abstrahiert, während einem *struct* mit seiner Definition gleichzeitig seine Repräsentation zugeordnet wird. Sie können mit Trees also genauso umgehen wie bisher mit *structs*, allerdings bringt der Satz an Operationen für Trees eine Einheitlichkeit in die Handhabung von Trees, die bei *structs* nicht möglich ist, eben weil deren Repräsentation durch die Definition feststeht.

Beispiel

Ein Beispiel: Sie bauen einen Binärbaum auf; jeder Knoten hat ein Datenfeld und zwei Nachfolger. In einer Implementation mit *structs* realisieren Sie die Nachfolger mit Hilfe von Pointern auf gleichartige *structs*; bei jedem Zugriff auf einen Nachfolger benötigen Sie dieses Wissen, weil Sie den Pointer erst dereferenzieren müssen. Bei einem Tree hingegen kümmern Sie sich nicht um die Implementation, sondern verwenden *immer* die für jeden Tree vorhandenen Tree-Operationen. Außerdem können Sie zwei beliebig komplexe Binärbäume ohne Wissen über die interne Struktur auf Gleichheit testen — bei einem *struct* ist ein selbstgeschriebenes Prädikat erforderlich, das Wissen über die Implementation zwingend erfordert!

Operationen

Die Arbeit mit Trees erfordert nur sehr wenige Operationen: Erzeugung eines Tree-Objekts, Selektion einer Komponente, Überschreiben einer Komponente mit einem neuen Wert und Test auf Gleichheit zweier Trees.

Trees in *C*

Aus der Wahl von *C* als Basissprache für das VDM-System ergeben sich zwei Aspekte, die in der bisher gegebenen Beschreibung von Trees *nicht* auftraten, da sie eben implementationsabhängig sind:

- Aus der geeigneten Repräsentation von Trees durch *structs*, in denen die Reihenfolge der Komponenten festgelegt ist, erwächst die Möglichkeit, zwei weitere Prädikate zu definieren: Tests auf „Größer oder gleich“ und auf “Größer als“, die allerdings nur sinnvoll sind, wenn die Komponenten diese Prädikate erlauben. Beispiel: ein Tree, der einen Studenten-Datensatz enthält, bestehend aus Name (ein VDM-String), Geburtsdatum (drei Nat1-Werte) und Matrikelnummer (Nat1), erlaubt eine Sortierung nach dem Namen als erster Komponente.

- Die Operationen werden durch *C*-Funktionen realisiert, die sich natürlich durch ihre Namen unterscheiden müssen. Die Selektions- und Überschreibe-Operationen gibt es aber für jede einzelne Komponente —

die übliche Form der Namenserweiterung mit dem Domain-Namen taugt also nicht. Auch die Erweiterung mit den Namen der Komponententypen ist nicht ausreichend, da mehrere Komponenten vom gleichen Typ sein können. Daher wird in der Spezifikation eines Tree gefordert, daß

Nomenklatur

jede Komponente einen (von Ihnen definierten) Namen erhält; er muß mit einem „s" beginnen, gefolgt von einem Unterstrich, danach folgt noch mindestens ein Kleinbuchstabe sowie weitere optionale Kleinbuchstaben, Ziffern und Unterstriche (siehe auch die DSL-Syntax-Definition in Anhang A). Die komponentenbezogenen Operationen werden dann mit den Komponentennamen *ohne* die Anfangssequenz „s_" erweitert: aus chg wird für eine Komponente „s_value1" also der Name chg_value1.

DSL

Um in der DSL-Spezifikation einen Tree *Atree* der Komponenten-Datentypen $\mathcal{A}_1$, $\mathcal{A}_2 \ldots \mathcal{A}_n$ zu definieren (deren Komponentennamen hier zufällig gewählt wurden, indem einfach die Sequenz „s_" dem Namen des Komponententyps vorangestellt wurde), schreiben Sie

$$Atree \quad :: \quad s\_a1 : A_1 \quad s\_a2 : A_2 \ldots \; s\_an : A_n$$

! →

Beachten Sie, daß hier *nicht* das Gleichheitszeichen verwendet wird, sondern zwei Doppelpunkte!

Spezifikation mit VDM

Für Trees gibt es in der VDM-Spezifikation zwei Vereinfachungen, was die Komponenten-Selektion in der **let**-Klausel betrifft:

Statt:
let b1 = s_B1 (tree), b2 = s_B2 (tree), ... , bn = s_bn (tree) **in** ...
schreiben Sie
let mk-Tree (b1,b2, ... ,bn) = tree **in** ...

Wenn Sie nur einige Komponenten benötigen, lassen Sie die übrigen weg:
let mk-Tree (b1, , ,b4, ... ,b_{n-1},) = tree **in** ...

Anwendung

Die übliche Anwendung ist die Modellierung von *Datensätzen*, also Sammlungen von verschiedenartigen Informationen über ein Objekt, verbunden mit der Möglichkeit, diese Datensätze mit Hilfe von Rekursion in eine bestimmte Struktur zu bringen, die z.B. die Sortierung nach bestimmten Kriterien erlaubt (z.B. Baumstrukturen oder verkettete Listen).

Beispiel:

Wir modellieren einen Studenten-Datensatz mit Name, Geburtsdatum und Matrikelnummer:

```
DOMAINS
Student :: s_name   : Str
           s_day    : Nat1
           s_month  : Nat1
           s_year   : Nat1
           s_matrno : Nat1
```

5.8.2 Tree-Implementationen

Für Trees stehen zwei Implementationen zur Verfügung:

- **struct** modelliert einen Tree mit einem *C–struct*. Die dynamische Speicherallokation ist unnötig und der Zugriff erfolgt direkt, aber Zuweisungen erfordern das Kopieren der gesamten Struktur. Die Realisierung von rekursiven Trees ist mit „struct“ *nicht* möglich.
- **ptstruct** erzeugt den Tree als *C–struct* auf dem Heap und liefert einen Pointer zurück. Der Zugriff erfolgt über die Dereferenzierung des Pointers (langsamer als direkter Zugriff), aber eine Zuweisung erfolgt durch Pointer-Zuweisung (schnell, wenn keine „echte“ Kopie benötigt wird, die mit copy / copy1 erzeugt werden muß). Mit „ptstruct“ können Sie rekursive Trees realisieren.

5.8.3 Tree-Operationen

Zur Veranschaulichung dienen folgende Variablen:

```
Student  s1,s2;
Str      name1,name2;
Nat1     day,month,year,matr_no;
```

Tree-Konstruktoren

mk Mit **mk** erzeugen Sie einen Tree aus der vollständigen Liste der Komponenten.
Beispiel — Erzeugung eines Studenten-Datensatzes:

name1 enthält "Meier Fritz";
s1 = mk_student(name1, 7, 12, 1955, 142857);

copy / copy1 Auch für Trees gibt es natürlich die Operationen zum Kopieren eines anderen Tree.

Tree-Kombinatoren

Nomenklatur Es gibt zwei Klassen von Kombinatoren; die einen zerstören den vorherigen Zustand des Tree (tragen also die neue Komponente direkt in den gewünschten Tree ein), die anderen kopieren den Tree vorher und erhalten daher den vorherigen Zustand. Beide Kombinatoren werden mit dem variablen Teil des Komponenten-Namens erweitert.

subst **subst** überschreibt die gewünschte Komponente, nachdem der Tree vorher kopiert wurde; es wird also gleichzeitig ein neuer Tree erzeugt, weshalb subst auch zu den Konstruktoren gerechnet werden könnte. Vorsicht: Zuweisungen

der Form
s1 = subst_xxx (s1, xxx);
erzeugen einen nicht-referenzierten Tree, weil der vorherige Inhalt von **s1** ja nicht zerstört wird — Sie sollten **subst** also immer mit zwei verschiedenen Tree-Objekten benutzen.
Beispiel — Aufnahme eines weiteren Fritz Meier, der sogar am gleichen Tag geboren ist, unter anderer Matrikelnummer:

s2 = subst_matrno (s1, 428571);

chg **chg** überschreibt die gewünschte Komponente bei gleichzeitiger Zerstörung des vorherigen Tree-Zustands. Die Sub-Strukturen werden allerdings, wie bei **subst**, nicht gelöscht; das müßte durch ein explizites **del** geschehen.
Beispiel — Änderung der Matrikelnummer des ersten Fritz Meier:

s1 = chg_matrno (s1, 857142);

Tree-Prädikate

is_eq Mit **is_eq** testen Sie, ob zwei Trees komponentenweise gleich sind.
Beispiel — der Vergleich der beiden Fritz Meier ergibt Unterschiede:

is_eq_student (s1,s2) == FALSE

is_gr Mit **is_gr** testen Sie komponentenweise, ob ein Tree größer als ein anderer Tree ist; das ist allerdings nur möglich, wenn das Prädikat **is_gr** auch für *alle* Komponententypen definiert ist.
Beispiel — der zuerst aufgenommene Fritz Meier hat die höhere Matrikelnummer (sie wurde ja auf 857142 geändert):

is_gr_student (s1, s2) == TRUE

is_ge Mit **is_ge** testen Sie komponentenweise, ob ein Tree größer oder gleich einem anderen Tree ist, d.h. ob eines der beiden obigen Prädikate erfüllt ist. Auch **is_gr** muß für *alle* Komponententypen definiert sein.

is_ge_student (s1, s2) == TRUE

Selektion von Tree-Komponenten

s Die Selektion einer Komponente hat den generischen Namen **s** und wird mit dem variablen Teil des Komponenten-Namens erweitert — es ergibt sich also jeweils der komplette Komponentenname.
Selektion der Komponenten des ersten Fritz Meier:

```
name1   = s_name (s1);
day     = s_day (s1);
month   = s_month (s1);
year    = s_year (s1);
matr_no = s_matrno (s1);
```

5.8.4 Beispiel: Der Binärbaum

Ein **Binärbaum** besteht aus *Knoten*, die Daten enthalten und eventuell einen oder zwei Nachfolger-Knoten. Hat ein Knoten keine Nachfolger (endet ein „Zweig“ des Baumes also bei diesem Knoten), wird er auch *Blatt* genannt. Der Ausgangsknoten eines Baumes heißt *Wurzel*. Die Nachfolge-Knoten werden üblicherweise als linker und rechter Nachfolger bezeichnet. Jeder Knoten bildet die Wurzel eines Teilbaumes, der wieder ein Binärbaum ist.

Anwendung

Ein Binärbaum dient dazu, die enthaltenen Daten in eine bestimmte Ordnung zu bringen, also muß die Relation „größer als“ für den Elementtyp definiert sein. Dazu wird ein neues Element in den Baum derart aufgenommen, daß für jeden Knoten gilt: sein Datenelement ist größer als alle Datenelemente in dem Teilbaum „unter“ seinem linken Nachfolger, aber kleiner als alle Datenelemente in dem Teilbaum „unter“ seinem rechten Nachfolger. Abb. 5.4 zeigt einen Binärbaum von natürlichen Zahlen, der dieser Ordnung unterliegt.

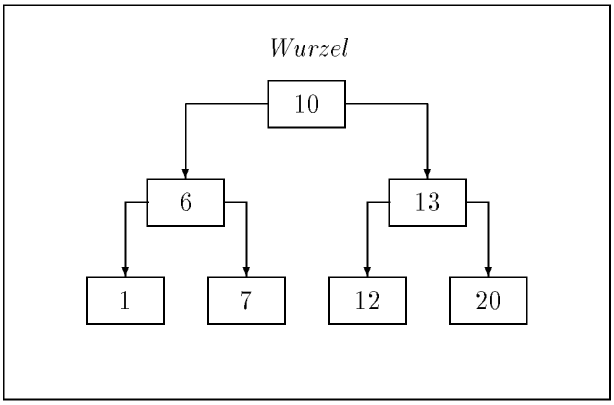

Abbildung 5.4: Beispiel für Trees: der Binärbaum

Der Binärbaum ist *balanciert*, wenn für *jeden* Teilbaum gilt: die Anzahl Knoten im linken „Sub-Baum“ unterscheidet sich von der Anzahl Knoten im rechten „Sub-Baum“ um maximal eins. Unser Beispiels-Baum ist also balanciert.

Aufgabe

Wir wollen zunächst einen gegebenen Set von natürlichen Zahlen in einen Binärbaum „hineinsortieren“; danach möchten wir weitere Werte einfügen; schließlich soll es möglich sein, zu prüfen, ob ein bestimmtes Element im Baum enthalten ist oder nicht. Dazu benötigen wir drei semantische Funktionen: conv_nset wandelt den Set in einen äquivalenten Binärbaum um, exist prüft, ob ein Wert bereits im Baum enthalten ist, insert fügt einen neuen Wert in den Baum ein. Die Funktion conv_nset kann die Funktion insert verwenden, denn sie fügt alle Set-Elemente in den anfänglich leeren Baum ein.

Domain

Zunächst müssen wir für den Binärbaum eine Domain-Gleichung aufstellen. Aus der obigen Definition ersehen wir, daß jeder Knoten einen optionalen linken und einen optionalen rechten Nachfolger hat, der jeweils wieder ein Binärbaum ist; jeder Knoten „enthält“ also keinen, einen oder zwei

Binärbäume. Es bietet sich daher an, einen Binärbaum als Optional zu modellieren, das den Wert nil hat, wenn er keinen Knoten enthält, sonst aber seinen Wurzelknoten. Ist ein Knoten vorhanden, enthält er einen linken und einen rechten Nachfolge-Baum sowie ein Datenelement (also eine natürliche Zahl), kann daher mit Hilfe eines Tree modelliert werden. Sie erkennen vielleicht, daß die Repräsentation eines Binärbaumes rekursiv ist: ein Baum ist ein Knoten, der eventuell wieder Bäume enthält usw. Die Rekursion bricht auf der Blatt-Ebene ab, weil dort alle „Sub-Bäume" leer sind (durch nil gekennzeichnet). Daraus ergibt sich folgende DSL-Spezifikation für einen Binärbaum (abgekürzt auch „Btree" genannt):

DOMAINS

Btree	=	[Node]	
Node	::	s_lt	: Btree
		s_data	: Nat0
		s_rt	: Btree
Nset	=	Nat0 –set	

IMPLEMENTATIONS

Btree	=	ptelem + ...
Node	=	ptstruct + ...
Nset	=	linklist + ...

Initialisierung Ein leerer Binärbaum ist durch den Wert nil gekennzeichnet, also erfolgt die Initialisierung mit Hilfe der Optional-Operation mk_nil.

exist Die Funktion **exist**, die die Existenz des Wertes e im (Teil-)Baum b ermittelt, ist sehr einfach zu spezifizieren: für den Fall, daß der Teilbaum nicht vorhanden ist (also $b =$ nil), ist das Ergebnis *FALSE*. Im anderen Fall wird e mit dem Datenelement der Wurzel von b verglichen — sind die Werte gleich, ist das Ergebnis *TRUE*; ist e kleiner als das Datenelement, muß im linken „Sub-Baum" weitergesucht werden, sonst im rechten. Die VDM-Spezifikation für exist und die C-Realisierung können wir sofort erstellen:

```
TYPE exist (b,e):   Btree   Nat0   ⟶   Bool
exist =
  cases b:
      nil  →  false
      mk-node (lt,data,rt)  →
          ((e = data) →  true,
           (e < data) →  exist (lt,e),
           T  →  exist (rt,e) )
```

Daraus wird die Funktion:

```
Bool exist (Btree b, Nat0 e)
{
  Node n;
  Nat0 data;
  if (is_nil_btree (b))
      return FALSE;
  else {
      n = s_node_btree (b);
      data = s_data (n);
      if (e == data)
          return TRUE;
      else
          if (e < data)
              return exist (s_lt (n),e);
          else
              return exist (s_rt (n),e);
  }
}
```

insert Die Funktion **insert** zum Einfügen eines neuen Wertes e arbeitet folgendermaßen: ist der (Teil-)Baum b leer, wird ein Baum mit *einem* Knoten (der das Datenelement e enthält) erzeugt und zurückgeliefert. Andernfalls wird e mit dem Datenelement der Wurzel von b verglichen:

- Sind die Werte gleich (ist e also bereits im Baum enthalten), passiert nichts und b wird zurückgeliefert.
- Ist e kleiner als das Datenelement, wird der *linke* „Sub-Baum“ ersetzt durch den linken „Sub-Baum“, in den e eingefügt wurde.
- Ist e größer als das Datenelement, erfolgt analog das Ersetzen des *rechten* „Sub-Baums“.

Hier die VDM-Spezifikation von insert:

```
TYPE insert (b,e):   Btree   Nat0   ⟶   Btree
insert =
  cases b:
      nil   →   mk-node (nil, e, nil)
      mk-node (lt,data,rt)   →
          ((e = data) →   b,
           (e < data) →   subst (lt,insert (lt,e)),
             T        →   subst (rt,insert (rt,e))  )
```

Auch diese Funktion können wir sofort in C implementieren, denn die Hilfsfunktion subst, die einen Teilbaum durch einen anderen ersetzt, steht uns als

! → Tree-Operation zur Verfügung. Beachten Sie, daß mit subst_node eine Kopie des Teilbaumes erzeugt wird; daher müssen wir das Original nachher mit del löschen.

```
Btree insert (Btree b,Nat0 e)
{
  Node n,n2;
  Nat0 data;
  if (is_nil_btree (b))
      return mk_opt_btree (mk_node (mk_nil_btree (), e, mk_nil_btree ()));
  else {
      n = s_node_btree (b);
      data = s_data (n);
      if (e == data)
          return b;
      else {
          if (e < data)
              n2 = subst_node (n, insert (s_lt (n), e));
          else
              n2 = subst_node (n, insert (s_rt (n), e));
          del_node (n);
          return n2;
      }
  }
}
```

conv_nset Die Funktion **conv_nset** soll aus einem Set von natürlichen Zahlen einen sortierten Binärbaum erzeugen, ist also ein Konstruktor. Dazu erzeugt sie zunächst einen leeren Binärbaum (mit mk_nil); dieser wird nun einer Hilfsfunktion ins_nset übergeben, die einen Set in einen existierenden Binärbaum einfügt. Weil der Set von ins_nset zerstört wird, übergibt conv_nset eine Kopie des Set.

```
Btree conv_nset (Nset ns)
{
  Btree b = mk_nil_btree ();
  return ins_nset (b, copy1_nset (ns));
}
```

ins_nset Die Hilfsfunktion **ins_nset** ist rekursiv. Ist der einzufügende Set leer, wird der aktuelle Binärbaum zurückgeliefert — sonst wird eine Zahl aus dem Set selektiert, in den Binärbaum eingefügt (mit insert) und aus dem Set entfernt; mit dem kleineren Set und dem größeren Binärbaum wird nun ins_nset erneut aufgerufen. Hier die zugehörige VDM-Spezifikation:

```
TYPE ins_nset (b, ns):   Btree   Nset   ⟶   Btree
ins_nset =
  if ns = {}
      then b
      else let e ∈ ns in
          ins_nset (insert (b, e), ns \ {e})
```

Diese Funktion müssen wir noch in C implementieren:

```
Btree ins_nset (Btree b,Nset ns)
{
  Nat0 e;
  if (is_empty_nset (ns))
      return b;
  else {
      e = sel_nset (ns);
      return ins_nset (insert (b,e),sub1_nset (ns,e));
  }
}
```

Wie bereits erwähnt, zerstört diese Funktion den Set, der ihr als Argument übergeben wird. Beachten Sie: die Struktur dieser Funktion ist identisch mit den mit dem Funktional binary für Sets erzeugten Funktionen, nur der Datentyp des ersten Arguments und des Resultats sind anders — die „Fold-Funktion" dient zur Verarbeitung des ersten Arguments (hier ein Tree, bei binary ein Set) und einem Element des zweiten Arguments (das sowohl hier als auch bei binary ein Set ist).

5.9 Unions

5.9.1 Was ist eine Union ?

Definition

Eine **Union** ist eine Klasse von Objekten, deren Datentyp nicht statisch festgelegt ist, sondern aus einer Menge mehrerer möglicher Datentypen stammt und sich dynamisch ändern kann.

Sprechweisen

Ein Objekt dieser Klasse wird oft selbst als Union bezeichnet; zur Vermeidung von Irritationen wird hier allerdings eine Trennung versucht. Die verschiedenen möglichen Datentypen eines Union-Objekts prägen den Begriff der „Union-Komponenten" oder „Union-Elemente", von denen immer eine(s) aktuell ist, die anderen sind sozusagen *virtuell*: es gibt sie irgendwo, und es gibt Operationen, um sie real zu machen. Der „Inhalt" eines Union-Objekts heißt so, weil die Verarbeitung erst nach einer Selektion möglich ist, nicht direkt.

Zusammenfassung

- Ein Union-Objekt enthält genau *ein* Objekt.
- Verschiedene Union-Objekte können verschiedene Datentypen haben;

ein Union-Objekt kann zu verschiedenen Zeiten verschiedene Datentypen haben.

- Der Datentyp eines Union-Objekts kann jederzeit ermittelt werden.
- Um ein Union-Objekt zu verarbeiten, muß der Inhalt *selektiert* werden.

unions Die *unions* in C tragen nicht zufällig den gleichen Namen: hinter ihnen steht das gleiche Konzept, die VDM-Unions können mit Hilfe von *unions* implementiert werden. Einen Vorteil haben die VDM-Unions allerdings — Sie können ihren aktuellen Typ ermitteln, weil er in Form einer Zusatzinformation mitgespeichert wird. Sie können also Fehler, die auf der falschen Interpretation des Inhalts einer *union* beruhen, mit VDM-Unions vermeiden, da die Selektion eines „falschen" Objekts nicht möglich ist.

Operationen Auch für Unions sind nur sehr wenige Operationen erforderlich: Erzeugung eines Union-Objekts, Ermittlung des aktuellen Typs eines Union-Objekts, Selektion des Inhalts, Überschreiben des Inhalts mit einem Wert vom gleichen oder anderen Typ und Test auf Gleichheit zweier Union-Objekte.

Nomenklatur Da Sie wie bei Trees auf verschiedene Datentypen innerhalb eines Objekts zugreifen können (im Unterschied zu Trees allerdings nicht gleichzeitig), können Sie auch für die Datentypen einer Union jeweils einen speziellen Namen festlegen, der den gleichen Konventionen folgt, also mit der Sequenz „**s_**" beginnt und noch mindestens einen weiteren Kleinbuchstaben enthält. Da im Unterschied zu Trees mehrfach auftretende Datentypen sinnlos sind (ein Union-Objekt enthält ja nur ein Objekt...), sind diese Namen nicht Pflicht! Wenn Sie Namen angegeben haben, werden die komponentenbezogenen Operationen wie bei den Trees mit dem variablen Teil des Komponenten-Namens erweitert — im anderen Falle erfolgt die Erweiterung mit dem Namen des jeweiligen Komponententyps. Sie sollten, wenn Sie von der Möglichkeit der Namensgebung Gebrauch machen möchten, sprechende Namen für die Komponenten wählen.

DSL Um in der DSL-Spezifikation eine Union Uni mit den möglichen Datentypen $\mathcal{A}_1, \mathcal{A}_2 \ldots \mathcal{A}_n$ zu definieren, wählen Sie eine der beiden folgenden Möglichkeiten aus:

$$\begin{aligned} Uni &= A_1 \mid A_2 \mid \ldots \mid A_n \qquad \text{oder} \\ Uni &= s\_c1 : A_1 \mid s\_c2 : A_2 \mid \ldots \mid s\_cn : A_n \end{aligned}$$

Anwendung Unions können Sie immer dann einsetzen, wenn Sie wissen, daß ein bestimmtes Objekt eine endliche Anzahl verschiedener Ausprägungen haben kann und Sie alle möglichen Ausprägungen in einer einheitlichen Struktur bearbeiten möchten.

Beispiele Eine Anwendung haben Sie bereits kennengelernt: das Optional ist eine Union des Elementtyps und eines Typs mit dem einzigen Wert nil.

Im Compilerbau benötigen Sie Unions, um die in der Grammatik definierten Alternativen für ein Lexem (Terminal oder Non-Terminal) zu modellieren, die ja in der BNF (Backus-Naur-Form) ebenfalls mit Hilfe der senkrechten Striche ausgedrückt werden. Ein Beispiel:

statement ::= while | if-else | do-while | for | assignment | call | goto

Ein Objekt, das ein Statement modellieren soll, muß alle diese Alternativen aufnehmen können.

Eine weitere Anwendung im Compilerbau ist die Symboltabelle. Die Konstanten eines Programms (als Beispiel) werden mit ihrem Namen und ihrem Wert gespeichert; der Wert kann aber einer der ordinalen Zahlentypen, einer der Gleitkomma-Zahlentypen oder auch ein Pointer sein, deren Repräsentation sich jeweils unterscheidet. Eine Union dieser Typen gibt Auskunft über den Typ und den Wert der Konstanten und kann zusammen mit dem Namen der Konstanten als Tree mit fester Größe modelliert werden. Hier ein Beispiel, das allerdings nur einige der genannten Typen abdeckt:

```
DOMAINS
Type   =  Float | Int | Char
Float  =  ELEMENTARY
Int    =  ELEMENTARY
IMPLEMENTATIONS
Tab    =  union + ...
Float  =  unsigned (CSTRUCT="float")
Int    =  scalar (LB=-32768,UB=32767)
```

5.9.2 Union-Implementationen

Für Unions stehen zwei Implementationen zur Verfügung:

- **union** modelliert eine Union mit einer *C–union*. Die dynamische Speicherallokation ist unnötig und der Zugriff erfolgt direkt, aber Zuweisungen erfordern das Kopieren der *union*. Die Realisierung von rekursiven Unions ist mit dieser Implementation nicht möglich.

- **ptunion** erzeugt eine *C–union* auf dem Heap und liefert einen Pointer zurück. Der Zugriff erfolgt über die Dereferenzierung des Pointers (langsamer als direkter Zugriff), aber eine Zuweisung erfolgt durch Pointer-Zuweisung (schnell, wenn keine „echte“ Kopie benötigt wird, die mit copy / copy1 erzeugt werden muß). Mit „ptunion“ können Sie rekursive Unions realisieren.

5.9.3 Union-Operationen

! → Folgende Operationen erfahren eine Erweiterung mit dem variablen Komponenten-Namen: mk, s, chg, ovwrt und subst.

Zur Veranschaulichung dienen die Variablen t1 und t2 vom oben definierten Typ Type für Konstanten-Beschreibungen in der Symboltabelle eines Compilers.

Union-Konstruktoren

mk Mit **mk** erzeugen Sie ein Union-Objekt aus einem Objekt eines Komponententyps. Daher wird mk mit dem variablen Komponenten-Namen erweitert.
Beispiel — Eintrag einer „int"-Konstante:

```
t1 = mk_int (14);
```

subst **subst** kopiert das Union-Objekt und überschreibt den Inhalt des neuen Union-Objekts mit dem angegebenen Wert, allerdings nur, wenn das Union-Objekt vorher den gleichen Typ hatte.
Beispiel — eine weitere "int"-Konstante soll aufgenommen werden:

```
t2 = mk_int (9);        oder
t2 = subst_int (t1, 9);
```

copy / copy1 Auch ein Union-Objekt können Sie natürlich durch Kopieren eines anderen Union-Objekts erzeugen.

Union-Kombinatoren

Es gibt zwei Klassen von Kombinatoren: die einen ändern den Inhalt des Union-Objekts, aber nicht den Typ, die anderen können auch den Typ verändern. Beiden Arten ist gemeinsam, daß das Union-Objekt überschrieben und der alte Inhalt damit zerstört wird. Sollte das Union-Objekt vor dem Aufruf einer dieser Operationen Sub-Strukturen haben, müssen Sie es eventuell vorher mit del löschen.

Nomenklatur Alle Kombinatoren werden mit dem variablen Teil des Komponenten-Namens erweitert.

chg Mit **chg** überschreiben Sie den Inhalt eines Union-Objekts, ohne dabei dessen Typ ändern zu können; ein solcher Versuch wird mit einem Laufzeitfehler quittiert!
Beispiel — Änderung des Wertes der ersten Konstanten (welch ein Widerspruch in sich...):

```
t1 = chg_int (t1, 14965);
```

ovwrt Mit **ovwrt** überschreiben Sie ein Union-Objekt ohne Rücksicht auf den

vorherigen Inhalt.
Beispiel — die erste Konstante stellt nun eine „Char"-Konstante dar (in dem gewählten Kontext *völlig* sinnlos):

```
t1 = ovwrt_char (t1, ´T´);
```

Union-Prädikate

is_eq Mit **is_eq** testen Sie, ob zwei Unions, deren aktueller Typ gleich ist, den gleichen Inhalt haben.

```
t1 = mk_int (14);   t2 = subst_int (t1, 9);
⟹ is_eq_type (t1, t2) == FALSE
```

is_gr Mit **is_gr** testen Sie zwei Unions des gleichen Typs, ob ihr Inhalt der Relation „göẞer als" genügt. is_gr können Sie nur anfordern, wenn es für alle möglichen Datentypen der Union definiert ist.

is_ge Mit **is_ge** testen Sie, ob für zwei Unions des gleichen Typs eines der beiden obigen Prädikate erfüllt ist. Es gilt die gleiche Restriktion wie für is_gr.

is **is** stellt eine ganze Klasse von Prädikaten dar: sie ermöglichen die Ermittlung des aktuellen Typs eines Union-Objekts. Zu jeder Zeit ist genau eines dieser Prädikate erfüllt. is wird mit dem variablen Teil des Komponenten-Namens erweitert.

```
is_int (t1) == TRUE ⇔ (is_char (t1) == FALSE ∧ is_float (t1) == FALSE)
```

Union-Attribute

Diskriminante Die Information, welchen Typ ein Union-Objekt gerade hat, ist im Objekt in Form eines Datums gespeichert, das einem Aufzählungstyp angehört. Dieser Aufzählungstyp wird implizit mit enum definiert, wenn Sie eine Union spezifizieren; die Elemente heißen **Diskriminanten** der Union, sie stehen für die möglichen Datentypen. Der Name einer Diskriminante ist bis auf den ersten Buchstaben identisch mit dem Selektornamen: das führende „s" wird durch ein „d" ersetzt. Sie können verwendet werden, um in einer switch-Anweisung abhängig vom aktuellen Typ bestimmte Aktionen auszuführen.

discr **discr** liefert die Diskriminante eines Union-Objekts.
Beispiel — Ausgabe des Konstantenwertes mittels spezieller Druckroutinen:

```
switch (discr_type (t1)) {
  case d_int:    print_int (t1); break;
  case d_float:  print_fl (t1); break;
  case d_char:   print_char (t1); break;
}
```

Selektion in einer Union

s Mit **s** greifen Sie auf den Inhalt eines Union-Objekts zu. Sie müssen allerdings die richtige, zum aktuellen Typ passende Version auswählen. Der Name wird jeweils mit dem variablen Teil des Komponenten-Namens erweitert, wodurch sich auch hier — wie bei den Trees — die Identität mit dem Komponenten-Namen ergibt.
Beispiel — die eben eingeführten Druckroutinen müssen den Inhalt des Union-Objekts selektieren:

```
void print_int (Type t)
{ printf("%d", s_int(t)); }
```

Funktionale

?switch Mit den beiden Funktionalen **sswitch** und **cswitch** können Sie sogenannte Switch-Funktionen erzeugen, die abhängig vom aktuellen Typ des Union-Objekts bestimmte Funktionen ausführen, also von der Struktur her ähnlich aufgebaut sind wie das Beispiel für die Operation discr; allerdings wird diesen Verarbeitungsfunktionen bereits der korrekt selektierte Inhalt des Union-Objekts übergeben.

Parameter Sie können den Resultattyp der einzelnen Verarbeitungsfunktionen (und damit auch der Switch-Funktion) mit dem Implementationsparameter **RESULT** selbst bestimmen — wenn Sie nichts angeben, werden Prozeduren vorausgesetzt. Außerdem können Sie jeder Verarbeitungsfunktion weitere Argumente übergeben, deren Datentypen Sie mit den Parametern **XX, XXX, XXXX,** ... festlegen; beim Aufruf der Switch-Funktion müssen Sie diese Argumente dann zusätzlich angeben.

Nomenklatur Die Namen der Verarbeitungsfunktionen entsprechen dem erweiterten Namen der Switch-Funktion. Bei sswitch werden die Namen mit den variablen Teilen der Komponenten-Namen erweitert, bei cswitch mit den Namen der möglichen Datentypen in der Union.

Ein Beispiel soll die Zusammenhänge verdeutlichen: Sie fordern in der DSL-Spezifikation für die Konstanten-Tabelle folgende Funktion an:

```
cswitch.sw (RESULT="Bool",XX="Nat0")
```

Daraus wird die folgende Switch-Funktion generiert:

```
Bool sw_type (Type t,Nat0 n)
{
  switch (discr_type (t)) {
       case d_float:   return sw_float (s_float (t), n);
       case d_int:   return sw_int (s_int (t), n);
       case d_char:   return sw_char (s_char (t), n);
  }
}
```

Die Wirkung von sswitch ist die gleiche, da bei der Definition von Type keine Komponenten-Namen angegeben wurden.

Die Verarbeitungsfunktionen sw_float, sw_int und sw_char mit den entsprechenden Prototypen müssen Sie natürlich definieren; die mühselige Schreibarbeit für die Switch-Funktion bleibt Ihnen jedoch erspart!

6. Rekursive Typdefinition

6.1 Verwaltung von variablen Datenmengen

In den meisten nicht-trivialen Programmen benötigen Sie Datenstrukturen und Operationen zur Verarbeitung von Objekten, deren Anzahl nicht von vornherein feststeht, sondern variabel ist. Zur Organisation dieser variablen Datenmengen gibt es verschiedene Ansätze.

6.1.1 Sequentielle Organisation

Sie können die Objekte z.B. lückenlos hintereinander in den Speicher schreiben — um nun Informationen über die Anzahl der Einzelobjekte zu erhalten, müssen Sie entweder das Ende des Speicherblocks durch eine besondere Marke kennzeichnen oder die Anzahl der Objekte in einer separaten Variablen abspeichern. Die Voraussetzung ist allerdings die Verfügbarkeit von ausreichendem zusammenhängenden Speicherplatz (sogenanntem *linearen* Speicherplatz).

Diese Art der Verwaltung wird für sequentielle Dateien verwendet, da der Platz auf einem Massenspeicher immer als linear angesehen werden kann. Entweder ist die Länge der Datei im Inhaltsverzeichnis des Massenspeichers abgelegt oder ein besonderes Zeichen dient als Ende-Markierung. Ein Beispiel: die frühen Versionen von *MS-DOS* verwendeten das ASCII-Zeichen Nr. 26 — auch „Control-Z“ genannt — als Ende-Kennung für Textdateien; in den neueren Versionen ist für alle Dateien der Längeneintrag im Verzeichnis relevant.

Im Hauptspeicher des Rechners allerdings kann mit dieser Methode nur selten erfolgreich gearbeitet werden, vornehmlich bei kleineren Datenmengen — der verfügbare lineare Speicher ist meist nicht sehr groß. Eine Anwendung sind Zeichenketten, deren Länge nur selten 256 Zeichen übersteigt: in *C* dient das Zeichen mit der Zeichensatz-Nummer 0 als Endekennung, das daher in *C*-Strings nicht vorkommen darf; die *Pascal*-Variante „Turbo-Pascal“ hingegen benutzt das erste Zeichen der Zeichenkette, um die aktuelle Länge zu speichern.

Zu der Schwierigkeit mit dem linearen Speicher kommt ein weiteres Problem: die Reihenfolge der einzelnen Objekte ist festgelegt und jede Umsortierung ist mit großem Aufwand verbunden (wenn man nicht mit zusätzlichen Index-Tabellen arbeitet, die wieder Speicherplatz kosten und die vielleicht auch einmal geändert werden müssen…). Gerade die Sortierung von Datensätzen nach verschiedenen Kriterien ist aber eine sehr oft benötigte Operation; eine andere Form der Datenstrukturierung, die diese Anforderungen besser unterstützt, ist dann notwendig.

6.1.2 Rekursionskonzepte

Eine rekursive Datenstrukturierung weist Eigenschaften auf, die sie der sequentiellen Anordnung überlegen macht. **Rekursion** bedeutet „Rückbezüglichkeit" in dem Sinne, daß eine Klasse (unter anderem) durch sich selbst definiert ist, daß ein Objekt dieser Klasse also weitere, gleichartige Objekte „enthält". Damit die Rekursion nicht unendlich tief ist, muß es allerdings eine abbrechende Alternative geben.

Was bedeutet „ein Objekt enthält ein anderes" ? Eine seiner Komponenten ist in diesem Fall ein *Verweis* auf das andere Objekt, ein „Zeiger", der z.B. in *C* als Variable vom Typ Pointer implementiert ist und darin die Adresse des enthaltenen Objekts speichert.

Daraus ergibt sich aber, daß die Objekte *nicht* sequentiell im Speicher angeordnet zu sein brauchen; die Verweise können sozusagen „kreuz und quer" durch den Speicherraum verlaufen ! Die logische Reihenfolge ist damit von der physikalischen Reihenfolge unabhängig — wir können von einer *abstrakten*, d.h. von der Repräsentation losgelösten Reihenfolge der Objekte sprechen.

Abstraktion

Erstes Konzept

Ein mögliches Konzept ist, daß jedes Objekt zusätzlich zu seinen eigenen Daten eine *feste* Anzahl weiterer, gleichartiger Objekte enthält, so daß ein Objekt also jederzeit von seinem „Vorgänger" aus erreicht werden kann. Ein besonderer Wert zur Anzeige, daß ein Objekt keine weiteren Nachfolger hat, ist hier nötig, genau wie bei sequentiellen Strukturen; dazu kann der Wert nil herangezogen werden, das Objekt wird also als *Optional* modelliert. Neben den Verweisen enthält ein Objekt auch Daten; es ist also ein *Tree*.

Enthält ein Objekt genau *ein* weiteres gleichartiges Objekt, bildet die Gesamtstruktur eine verkettete Liste (wie $\mathcal{A}$ im folgenden Beispiel). Bei n Verweisen (wie $\mathcal{C}$ im folgenden Beispiel) entsteht im allgemeinen eine Baumstruktur mit einem Objekt an der „Spitze" und n Nachfolgeknoten auf jeder Ebene — wenn es keine Rückwärtsbezüge im Baum gibt. Ein typisches Beispiel für solche Rückwärtsbezüge ist die doppelt verkettete Liste: ein Knoten enthält seinen Nachfolger und seinen Vorgänger. Nur wenn *kein* Verweis auf ein hierarchisch höheres Objekt existiert, handelt es sich um einen echten Baum.

A = [B]
B :: s_data : Data   s_next : A
C = [D]
D :: s_data : Data   s_first : C . . . s_last : C

Wie aber können Sie Objekte mit einer variablen Anzahl von Nachfolgern erzeugen ? Man unterscheidet vier Typen, die jeweils zunehmende Flexibilität aufweisen. Bei den ersten drei Typen enthält ein Knoten mit Nachfolgern selbst keine Daten; die Daten sind nur in den sogenannten *Blättern*, das sind Knoten ohne Nachfolger, gespeichert — indem ein Knoten entweder Verweise auf Nachfolgeknoten *oder* Daten enthält. Der vierte Typ entsteht durch Kombination eines dieser Typen mit dem ersten Konzept, das auf Trees basiert.

Zweites Konzept

Mit Sets, die entweder gleichartige Sets oder Datenelemente enthalten, können Sie eine einfache rekursive Struktur aufbauen; die Daten sind allerdings aufgrund der einzig möglichen, zufälligen Element-Selektion nur schlecht gezielt erreichbar, da ein Datum jeweils gesucht werden muß. Diese Struktur eignet sich, wenn in der Regel die gesamte Struktur zu bearbeiten ist.

$\mathsf{E} = \mathsf{F}\text{-}\mathsf{set}$
$\mathsf{F} = \mathsf{E} \mid \mathsf{Data}$
$\mathsf{Data} = \ldots$

Die Daten sind natürlich von beliebigem Typ; häufig handelt es sich um Trees, die ja mehrere Daten zusammenfassen.

Drittes Konzept

Mit Tuples erreichen Sie zusätzlich, daß Sie gezielt auf die Daten zugreifen können, indem Sie jeweils eine Liste von Indizes als Zugriffspfad angeben; die Blätter sind auf diese Weise eindeutig und gezielt erreichbar.

$\mathsf{G} = \mathsf{H}\,\star$
$\mathsf{H} = \mathsf{G} \mid \mathsf{Data}$
$\mathsf{Data} = \ldots$

Viertes Konzept

Wählen Sie statt Tuple bzw. Set die Map, können Sie die Zugriffspfade mit Hilfe beliebiger Datentypen definieren, da die Argumentwerte der Map den Zugriffspfad eindeutig charakterisieren *und* von beliebigem Typ sein können! Mit anderen Worten: Sie können mit Maps auch die Tuple-Rekursion simulieren, indem Sie den Datentyp Nat1 als Argument-Domain wählen; ebensogut sind aber Tuples, Sets, Trees oder wieder Maps als Argumentwerte möglich.

$\mathsf{J} = \mathsf{K} \xrightarrow{m} \mathsf{L}$
$\mathsf{K} = \textit{Zugriffspfad-Typ}$
$\mathsf{L} = \mathsf{J} \mid \mathsf{Data}$
$\mathsf{Data} = \ldots$

Beispiel:

Ein Beispiel haben Sie bereits kennengelernt: das hierarchische Datei-System, in dem die Daten in Dateien (z.B. als Tuples modelliert) am Ende eines Verzeichnispfades stehen. Ein Verzeichnis (Directory) ist eine Map von Strings auf Verzeichniseinträge; die Strings enthalten die Namen der Einträge, die entweder Dateien sind oder wiederum Verzeichnisse. Die inneren Knoten sind zwar durch Strings identifizierbar, enthalten aber keinerlei Daten. Die Zugriffspfade sind durch die Abfolge von Verzeichnisnamen und einem Dateinamen am Ende *eindeutig* definiert. Die Domain-Gleichungen haben folgendes Aussehen:

DOMAINS
Direc = Str $\overrightarrow{m}$ Direntry
Direntry = Datfile | Direc

Fünftes Konzept

Schließlich können Sie auch in den inneren Knoten Daten unterbringen, indem Sie den Elementtyp von Tuples, Sets bzw. Maps anders wählen: statt der *Union*, bestehend aus Daten ODER Verweisen, nehmen Sie den *Tree*, bestehend aus Daten UND Verweisen! Der Tree enthält dann als „Verweis-Komponente“ keinen gleichartigen Tree, sondern wieder ein Tuple, einen Set oder eine Map von Trees. Ein Beispiel:

A = B –set
B :: s_data : Data s_sub : A

Eine Anwendung dafür finden Sie übrigens im Kapitel 7.

Organisation der Knoten und Verweise

Organisation

Wie die Verweise organisiert sind, ist nicht determiniert. Es gibt zahlreiche grundlegende Datenstrukturen für die Verwaltung der Objekte, die mehr oder weniger komplex sind: die einfachen Strukturen haben ein schlechteres Zeitverhalten (für Zugriffe, Sortierung usw.), benötigen aber auch nur wenig Verwaltungsinformation; die komplexen Strukturen basieren auf umfangreichen, raffinierten Verweis-Pfaden, die natürlich mehr Speicherplatz und kompliziertere Algorithmen zur Manipulation der Objekte benötigen, aber einen wesentlich schnelleren Zugriff erlauben. Zur ersten Kategorie zählen z.B. einfach verkettete Listen, zur zweiten die sogenannten B-Bäume, deren Einfüge- und Löschoperationen immer einen balancierten, im Mittel zu 75–80 % ausgelasteten Baum erzeugen.

Einige Datenstrukturen bieten einen Kompromiß zwischen den beiden genannten Extremen und sind für bestimmte Probleme und Konstellationen die ideale Lösung; ein Binärbaum zum Beispiel, den Sie im Abschnitt 5.8.4 kennengelernt haben, benötigt pro Knoten zwei Verweise auf weitere Knoten und bietet eine recht einfache Möglichkeit der Einsortierung neuer Objekte, kann aber leicht „entarten“ — wenn die Objekte vor der Einsortierung bereits sortiert sind, entsteht eine verkettete Liste, weil alle Knoten nur *einen* Nachfolger erhalten. Dabei wird der Speicherplatz für den jeweiligen zweiten Verweis innerhalb eines Objekts verschwendet und die sogenannte „Tiefe“ des Baumes, also die Anzahl von Stufen, um das am weitesten von der Wurzel enfernte Objekt zu erreichen, ist maximal — die Suchzeiten steigen an. Wenn Sie aber geeignete Randbedingungen schaffen, ist der Binärbaum eine effiziente Organisationsform. Seine Domain-Gleichung:

```
Btree  =  [ Node ]
Node   ::  s_lt : Btree   s_data : Data   s_rt : Btree
```

6.2 Implementation rekursiver Strukturen

Die verschiedenen Rekursionskonzepte wurden mit Hilfe von VDM–Domain-Gleichungen illustriert und können natürlich mit VDM realisiert werden. Der große Vorteil ist die Vermeidung des Umgangs mit Pointern auf Objekte, um den Sie in *C* nicht herumkommen und der sehr fehlerträchtig ist. Sie können mit Hilfe von VDM allgemeine Datenorganisationsformen wie z.B. Binärbäume oder auch B-Bäume ohne übermäßigen Testaufwand realisieren und bei Bedarf wiederverwenden, da VDM von Implementations-Details abstrahiert. Da solche Organisationsformen in sehr vielen Anwendungen benötigt werden, sinkt die Erstellungszeit für solche Anwendungen bei Verwendung von VDM deutlich.

Auswahl der Implementationen

Für die an rekursiven Strukturen beteiligten Datentypen können Sie nicht jede Implementation verwenden; weil Verweise auf Nachfolgeknoten benötigt werden, müssen die Implementationen auf Pointern basieren. Ein Beispiel: Bei Auswahl der *struct*-Implementation für den oben angegebenen Binärbaum erzeugt der VDM-Compiler folgende Typdefinition, aus der der C-Compiler keinen Code erzeugen kann, weil die Rekursion nicht auflösbar ist:

```
typedef struct {
        Btree   s_lt;
        Data    s_data;
        Btree   s_rt;
} Btree;
```

Wählen Sie stattdessen die *ptstruct*-Implementation, die auf Pointern basiert, kann ein korrekter *C*-Quelltext generiert werden:

```
typedef Struct_btree * Btree;

typedef struct {
        Btree   s_lt;
        Data    s_data;
        Btree   s_rt;
} Struct_btree;
```

Mehrstufige Rekursion

Meist benötigen Sie zwei „über Kreuz“ rekursive Strukturen oder sogar mehrere Strukturen, die einen Zyklus von Verweisen enthalten, ohne daß eine der Einzelstrukturen rekursiv ist, weil die DSL-Syntax verlangt, daß alle verschachtelten Datentypen in einzelne Definitionen zerlegt werden. In diesem

Fall muß mindestens *einer* der beteiligten Datentypen mit Hilfe von Pointern implementiert werden. Ein Beispiel dafür sind die beiden Datentypen Btree und Node für Binärbäume.

Der VDM-Compiler kann feststellen, ob die Rekursion auflösbar ist oder nicht, indem er die gewählten Implementationen analysiert und die Typdefinitionen in einer auflösbaren Reihenfolge anordnet, wenn er eine Pointer-Implementation findet. Gelingt ihm das nicht, bricht er mit folgender Fehlermeldung ab: **dependency cycle detected**

6.3 Beispiel: Syntaxdefinition von Programmiersprachen

Spezifikationen von Programmiersprachen, die mit Hilfe von kontextfreien Grammatiken erstellt werden, machen regen Gebrauch von Rekursion — das beste Beispiel ist die Definition einer Anweisung in C, die hier mit Hilfe von VDM modelliert ist:

```
DOMAINS
Stmt      =   Expr | Return | Break | Cont | Goto | Label |
              If | While | Switch | Do | For | Block | Nullstmt
Return    =   Optexpr
Break     =   ELEMENTARY
Cont      =   ELEMENTARY
Goto      ::  s_gotoid : Id   s_gotostmt : Stmt
Label     ::  s_labelid : Id   s_labelstmt : Stmt
If        ::  s_ifexpr : Expr   s_ifthen : Stmt s_ifelse : Optstmt
While     ::  s_whileexpr : Expr   s_whilestmt : Stmt
Switch    ::  s_swexpr : Expr   s_swcase : Case
Do        ::  s_dostmt : Stmt   s_doexpr : Expr
For       ::  s_forinit : Optexpr s_forexit : Optexpr
              s_forincr : Optexpr s_forstmt : Stmt
Block     ::  s_blkhd : Blkhead   s_blkstmts : Stmts
Nullstmt  =   ELEMENTARY
Optexpr   =   [ Expr ]
Optstmt   =   [ Stmt ]
Blkhead   =   [ Decls ]
Decls     ::  s_decl : Decl   s_blkhead : Blkhead
Decl      =   Typedef | Extern | Localdata
Stmts     =   [ Single ]
Single    ::  s_sstmt : Stmt   s_single : Stmts
```

Einige Anmerkungen dazu:

Nomenklatur

Die Abkürzungen beziehen sich auf die üblichen englischen Bezeichnungen: *statement* (Anweisung), *expression* (Ausdruck), *identifier* (Bezeichner).

Case / Expr Die beiden Non-Terminale Case und Expr sind hier nicht weiter aufgelöst.

Terminale Die Terminalsymbole von C sind für die Modellierung eines Programms mit einem abstrakten Datentyp nicht wichtig, wenn der Quelltext richtig analysiert wurde (ist z.B. ein Stück des Quelltexts als for-Statement erkannt worden und wurden die maximal vier Bestandteile korrekt in einer Variablen vom Typ For untergebracht, ist der identifizierende Text for mit Klammern und Semikola nicht mehr erforderlich).

Return Die **return**-Anweisung hat ein optionales Argument, den Rückgabewert, der aus dem angegebenen Ausdruck berechnet wird.

ELEMENTARY Die Vereinbarung einiger ADT als „ELEMENTARY" soll anzeigen, daß keine weitere Information zur Interpretation der Anweisung nötig ist (z.B. wird der Befehl break durch eine Variable repräsentiert, deren Anwesenheit dem Code-Generator anzeigt, daß der break-Befehl im Programm stand; das reicht aus, um den Quelltext richtig zu interpretieren).

for Alle Ausdrücke innerhalb der **for**-Schleife (Initialisierung, Abbruchbedingung, Inkrementierung) sind optional.

Block Ein Block beginnt mit einer optionalen, beliebig langen Liste (Blkhead) von Typdefinitionen, Deklarationen von externen Variablen und Definition von lokalen Variablen. Danach folgt eine optionale, beliebig lange Sequenz von Anweisungen, die auch leer sein kann; beide Bestandteile wurden durch eine verkettete Liste rekursiv definiert.

7. Ein größeres Beispiel

Nachdem Sie nun die einzelnen VDM-Basistypen mit kleinen Beispielen für ihre Anwendung kennengelernt haben, möchten Sie vielleicht einmal ein größeres Beispiel sehen, in dem die ADT „zusammenspielen".

Aufgabe

Wir wollen aus der in Abschnitt 5.6.4 vorgestellten qualitativen Stückliste eine **quantitative Stückliste** machen. Wir benötigen folgende Funktionen:

- Laden und Speichern der Stückliste
- Aufnahme einer neuen Teil-Beschreibung in die Stückliste (Erweiterung)
- Änderung einer vorhandenen Teil-Beschreibung
- Ausgabe der sogenannten „Struktur-Stückliste" für ein Teil

Struktur-Stückliste

Eine *Struktur-Stückliste* stellt die Zusammensetzung eines Teils in einer hierarchischen Struktur dar, jeweils mit der Anzahl benötigter Teile — sie wird in der Fertigung benötigt. Im Gegensatz dazu zeigt die *Mengenübersichts-Stückliste* nur an, welche Teile wie häufig benötigt werden und ist damit für den Einkauf geeignet; kommt ein Teil an verschiedenen Stellen vor, werden die Einzelhäufigkeiten kumuliert und gesammelt ausgegeben.

7.1 Quantitative Stückliste — das Datenmodell

„Bag"

Wie modellieren wir eine solche Stückliste ? Wir erweitern die qualitative Beschreibung, indem wir zu jedem Bestandteil eines Teils auch seine Anzahl speichern. Dies gelingt uns mit einer sogenannten **Bag**, das ist ein Set, in dem ein Element mehrfach vorkommen darf. Wir ordnen also jedem Teil im Set seine Häufigkeit zu, indem wir eine Map von Teilnummern (ganze Zahlen größer als 0) auf Häufigkeiten definieren:

$$\begin{aligned}
\mathsf{Plist} &= \mathsf{Nat1} \ \xrightarrow[m]{} \ \mathsf{Pnbag} \\
\mathsf{Pnbag} &= \mathsf{Nat1} \ \xrightarrow[m]{} \ \mathsf{Nat1} \\
\mathsf{Pnset} &= \mathsf{Nat1}\text{-}\mathsf{set}
\end{aligned}$$

Die Stückliste Plist (Part list) ordnet jedem Teil eine Bag seiner Bestandteile zu, die wiederum jedem Bestandteil die benötigte Anzahl zuordnet. Die Argumentmenge der Bag ist identisch mit dem Bestandteil-Set der qualitativen

Stückliste; um sie von der Häufigkeitsinformation zu separieren, führen wir auch wieder den Datentyp Pnset ein.

Speicherung

Wir wollen die Stückliste zum Abspeichern in ein *Tuple* umwandeln; uns kommt dabei gelegen, daß die Teilnummern und die Häufigkeiten den gleichen Datentyp haben, nämlich Nat1 . Für ein Tuple mit unstrukturierten Elementen (z.B natürliche Zahlen) gibt es ja Lade- und Speicher-Operationen, nämlich fread und fwrite. Die Speicher-Form der Stückliste modellieren wir:

$$\mathsf{Pntup} \quad = \quad \mathsf{Nat0} \star$$

Hier wählen wir (überraschenderweise) den Datentyp Nat0 als Elementtyp; das hat Vorteile für Speichern und Laden, wie Sie dort sehen werden!

Struktur...

Die Struktur-Stückliste Slist (Structured part list) ist ein Hierarchiebaum, dessen Knoten beliebig viele Nachfolger haben können, und die jeweilige Teilnummer und seine Häufigkeit enthalten. Also benötigen wir eine rekursive Struktur. Da ein Knoten jeweils durch die Teilnummer „charakterisiert“ wird, modellieren wir die Bestandteile eines Teils als Map von Teilnummern auf deren Daten; diese Daten sind die Häufigkeit des Bestandteils im Gesamt-Teil und eine eventuell leere Map von untergeordneten Bestandteilen:

$$\begin{array}{lll} \mathsf{Slist} & = & \mathsf{Nat1} \ \overrightarrow{m} \ \mathsf{Pnrec} \\ \mathsf{Pnrec} & :: & \mathsf{s\_times}: \ \mathsf{Nat1} \\ & & \mathsf{s\_parts}: \ \mathsf{Slist} \end{array}$$

Implementation

Für die diversen Maps kann jeweils die Implementation *scaldom* gewählt werden, da der Argumenttyp ein Ordinaltyp ist; dann muß der Tree allerdings (wegen der Rekursion) auf Pointern basierend mit *ptstruct* implementiert werden. Die Implementation des Tuple durch *array* ist schnell; für Sets gibt es nur die *linklist*.

7.2 Die semantischen Funktionen

7.2.1 Laden und Speichern

Wir müssen eine geeignete Tuple-Repräsentation für die Stückliste finden, um die vorhandenen Operationen fread und fwrite verwenden zu können; für jedes beschriebene Teil in der Stückliste (also in deren Argumentmenge) erzeugen wir daher den folgenden, variabel langen Datensatz:

- Die Teilnummer aus **dom** plist
- Für jedes Bestandteil in der zugeordneten Bag:
  - Teilnummer (aus **dom** pnbag)
  - Häufigkeit

- *ZERO* als Ende-Kennung des Datensatzes — die als Teilnummer verbotene Null (deshalb muß das Tuple Nat0 als Elementtyp haben!)

! → Wir könnten die Teilnummer auch als Optional modellieren und das Ende mit nil markieren. Wir verwenden hier nur deshalb direkt den für Nat1 ungültigen Wert, damit das Beispiel nicht den Rahmen sprengt — Sie sollten aber in einer vergleichbaren Situation an die Optionals denken!

read_plist Die Funktion **read_plist** soll ein Tuple aus einer Datei lesen und in das Plist-Format konvertieren. Dazu wird das Tuple sequentiell verarbeitet: das erste Element eines Datensatzes ist die Nummer des zu beschreibenden Teils *pn*, sie wird entnommen (mit sel_pntup) und eine (zunächst leere) Bag der Bestandteile erzeugt (mk_empty_pnbag). Nun wird das nächste Element aus dem Tuple entnommen; wenn es ungleich ZERO ist, ist es ein Bestandteil *sub_pn* von *pn* und das nächste Element gibt seine Häufigkeit an. Jedes gefundene Paar wird der Bag hinzugefügt (union1_pnbag), bis schließlich der Datensatz mit ZERO endet; dann wird *pn* mit seiner Bag in die Stückliste eingefügt (mit union1_plist). Sind noch weitere Daten im Tuple, wird das nächste Teil *pn* bearbeitet (diese Prüfung muß für die korrekte Bearbeitung des leeren Tuple jeweils am Anfang erfolgen). Ist die Stückliste komplett, wird das Tuple wieder gelöscht (del_pntup).

```
Plist read_plist (Str datafile)
{
  Pntup flist;
  Pnbag pbag;
  Nat1 pn;
  Nat0 sub_pn;
  Nat1 length;
  Nat1 i = 1;
  Plist plist = mk_empty_plist ();

  flist = fread_pntup (datafile);
  length = len_pntup (flist);
  while (i <= length) {
      pn = sel_pntup (flist, i++);
      sub_pn = sel_pntup (flist, i++);
      pbag = mk_empty_pnbag ();
```

```
        while (sub_pn != ZERO) {
            pbag = union1_pnbag (pbag, sub_pn, sel_pntup (flist, i++));
            sub_pn = sel_pntup (flist, i++);
        }
        plist = union1_plist (plist, pn, pbag);
    }
    del_pntup (flist);
    return plist;
}
```

Wenn der Lesevorgang mit fread nicht erfolgreich war, wird eine leere Stückliste zurückgegeben. ! → Diese Funktion ist ein *Konstruktor* — eventuell müssen Sie also vor dem Aufruf die Stückliste, der Sie das Resultat dieser Funktion zuweisen wollen, explizit löschen!

write_plist Die Funktion **write_plist** soll ein ebenes Tuple aus der strukturierten Stückliste erzeugen und diese in eine Datei schreiben. Ablauf: ein Teil *pn* aus der Stückliste wird gewählt (sel_dom_plist) und dem Tuple mit appr_pntup angehängt. Dann wird die zugehörige Bag mit sel_plist ermittelt und das Teil aus der Stückliste entfernt (sub1_plist); in einer Schleife werden der Reihe nach die Bestandteile *sub_pn* der Bag entnommen (auf die gleiche Weise wie bei den Teilen *pn*) und ihre Nummern gemeinsam mit ihren Häufigkeiten dem Tuple angehängt (mit conc_pntup). Ist die Bag leer, wird dem Tuple die Ende-Kennung des Datensatzes (ZERO) angehängt und der Ablauf wiederholt sich, wenn die Stückliste noch Teile enthält (damit eine leere Stückliste korrekt bearbeitet wird, muß diese Prüfung zu Beginn erfolgen). Schließlich wird das fertige Tuple mit fwrite_pntup in die Datei geschrieben. Da die Stückliste bei diesem Vorgang zerstört wird, wird vorher eine „Arbeitskopie“ angelegt, ebenso von jeder Bag, die der Stückliste entnommen wird. Das Funktionsergebnis entspricht dem Regebnis von fwrite.

```
Bool write_plist (Plist plist, Str datafile)
{
    Plist plwork = copy1_plist (plist);
    Pntup flist = mk_empty_pntup ();
    Pnbag pbag;
    Nat1 pn, sub_pn, count;
    Bool result;

    while (! is_empty_plist (plwork)) {
        pn = sel_dom_plist (plwork);
        pbag = copy1_pnbag (sel_plist (plwork, pn));
        plwork = sub1_plist (plwork, pn);
        flist = appr_pntup (flist, pn);
```

```
        while (! is_empty_pnbag (pbag)) {
            sub_pn = sel_dom_pnbag (pbag);
            count = sel_pnbag (pbag, sub_pn);
            pbag = sub1_pnbag (pbag, sub_pn);
            flist = conc_pntup (flist, mk2_pntup (sub_pn, count));
        }
        flist = appr_pntup (flist, ZERO);
    }
    result = fwrite_pntup (flist, datafile);
    del_pntup (flist);
    return result;
}
```

7.2.2 Eingabe einer Teil-Beschreibung

Die Eingabe unterteilt sich in folgende Abschnitte:

1. Eingabe des zu beschreibenden Teils — ist eine Teilbeschreibung schon vorhanden, wird gefragt, ob diese geändert werden soll
2. Eingabe der Liste der Bestandteile, für jedes Teil folgende Schritte:
   - Eingabe der Nummer eines Bestandteils
   - Eingabeprüfung — ist eine der folgenden Bedingungen nicht erfüllt, muß die Bestandteilnummer erneut eingegeben werden:
     - Ist es bereits in der Stückliste beschrieben ?
     - Ist es in *dieser* Teilbeschreibung noch *nicht* vorhanden ?
     - Ist das zu beschreibende Teil *nicht* zur Erstellung dieses Teils notwendig (dann ergäbe sich eine rekursive Beschreibung) ?
   - Eingabe der Häufigkeit dieses Teils
3. Wenn alles korrekt war: Eintrag der kompletten Beschreibung in die Stückliste (wird ein Teil re-definiert, muß der vorherige Eintrag gelöscht werden)

Wie beendet man die Eingabe der Bestandteil-Liste ? Hier nutzen wir wieder die ungültige Teilnummer ZERO, die uns beim Laden und Speichern schon geholfen hat. Außerdem ist es wünschenswert, die Eingabe der Beschreibung abbrechen zu können, wenn ein Bestandteil noch nicht in der Stückliste enthalten ist — vielleicht möchte man es zunächst eingeben.

new_part Die Funktion **new_part** gliedert sich entsprechend der oben angegebenen Abschnitte der Eingabe. Zunächst wird eine Teilnummer *pn* angefordert und mit is_in_plist geprüft, ob das Teil bereits beschrieben ist; wenn ja, muß die

Neueingabe einer Beschreibung bestätigt werden. Dieser Ablauf wiederholt sich solange, bis entweder die Nummer ZERO für den Abbruch der Eingabe oder ein gültiges, zu beschreibendes Teil eingegeben wurde. Wurde ZERO eingegeben, wird die Funktion mit unveränderter Stückliste verlassen.

Nun wird eine leere Bag zur Aufnahme der Bestandteile erzeugt (mk_empty_pnbag). In einer Schleife werden nun Bestandteilnummer *sub_pn* und Häufigkeit eingegeben, bis die Eingabe mit der Nummer ZERO beendet oder nach Eingabe eines unbekannten Teils abgebrochen wird. Das Flag exist gibt an, ob das Teil in der Stückliste beschrieben ist (is_in_plist); wenn nicht, wird gefragt, ob die Eingabe abgebrochen werden soll und bei positiver Anwort die Funktion mit unveränderter Stückliste verlassen. Ist das Teil aber vorhanden, wird mit is_in_pnbag geprüft, ob es bereits als Bestandteil von *pn* eingegeben wurde. Wenn nicht, muß nur noch geprüft werden, ob es das zu beschreibende Teil *pn* selbst als Bestandteil hat — dazu wird mit der Hilfsfunktion parts_pn ein Set sämtlicher Bestandteile von *sub_pn* ermittelt und mit is_in_pnset geprüft, ob das Teil *pn* nicht in diesem Set enthalten ist. War *sub_pn* ungültig, muß die Eingabe wiederholt werden. Wurde schließlich eine gültige Nummer eingegeben (also auch nicht ZERO zum ordnungsgemäßen Ende der Eingabe), wird deren Häufigkeit erfragt, die natürlich mindestens 1 sein muß. Das vollständige Paar wird nun mit union1_pnbag der Bag von Bestandteilen hinzugefügt.

Nach erfolgreicher Beendigung der Eingabe wird die Stückliste geändert: ist das Teil *pn* neu (wird es also nicht re-definiert), kann das Paar *pn* und Bag mit union1_plist in die Stückliste eingetragen werden; sonst muß der bestehende Eintrag überschrieben und dabei gleichzeitig gelöscht werden — das macht ovwrt1d_plist. Die Stückliste wird nun zurückgeliefert.

Auf die Prüfung der Benutzereingaben wird hier übrigens verzichtet, damit die Funktion nicht den Rahmen sprengt.

```
Plist new_part (Plist plist)
{
  Pnbag pbag;
  Pnset ps;
  Bool ovwrt, exist, twice, recursive;
  int c;
  Nat0 pn, sub_pn, count;
```

```
do {
    puts ("Neue Teilnummer (Abbruch mit 0):");
    scanf ("%ld", &pn);
    ovwrt = is_in_plist (pn, plist);
    if (ovwrt) {
        puts ("Teil ist vorhanden - neu definieren ?");
        c = toupper (getchar ());
    }
    else
        c = 'J';
} while (c != 'J');
if (pn == ZERO)
    return plist;

pbag = mk_empty_pnbag ();
do {
    do {
        printf ("Bestandteil von %ld (Ende mit 0):\n", pn);
        scanf ("%ld", &sub_pn);
        exist = sub_pn == ZERO || is_in_plist (sub_pn, plist);
        brk = twice = recursive = FALSE;
        if (sub_pn != ZERO)
            continue;           /* Vorzeitiges Ende der Schleife */
        if (! exist) {
            printf ("Teil %ld nicht definiert.\n", sub_pn);
            printf ("Eingabe fuer %ld abbrechen ?\n", pn);
            c = toupper (getchar ());
            if (c == 'J') {
                del_pnbag (pbag);
                return plist;
            }
        }
        else {
            twice = is_in_pnbag (sub_pn, pbag);
            if (twice)
                puts ("Teil ist schon angegeben.");
            else {
                ps = parts_pn (plist, sub_pn);
                recursive = is_in_pnset (pn, ps);
                del_pnset (ps);
                if (recursive)
                    puts ("Rekursive Beschreibung.");
            }
        }
    } while (! exist || twice || recursive);
```

```
            if (sub_pn != ZERO) {
                do {
                    printf("Haeufigkeit des Teils %ld:\n", sub_pn);
                    scanf("%ld", &count);
                    if (count < ONE)
                        puts("Haeufigkeit bitte groesser als 0 !");
                } while (count < ONE);
                pbag = union1_pnbag(pbag, sub_pn, count);
            }
        } while (sub_pn != ZERO);
        if (ovwrt)
            plist = ovwrt1d_plist(plist, pn, pbag);
        else
            plist = union1_plist(plist, pn, pbag);
        return plist;
    }
```

parts_pn Die Hilfsfunktion **parts_pn** liefert, wie Sie es schon von der qualitativen Stückliste her kennen, den Set *aller* Bestandteile eines Teils — dazu wird der Set aller direkten Bestandteile ermittelt und damit die Funktion **parts_pnset** aufgerufen, die zu einem Set den Set aller Bestandteile ermittelt. Die Funktion

→ 100

parts_pnset finden Sie auf S. 100, während parts_pn etwas anders aussieht als ihr Pendant: damit erkannt wird, ob *pn* direkt durch sich selbst definiert wird, wird das Teil in einen ein-elementigen Set umgewandelt; dann enthält der Set aller Bestandteile von *sub_pn* auch *sub_pn* selbst und ein separater Test auf Gleichheit von *pn* und *sub_pn* kann entfallen.

```
Pnset parts_pn (Plist plist, Nat1 pn)
{ return parts_pnset(plist, mk_one_pnset(pn)); }
```

Die von parts_pnset benötigte Hilfsfunktion subparts_pnset können wir mit Hilfe von fold für Sets auf die gleiche Weise realisieren wie für eine qualitative

→ 101

Stückliste — sie finden diese Funktion auf S. 101. Ein wichtiger Unterschied betrifft allerdings die Konvertierungsfunktion, die ja keine qualitative Stückliste als Argument erhält und demzufolge den Set der direkten Bestandteile eines Teils auf eine andere Weise ermitteln muß! Da der Set der direkten Bestandteile die Argumentmenge der zugeordneten Bag ist, brauchen wir eine Funktion dom_pnbag, die uns diese Argumentmenge liefert. Wie auf S. 96 bereits gesehen, ist sie mit Hilfe von fold für Maps realisierbar:

```
IMPLEMENTATIONS
Pnbag  =  ...
          + fold.dom (RESULT="Pnset",
                      CVFCT="arg",
                      NULLFCT="mk_empty",
                      FOLDFCT="union")
```

mit der Konvertierungsfunktion
Pnset arg_nat1_nat1 (Nat1 pn, Nat1 sub_pn)
{ return mk_one_pnset (pn); }

Dann sieht die Konvertierungsfunktion für subparts_pnset so aus:

Pnset get_nat1 (Nat1 pn, Plist plist)
{ return dom_pnbag (copy1_pnbag (sel_plist (plist, pn))); }

7.2.3 Struktur-Stückliste

Die Struktur-Stückliste für ein Teil ist rekursiv und muß daher durch eine rekursive semantische Funktion aufgebaut werden. Für Testzwecke kann ihre Ausgabe auf dem Bildschirm mit Hilfe der Operation pr_slist erfolgen, so daß wir uns auf ihre Konstruktion beschränken wollen.

Beispiel: Das Teil #10 besteht aus zwei Teilen #14 und einem Teil #3, welches unzerlegbar ist; das Teil #14 wiederum enthält einmal Teil #3, sechsmal das atomare Teil #31 und dreimal das Teil #9, welches wiederum viermal das Teil #31 enthält. Die Struktur-Stückliste hat dann folgendes schematisches Aussehen:

			Teil #10	
	#14	(2x)		#3 (1x)
#3 (2x)	#9	(6x)	#31(12x)	
	#31 (24x)			

Auf jeder Stufe soll also die Anzahl der benötigten Teile angegeben sein, die von der Anzahl übergeordneter Teile abhängt; diese Angabe muß der Funktion als Argument übergeben werden. Das zu strukturierende Teil wird *einmal* benötigt, also ist das Argument beim erstmaligen Aufruf gleich 1. Außerdem wird für das zu strukturierende Teil auf jeder Stufe die Bag der Bestandteile ermittelt und ebenfalls an die Funktion übergeben. Damit diese Bags beim rekursiven Aufruf auch zugreifbar sind, ist die Stückliste ein weiteres, allerdings unveränderliches Argument.

build_slist Diese Funktion heißt **build_slist** und führt, solange die als Argument erhaltene Bag der Bestandteile nicht leer ist, folgende Schritte aus:

- Entnahme eines Bestandteils aus der Bag (sel_dom_pnbag)
- Ermittlung der zugehörigen Häufigkeit (sel_pnpag)
- Löschen dieses Paares aus der Bag (sub1_pnbag)
- Ermittlung der zu diesem Bestandteil gehörigen „Sub-Bag“ aus der Stückliste (sel_plist) — dabei Erstellung einer Kopie, weil build_slist die ihr übergebene Bag zerstört

- Konstruktion eines Knotens:
  - Gesamthäufigkeit als Produkt aus Häufigkeit des Teils und Häufigkeit des übergordneten Teils
  - Strukturierte Liste der untergeordneten Bestandteile (mit build_slist aus der „Sub-Bag“ und der Gesamthäufigkeit des Teils erstellt)
- Einfügen des Knotens in die Struktur-Stückliste (union1_slist)

Zu Beginn der Funktion wird eine lokale Struktur-Stückliste im leeren Zustand erzeugt (mk_empty_slist), die dann als Resultat zurückgeliefert wird.

```
Slist build_slist (Pnbag pbag, Nat1 times, Plist plist)
{
  Slist slist = mk_empty_slist ();
  Pnrec prec;
  Pnbag sub_bag;
  Nat1 sub_pn, count;

  while (! is_empty_pnbag (pbag)) {
      sub_pn = sel_dom_pnbag (pbag);
      count = times * sel_pnbag (pbag, sub_pn);
      pbag = sub1_pnbag (pbag, sub_pn);
      sub_bag = copy1_pnbag (sel_plist (plist, sub_pn));
      prec = mk_pnrec (count, build_slist (sub_bag, count, plist));
      slist = union1_slist (slist, sub_pn, prec);
  }
  return slist;
}
```

7.3 Das Steuerprogramm

Ein Steuerprogramm ist trivial zu erstellen, daher wollen wir hier nur die Struktur grob umreißen.

- Initialisierung einer Stückliste
- Schleife:
  - Anzeige des Menüs der vier Funktionen
  - Auswahl einer Funktion
  - Aufruf der jeweiligen semantischen Funktion
  - Ausgabe einer Meldung über den Erfolg der Aktion
- Beendigung des Programms nach Rückfrage, ob der neueste Stand der Stückliste abgespeichert werden soll

Natürlich gibt es zahlreiche Erweiterungsmöglichkeiten:

- Variable Dateinamen für Stücklisten, um mehrere Stücklisten verwalten zu können (z.B. in einer Map von Strings auf Tuples)
- Formatierte Ausgabe der Struktur-Stückliste
- Formatierte Ausgabe der gesamten Stückliste
- Erstellung einer Mengenübersichts-Stückliste

8. Online-Hilfe mit *vdminfo*

Wie Sie bereits in Abschnitt 3.2.4 erfahren haben, sind die VDM-Basistypen zusammen mit ihrer Dokumentation in der VDM-Bibliothek gespeichert. *vdminfo* ist ein kommandozeilengesteuertes Anfrageprogramm, mit dem Sie auf die Dokumentation zugreifen können, indem Sie es mit entsprechend gewählten Parametern aufrufen, Sie bekommen also Hilfe während der Arbeit am Rechner. Wie Sie mit *vdminfo* arbeiten, beschreibt dieses Kapitel.

8.1 Die Struktur der VDM-Bibliothek

Um die Benutzung von *vdminfo* verstehen zu können, sollten Sie die grundsätzliche Struktur der VDM-Bibliothek kennen; sie ist in Abb. 8.1 dargestellt.

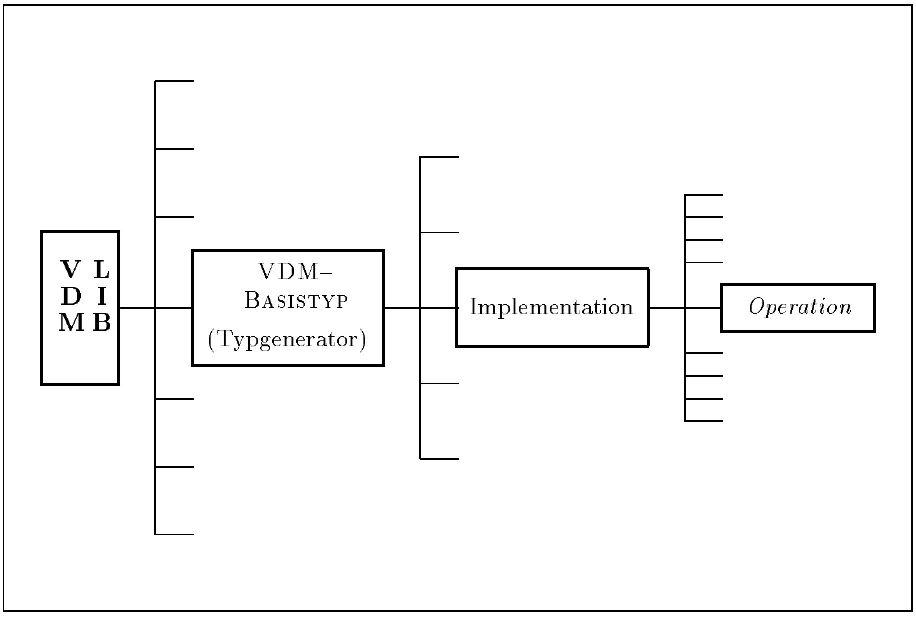

Abbildung 8.1: Struktur der VDM-Bibliothek

In der VDM-Bibliothek sind einige grundlegende Basistypen enthalten, die jeweils in mindestens einer Implementation vorliegen. Implementationen sind grundlegende Datenstrukturen, die den ADT realisieren (z.B. die verkettete Liste); dazu gehören einerseits alle Verwaltungsfunktionen für diese Datenstruktur — mit denen Sie *nie* direkt in Berührung kommen —, andererseits die mit Hilfe dieser Verwaltungsfunktionen codierten, zum Basistyp gehörenden Operationen.

Wie der Abbildung zu entnehmen ist, ist die VDM-Bibliothek hierarchisch aufgebaut. Um Informationen über eine Implementation zu erhalten, müssen Sie daher angeben, auf welchen Basistyp Sie sich beziehen — brauchen Sie eine Angabe über eine Operation, müssen Sie zusätzlich noch die gewünschte

! $\rightarrow$ Implementation nennen. Ein Hinweis: wie bereits erwähnt, soll *grundsätzlich* jede Implementation alle Operationen bereitstellen; es ist also meistens egal, welche Implementation Sie dabei angeben. Allerdings sind einige Operationen in bestimmten Implementationen sehr effizient realisiert, während sie in anderen Implementationen nur in einer generellen, nicht besonders schnellen Version vorliegen. Wenn Sie detaillierte Informationen über eine Implementation anfordern, erhalten Sie eine Kurzbeschreibung aller Operationen, gegliedert nach speziellen und generellen Realisierungen.

Die wenigen Operationen, die nicht in allen Implementationen vorhanden sind, sind bei den jeweiligen VDM-Basistypen im Kapitel 5 und im Anhang B angegeben.

8.2 Der Aufruf von *vdminfo*

Format: *vdminfo* hat folgendes Befehlsformat:

vdminfo [Optionen] [ADT [Implementation [Operation]]]

Die VDM-Basistypen haben innerhalb der Bibliothek abgekürzte Bezeichnungen mit maximal vier Zeichen. Es stehen momentan die in Abb. 8.2 angegebenen Basistypen zur Verfügung; die jeweiligen Implementationen sind ebenfalls angeführt.

Optionen Die Optionen von *vdminfo* haben folgende Bedeutung:

–e : Ausgabe der vordefinierten Basistypen
–d : detaillierte Dokumentation
–c : *C*-Funktionsprototypen der Operation

Option –e Das VDM-System kennt einige vordefinierte Datentypen, die Sie auch schon kennengelernt haben: Nat0, Nat1, Intg, Bool, Char, Str. Eine DSL-Spezifikation dieser vordefinierten Typen erhalten Sie mit der Option **–e**.

Option –d Die Option **–d**, allein mit einem VDM-Basistyp angegeben, liefert Ihnen die verfügbaren Implementationen mit deren Parametern. Bei zusätzlicher Angabe einer Implementation erhalten Sie eine Liste aller Operationen, die damit verfügbar sind, inklusive den Funktionsspezifikationen.

Beispiel: Sie möchten die verfügbaren Implementationen des ADT „Tree“ wissen:

vdminfo –d tree

liefert

implementations for constructor „tree“

implementation „ptstruct“:

implementation „struct“:

Implementation of trees as records (Pascal) or structures (C)
No dynamic data allocation on a heap is required, but
assignments require copying of the whole structure.

Option **–c** Bei detaillierter Ausgabe der Operationen einer Implementation (mit Option –d) ist die VDM-übliche Notation angegeben. Die Option **–c** bei Angabe einer speziellen Operation liefert den entsprechenden Prototyp der *C*-Funktion, mit der diese Operation realisiert ist.

Beispiel: **vdminfo –d** set linklist

liefert für die Operation is_eq:

is_eq (x1,x2) = (x1 = x2)

TYPE: Domain Domain --> Bool

vdminfo –c set linklist is_eq

liefert:

Bool is_eq (Domain x1, Domain x2)

Domain Der Name „Domain“ steht für den Namen des selbstdefinierten abstrakten Datentyps; seine Domain-Gleichung würde also im Falle eines „Set of Element“ lauten:

Domain = Element –set.

KURZNAME	VDM-BASISTYP	IMPLEMENTATIONEN
elem	Elementare Datentypen	bool ptr scalar unsigned
set	Set (Menge)	linklist
tupl	Tuple (Liste)	array linklist
map	Map (Abbildung)	dir heap heapmap linklist scaldom
tree	Tree (Verbund)	ptstruct struct
unio	Union (Variante)	ptunion union
opti	Optional	elemonil ptelem

Abbildung 8.2: ADT in der VDM-Bibliothek: Namen, Implementationen

Anhang A. DSL-Syntax

Die Syntax einer DSL-Spezifikation wird in der Backus-Naur-Form definiert. Terminalsymbole sind umrahmt. Die geschweiften Klammern bedeuten das
! → **ein**- bis **n**-malige Auftreten des geklammerten Ausdrucks.

DSL-Specification ::=
$\boxed{\text{DOMAINS}}$ { Domain-Equation }
$\boxed{\text{IMPLEMENTATIONS}}$ { Impl-Description }

Domain-Equation ::= Tree-Def | Other-Def

Tree-Def ::= Domain-Name $\boxed{::}$ { Selector-Name $\boxed{:}$ Domain-Name }

Other-Def ::= Domain-Name $\boxed{=}$
Set-Dom | Tuple-Dom |
Map-Dom | Union-Dom |
Optional-Dom | Token-Dom

Set-Dom ::= Domain-Name $\boxed{\text{–set}}$

Tuple-Dom ::= Domain-Name ($\boxed{\star}$ | $\boxed{+}$)

Map-Dom ::= Domain-Name $\boxed{\texttt{-m->}}$ Domain-Name

Union-Dom ::= [Selector-Name $\boxed{:}$] Domain-Name
{ $\boxed{|}$ [Selector-Name $\boxed{:}$] Domain-Name }

Optional-Dom ::= $\boxed{[}$ Domain-Name $\boxed{]}$ }

Token-Dom ::= $\boxed{\text{ELEMENTARY}}$

Impl-Description ::= Domain-Name $\boxed{=}$
[Implementation [$\boxed{(}$ Parameter $\boxed{)}$]]
{ $\boxed{+}$ Enrichment [$\boxed{(}$ Parameter $\boxed{)}$]}

Parameter ::=
Param-Name [$\boxed{=}$ Param-Value] [$\boxed{,}$ Parameter]

Param-Value ::= $\boxed{\text{"}}$ String $\boxed{\text{"}}$ | Number | $\boxed{0}$ | Name

Domain-Name ::= Uppercase { Cont-Char }

Selector-Name ::= $\boxed{\text{s\_}}$ { Cont-Char }

String ::= { Every-Char }

Number ::= { Digit }

Name ::= (Uppercase | Lowercase) [{ Cont-Char | Uppercase }]

Cont-Char ::= Lowercase | Digit | $\boxed{\text{\_}}$

Einige Non-Terminalsymbole werden informal erläutert:

- **Uppercase** sind die Großbuchstaben
- **Lowercase** sind die Kleinbuchstaben
- **Digit** sind die Ziffern
- **Every-Char** sind alle Zeichen des Zeichensatzes
- **Implementation** ist der Name einer der für den zugrundeliegenden Typgenerator definierten Implementationen
- **Enrichment** ist der generische Name einer der für den zugrundeliegenden Typgenerator definierten Operationen
- **Param-Name** ist der Name eines für die Implementation oder die Operation definierten Parameters

Anhang B. Die Operationen der ADT

Zu jedem VDM-Basistyp werden hier alle verfügbaren Operationen mit ihrer Syntax und kurzer Erläuterung in alphabetischer Reihenfolge aufgelistet. Hinzu kommen eventuell zusätzlich nötige Definitionen, *preconditions* und spezielle Anmerkungen wie z.B. über Referenzparameter oder die Arbeitsweise; falls eine Operation nicht in allen Implementationen zur Verfügung steht, ist das ebenfalls angegeben. Ganz wichtig: zur Vermeidung unnötigen impliziten Kopieraufwandes wird bei vielen Operationen das manipulierte Objekt zerstört ! Auf diese wird besonders hingewiesen, Sie finden am Rand das

! $\rightarrow$ kleine Ausrufungszeichen (wie hier). Auf die gleiche Weise wird auf Referenzparameter hingewiesen (beabsichtigte Veränderung).

Bei Verletzung einer *precondition* wird in jedem Fall eine Fehler-Prozedur aufgerufen !

Die Syntax wird in ANSI–*C* spezifiziert ! Dies geschieht aus Gründen der Platzersparnis. Bei selbstdefinierten Funktionen achten Sie bitte darauf, ob Ihr Compiler ANSI-*C* verarbeitet.

Erweiterung der Namen

Sie finden hier die *generischen* Operationsnamen, die für ihre Anwendung noch erweitert werden: dazu wird erst ein Unterstrich angehängt, danach folgt in der Regel der ADT-Name, für den die Operation erzeugt wird. Bei Trees und Unions gibt es aber Operationen, die mit dem Namen der Komponente erweitert werden — diese Ausnahmen sind explizit angegeben. **Alle** Großbuchstaben werden allerdings bei der Erweiterung der Operationsnamen in Kleinbuchstaben umgewandelt, so daß diese nur aus Kleinbuchstaben, Ziffern und Unterstrichen bestehen !

Einige Operationen sind für alle ADT definiert, sie werden im ersten Abschnitt separat behandelt.

Generelle Operationen

Domain **copy** *(Domain x1)*
Eine physikalische Kopie des Objekts `x1` wird erzeugt; bei strukturierten Werten werden *alle* Sub-Strukturen ebenfalls kopiert.

Domain **copy1** *(Domain x1)*

Eine physikalische Kopie des Objekts x1 wird erzeugt; bei strukturierten Objekten werden die Sub-Strukturen *nicht* kopiert, stattdessen weisen Pointer auf die Original-Substrukturen.

Str⋆ **cvlines** *(Domain x1)*
Liefert den Wert von x1 als formatierter Text (Hilfsfunktion für print)
Die Darstellung der ELEMENTARY-Objekte entspricht dem Ergebnis der Operation cvstr

void **del** *(Domain x1)*
Das Objekt x1 gelöscht (bei strukturierten Objekten werden *alle* Sub-Strukturen ebenfalls gelöscht).

void **del1** *(Domain x1)*
Das Objekt x1 gelöscht (bei strukturierten Objekten werden die Sub-Strukturen *nicht* gelöscht).

Nat0 **hash** *(Domain x1)*
Ermittlung eines Hash-Wertes für das Objekt x1 (Hilfsfunktion für die Implementation von Sets und Maps mit Hilfe von Hash-Tabellen). Für strukturierte Objekte wird die Summe der Element-Hash-Werte gebildet.
Nat0, Nat1, Intg, Char: Ergebnis ist ein vorzeichenloser Wert
Parameter HASHSHIFT (optional): Anzahl von zusätzlichen, bitweisen Shifts nach rechts
Bool: Hash-Funktion ist die Identitäts-Funktion
Implementation „ptr“: Adresse wird in „unsigned int“ konvertiert und ein Shift um 2 Bits nach rechts durchgeführt

Bool **is_eq** *(Domain x1,Domain x2)*
Test auf Gleichheit

void **pr** *(Domain x1)*
Ausgabe des Wertes von x1 auf **stderr** mit Hilfe der jeweiligen cvlines–Operation, (strukturierte Objekte werden in angemessener Form dargestellt)

void **print** *(Domain x1)*
Ausgabe des Wertes von x1 auf **stderr** mit Hilfe der jeweiligen cvlines–Operation, unformatiert
print Sollte nicht benutzt werden, denn pr erzeugt eine formatierte Ausgabe!

ELEMENTARY (elem)

Domain **and** *(Domain x1,Domain x2)*
Logisches AND
Nat0, Nat1, Intg, Char: Bitweises logisches AND
Bool: Beide Argumente werden ausgewertet!

Str **cvstr** *(Domain x1)*
Liefert die Repräsentation von `x1` als String
Nat0, Nat1, Intg: minimal notwendige Stellenzahl
Bool: entweder „TRUE“ oder „FALSE“
Implementation „ptr“: Hexadezimale Adresse

Domain **false** *(void)*
Liefert die definierte Konstante *FALSE* (FALSE = 0)

Bool **is_ge** *(Domain x1,Domain x2)*
Test, ob `x1` $\geq$ `x2` ist

Bool **is_gr** *(Domain x1,Domain x2)*
Test, ob `x1` $>$ `x2` ist

Domain **new** *(Domain x1)*
Bei jedem Aufruf wird ein anderer Wert geliefert als bei allen vorherigen Aufrufen
Nur für die Implementationen „scalar“ und „unsigned“ verfügbar

Domain **not** *(Domain x1)*
Logisches NOT
Nat0, Nat1, Intg, Char: Bitweises logisches NOT

Domain **or** *(Domain x1,Domain x2)*
Logisches OR
Nat0, Nat1, Intg, Char: Bitweises logisches OR
Bool: Beide Argumente werden ausgewertet!

Domain **true** *(void)*
Liefert die definierte Konstante *TRUE* (TRUE = 1)

Domain **xor** *(Domain x1,Domain x2)*
Logisches Exklusiv-OR (XOR)
Nat0, Nat1, Intg, Char: Bitweises logisches XOR
Bool: Beide Argumente werden ausgewertet!

MAP (map)

Die Implementation „heap“ enthält nur sehr wenige Operationen; diese sind explizit gekennzeichnet.

func1 **binary** *(func2)*
#typedef Domain func1 (Domain,Domain)
#typedef Domain func2 (Domain,Element1)
Funktional
Der Name der Argument-Funktion wird mit dem Implementationsparameter FOLDFCT festgelegt; dient zur Realisierung von Funktionen mit zwei Maps als Argumenten, die wieder eine Map liefern, mit Hilfe von Funktionen mit einer Map und einem Argumentwert als Argumenten.
Beispiel: ovwrt = binary (FOLDFCT=„ovwrt1“))

Nat0 **card** *(Domain x1)*
Kardinalität (Anzahl der Paare) der Map x1

Element1 **ext** *(Domain$\star$ x1,Element2 x2)*
Die Map x1 wird um ein Paar (random $\mapsto$ x2) erweitert; der (zufällig) ermittelte Argumentwert wird zurückgeliefert, die Map ist Referenz-Parameter
! $\rightarrow$ x1 wird verändert
NUR in den Implementationen “heapmap“ und „heap“ vorhanden

func **fold** *(func1,func2,func3)*
#typedef Result func (Domain,x1,. . . ,xn)
#typedef Result func1 (Element1,Element2,x1,. . . ,xk)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result,x1,. . . ,xn)
Funktional
Die Namen der drei Argument-Funktionen werden mit den Implementationsparametern CVFCT, NULLFCT und FOLDFCT festgelegt, der Resultattyp mit dem Parameter RESULT. Die x1,. . . ,xn stellen zusätzliche optionale Parameter dar, die an func1 bzw. func3 übergeben werden; ihre Datentypen können mit den Implementationsparametern X, XX und XXX für func1 sowie Y, YY, YYY, Z, ZZ und ZZZ für func3 definiert werden. Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiel: Invertieren einer Map mit inverse, wenn sie bijektiv ist.

Hilfsfunktion:
swap_e1_e2 (x1,x2) = [(x2 $\mapsto$ x1)]
TYPE: Elem1 Elem2 $\longrightarrow$ Map2
DOMAINS
Map1 = E1 -m-> E2
Map2 = E2 -m-> E1

IMPLEMENTATIONS
Map1 = ...
  + fold.inverse (RESULT=¨Map2¨,
    CVFCT=„swap“,
    NULLFCT=„mk_empty“,
    FOLDFCT=„union“)

void **forall** *(Domain x1,proc x2)*
#typedef void proc (Element1,Element2)
Für alle Paare in der Map x1 wird die Prozedur x2 ausgeführt
Ist die Map geordnet, werden die Objekte in dieser Reihenfolge verarbeitet.

void **forrange** *(Domain x1,proc x2,Element1 x3,Element1 x4)*
PRECONDITION: Für den ADT Element1 muß eine Relation $\leq$ existieren !
#typedef void proc (Element1,Element2)
Für alle Paare (a $\mapsto$ b) in der Map x1 mit $x3 \leq\ a \leq\ x4$ wird die Prozedur x2 ausgeführt

void **forsome** *(Domain x1,predicate x2,proc x3)*
#typedef Bool predicate (Element1)
#typedef void proc (Element1,Element2)
Für alle Paare (a $\mapsto$ b) in der Map x1, für die das Prädikat x2 (a) erfüllt ist, wird die Prozedur x3 ausgeführt

Domain **fread** *(Str x1)*
PRECONDITION: (Arg-Dom = Str) $\wedge$ (Co-Dom hat Operation fread)
Eine Map wird aus dem Verzeichnis x1 des Datei-Systems gelesen — jede Datei repräsentiert ein Paar (Dateiname $\mapsto$ Tuple), jedes Unterverzeichnis ein Paar (Verzeichnisname $\mapsto$ Map)
Bei einem Fehler wird die leere Map zurückgeliefert !

Bool **fwrite** *(Domain x1,Str x2)*
PRECONDITION: Arg-Dom = Str $\wedge$ Co-Dom kennt Operation fread
Eine Map x1 wird in das Verzeichnis x2 des Datei-Systems geschrieben — jede Datei repräsentiert ein Paar (Dateiname $\mapsto$ Tuple), jedes Unterverzeichnis ein Paar (Verzeichnisname $\mapsto$ Map)
Bei einem Fehler wird *FALSE* zurückgeliefert, sonst *TRUE*

Domain **inters** *(Domain x1,Domain x2)*
Der Durchschnitt der beiden Maps x1 und x2 wird geliefert
! $\rightarrow$ x1 und x2 werden zerstört, Sub-Strukturen allerdings nicht !

Domain **intersd** *(Domain x1,Domain x2)*
Der Durchschnitt der beiden Maps x1 und x2 wird geliefert Sämtliche Sub-

Strukturen, die nicht im Resultat enthalten sind, werden mit del gelöscht.
! → x1 und x2 werden zerstört

Bool **is_empty** *(Domain x1)*
Test auf leere Map

Bool **is_in** *(Element1 x1,Domain x2)*
Test, ob das Element x1 in der Argumentmenge der Map x2 enthalten ist

Bool **is_in_dom** *(Element1 x1,Domain x2,Element2⋆ x3)*
Test, ob das Element x1 in der Argumentmenge der Map x2 enthalten ist; wenn ja, wird der zugehörige Map-Wert im Referenzparameter x3 zurückgegeben
! → x3 wird verändert

Bool **is_in_max** *(Element1 x1,Domain x2,Element2⋆ x3)*
Test, ob es ein Paar (a ↦ b) mit $a \leq x1$ gibt; wenn ja, wird das b mit dem maximalen a im Referenzparameter x3 gespeichert
! → x3 wird verändert

Bool **is_in_min** *(Element1 x1,Domain x2,Element2⋆ x3)*
Test, ob es ein Paar (a ↦ b) mit $a \geq x1$ gibt; wenn ja, wird das b mit dem minimalen a im Referenzparameter x3 gespeichert
! → x3 wird verändert

Bool **is_in_rng** *(Element2 x1,Domain x2,Element1⋆ x3)*
Test, ob x1 im Co-Domain der Map x2 existiert; wenn ja, wird der zugehörige Argumentwert im Referenzparameter x3 gespeichert
! → x3 wird verändert

Bool **is_submap** *(Domain x1,Domain x2)*
Test, ob die Map x1 in der Map x2 enthalten ist

Domain **mk_empty** *(void)*
Die leere Map wird geliefert
In der Implementation „heap“ vorhanden

Domain **mk_one** *(Element1 x1,Element2 x2)*
Die einelementige Map [(x1 ↦ x2)] wird geliefert

Domain **ovwrt** *(Domain x1,Domain x2)*
Die Map x2 wird mit der Map x1 vereinigt; bei Argumentwerten, die in beiden Maps existieren, wird das Paar aus x2 genommen
! → x1 und x2 werden zerstört

Domain **ovwrt1** *(Domain x1,Element1 x2,Element2 x3)*
Das Paar [x2 $\mapsto$ x3] wird in die Map x1 eingetragen — war is_in (x1,x2) erfüllt, wird das alte Paar überschrieben
! $\rightarrow$ x1 wird zerstört
In der Implementation „heap“ vorhanden

Domain **ovwrt1d** *(Domain x1,Element1 x2,Element2 x3)*
Das Paar [x2 $\mapsto$ x3] wird in die Map x1 eingetragen — ist x2 bereits in der Argumentmenge der Map enthalten, wird das alte Paar mit del gelöscht
! $\rightarrow$ x1 wird zerstört

Domain **ovwrtd** *(Domain x1,Domain x2)*
Die Map x2 wird mit der Map x1 vereinigt; bei Argumentwerten, die in beiden Maps existieren, wird das Paar aus x2 genommen und das entsprechende Paar aus x1 mit del gelöscht
! $\rightarrow$ x1 und x2 werden zerstört

Element2 **sel** *(Domain x1,Element1 x2)*
PRECONDITION: is_in (x1,x2) = TRUE
Der x2 zugeordnete Map-Wert wird geliefert
In der Implementation „heap“ vorhanden

Element1 **sel_dom** *(Domain x1)*
PRECONDITION: x1 $\neq$ []
Irgendein Argumentwert der Map x1 wird geliefert

Domain **sub** *(Domain x1,Domain x2)*
Subtraktion zweier Maps: aus der Map x1 werden alle Paare, deren Argumentwert auch in x2 enthalten ist, entfernt
! $\rightarrow$ x1 wird zerstört

Domain **sub1** *(Domain x1,Element1 x2)*
Das Paar (x2 $\mapsto$ sel (x1,x2)) wird aus der Map entfernt
! $\rightarrow$ x1 wird zerstört
In der Implementation „heap“ vorhanden

Domain **sub1d** *(Domain x1,Element1 x2)*
Das Paar [x2 $\mapsto$ sel (x1,x2)] wird aus der Map entfernt und, wenn es tatsächlich enthalten war, mit del gelöscht
! $\rightarrow$ x1 wird zerstört

Domain **subd** *(Domain x1,Domain x2)*
Subtraktion zweier Maps: aus der Map x1 werden alle Paare, deren Argumentwert auch in x2 enthalten ist, entfernt
Alle nicht weiter verwendeten Sub-Strukturen von x1 werden mit del gelöscht

! → x1 wird zerstört

Domain **union** *(Domain x1,Domain x2)*
PRECONDITION: $(\forall$ i $\in$ dom x1 $\cap$ dom x2$)$ $($x1(i) $=$ x2(i)$)$
Vereinigung zweier Maps
! → x1 und x2 werden zerstört

Domain **union1** *(Domain x1,Element1 x2,Element2 x3)*
PRECONDITION: is_in (x1,x2) $\supset$ sel (x1,x2) $=$ x3
Das Paar (x2 $\mapsto$ x3) wird in die Map x1 eingetragen
! → x1 wird zerstört
In der Implementation „heap“ vorhanden

Domain **union1d** *(Domain x1,Element1 x2,Element2 x3)*
PRECONDITION: is_in (x1,x2) $\supset$ sel (x1,x2) $=$ x3
Das Paar (x2 $\mapsto$ x3) wird in die Map x1 eingetragen; ist das *gleiche* Paar bereits in der Map enthalten, wird ein Paar gelöscht
! → x1 wird zerstört

Domain **uniond** *(Domain x1,Domain x2)*
PRECONDITION: $(\forall$ i $\in$ dom x1 $\cap$ dom x2$)$ $($x1(i) $=$ x2(i)$)$
Vereinigung zweier Maps; wenn ein Paar in beiden Maps enthalten ist, wird eines davon mit del gelöscht
! → x1 und x2 werden zerstört

OPTIONAL (opti)

Bool **is_element** *(Domain x1)*
Test, ob aktueller Wert des Optionals x1 gerade NICHT **nil** ist
Der Name wird mit dem Namen des Elementtyps (wie oben angedeutet) UND dem Namen des Optionaltyps erweitert!

Bool **is_ge** *(Domain x1,Domain x2)*
Test, ob x1 $\geq$ x2 ist

Bool **is_gr** *(Domain x1,Domain x2)*
Test, ob x1 $>$ x2 ist

Bool **is_nil** *(Domain x1)*
Test, ob aktueller Wert des Optionals x1 gerade **nil** ist

Bool **is_opt** *(Domain x1,Element⋆ x2)*
Test, ob aktueller Wert des Optionals x1 gerade ein gültiger Element-Wert ist; wenn ja, wird das Element im Referenzparameter x2 geliefert
! $\rightarrow$ x2 wird verändert

Domain **mk_element** *(Element x1)*
Liefert ein Optional mit dem aktuellen Wert x1
Identisch mit mk_opt
Der Name wird mit dem Namen des Elementtyps (wie oben angedeutet) UND dem Namen des Optionaltyps erweitert!

Domain **mk_nil** *(void)*
Liefert ein Optional mit dem aktuellen Wert **nil**

Domain **mk_opt** *(Element x1)*
Liefert ein Optional mit dem aktuellen Wert x1

Element **s_element** *(Domain x1)*
PRECONDITION: is_nil(x1) = FALSE
Liefert den aktuellen Wert des Optionals x1
Der Name wird mit dem Namen des Elementtyps (wie durch „element“ angedeutet) UND dem Namen des Optionaltyps erweitert!

SET (set)

func1 **binary** *(func2)*
#typedef Domain func1 (Domain,Domain)
#typedef Domain func2 (Domain,Element)
Funktional
Der Name der Argument-Funktion wird mit dem Implementationsparameter FOLDFCT festgelegt; dient zur Realisierung von Funktionen mit zwei Sets als Argumenten, die wieder einen Set liefern, mit Hilfe von Funktionen mit einem Set und einem Element als Argumenten.
Beispiel: union = binary (FOLDFCT=„union1“))

Nat0 **card** *(Domain x1)*
Kardinalität (Anzahl der Elemente) des Set x1

Domain **diff** *(Domain x1,Domain x2)*
Set-Differenz: aus dem Set x1 werden alle Elemente, die auch in x2 enthalten ist, entfernt
! → x1 wird zerstört

Domain **diffd** *(Domain x1,Domain x2)*
Set-Differenz: aus dem Set x1 werden alle Elemente, die auch in x2 enthalten ist, entfernt und dabei mit del gelöscht
! → x1 wird zerstört

func **fold** *(func1,func2,func3)*
#typedef Result func (Domain,x1,...,xn)
#typedef Result func1 (Element,x1,...,xk)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result,x1,...,xn)
Funktional
Die Namen der drei Argument-Funktionen werden mit den Implementationsparametern CVFCT, NULLFCT und FOLDFCT festgelegt, der Resultattyp mit dem Parameter RESULT. Die x1,...,xn stellen zusätzliche optionale Parameter dar, die an func1 bzw. func3 übergeben werden; ihre Datentypen können mit den Implementationsparametern X, XX und XXX für func1 sowie Y, YY, YYY, Z, ZZ und ZZZ für func3 definiert werden. Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiel: Distributive Vereinigung der Elemente eines Set von Sets

DOMAINS
Set1 = E1 –set
Set2 = Set1 –set
IMPLEMENTATIONS
Set2 = ...+ fold.dunion (NULLFCT=„mk_empty“, FOLDFCT=„union“)

void **forall** *(Domain x1,proc x2)*
#typedef void proc (Element)
Für alle Elemente im Set x1 wird die Prozedur x2 ausgeführt

void **forsome** *(Domain x1,predicate x2,proc x3)*
#typedef Bool predicate (Element)
#typedef void proc (Element)
Für alle Elemente a im Set x1, für die das Prädikat x2 (a) erfüllt ist, wird die Prozedur x3 ausgeführt

Domain **inters** *(Domain x1,Domain x2)*
Der Durchschnitt der beiden Sets x1 und x2 wird geliefert
! $\rightarrow$ x1 und x2 werden zerstört

Domain **intersd** *(Domain x1,Domain x2)*
Der Durchschnitt der beiden Sets x1 und x2 wird geliefert
Die nicht im Resultat enthaltenen Elemente werden mit del gelöscht
! $\rightarrow$ x1 und x2 werden zerstört

Bool **is_empty** *(Domain x1)*
Test auf leeren Set

Bool **is_in** *(Element x1,Domain x2)*
Test, ob das Element x1 im Set x2 enthalten ist

Bool **is_subset** *(Domain x1,Domain x2)*
Test, ob der Set x1 im Set x2 enthalten ist

Domain **mk_empty** *(void)*
Der leere Set wird geliefert

Domain **mk_one** *(Element x1)*
Der einelementige Set { x1 } wird geliefert

Element **sel** *(Domain x1)*
Irgendein Element des Set x1 wird geliefert

Domain **sub1** *(Domain x1,Element x2)*
Das Element x2 wird aus dem Set x1 entfernt

! → x1 wird zerstört

Domain **sub1d** *(Domain x1,Element x2)*
Das Element x2 wird aus dem Set x1 entfernt; war es tatsächlich vorhanden, wird es mit del gelöscht
! → x1 wird zerstört

Domain **union** *(Domain x1,Domain x2)*
Vereinigung zweier Sets
! → x1 und x2 werden zerstört

Domain **union1** *(Domain x1,Element x2)*
Das Element x2 wird in den Set x1 eingetragen
! → x1 wird zerstört

Domain **union1d** *(Domain x1,Element x2,Element2 x3)*
Das Element x2 wird in den Set x1 eingetragen; ist x2 bereits in x1 enthalten, wird es mit del gelöscht
! → x1 wird zerstört

Domain **uniond** *(Domain x1,Domain x2)*
Vereinigung zweier Sets; ist ein Element in beiden Maps enthalten, wird eines davon mit del gelöscht
! → x1 und x2 werden zerstört

TREE (tree)

Domain **chg** *(Domain x1,Component-x x2)*
Ersetzen einer Komponente des Tree x1 mit x2
Nomenklatur Der Name wird nicht mit dem Namen des jeweiligen Elementtyps erzeugt, sondern mit dem Namen der Komponente OHNE das führende s_ .
Das überschriebene Objekt wird NICHT gelöscht.
! → x1 wird zerstört (im Gegensatz zu subst)

func **cfold** *(func11,func12,...,func1n,func2,func3)*
#typedef Result func (Domain)
#typedef Result func11 (Component1)
#typedef Result func12 (Component2)
⋮
#typedef Result func1n (Componentn)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result)
Funktional
Die Namen der Argument-Funktionen werden mit den Implementationsparametern CVFCT, NULLFCT und FOLDFCT festgelegt, der Resultattyp mit dem Parameter RESULT. CVFCT enthält pro Komponente des Tree eine Konvertierungsfunktion, die mit dem Namen des jeweiligen Elementtyps erweitert wird. Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiele für fold–Funktionen siehe Sets, Maps, Tuples

Bool **is_ge** *(Domain x1,Domain x2)*
Test, ob x1 $\geq$ x2 ist

Bool **is_gr** *(Domain x1,Domain x2)*
Test, ob x1 $>$ x2 ist

Domain **mk** *(Component1 x1,...,Componentn xn)*
Erzeugt einen Tree aus den Komponenten

Component-x s *(Domain x1)*
Eine Komponente des Tree wird geliefert
Nomenklatur Der Name wird nicht mit dem Namen des jeweiligen Elementtyps erweitert, sondern mit dem Namen der Komponente OHNE das führende s_ . Der Operationsname ist also identisch mit dem Komponentennamen !

func **sfold** *(func11,func12,...,func1n,func2,func3)*
#typedef Result func (Domain)
#typedef Result func11 (Component1)
#typedef Result func12 (Component2)

⋮

#typedef Result func1n (Componentn)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result)
Funktional
Die Namen der drei Argument-Funktionen werden mit den Implementationsparametern CVFCT, NULLFCT und FOLDFCT festgelegt, der Resultattyp mit dem Parameter RESULT. CVFCT enthält pro Komponente des Tree eine Konvertierungsfunktion, die mit dem Namen der jeweiligen Selektor-Operation erweitert wird. Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiele für fold–Funktionen siehe Sets, Maps, Tuples

Domain **subst** *(Domain x1,Component-x x2)*
Ersetzen einer Komponente des Tree x1 mit x2, wobei der Tree kopiert wird — das Original bleibt also unversehrt (im Unterschied zu chg)

Nomenklatur Der Name wird nicht mit dem Namen des jeweiligen Elementtyps erzeugt, sondern mit dem Namen der Komponente OHNE das führende s_ .

TUPLE (tupl)

Domain **appl** *(Element x1, Domain x2)*
Das Element x1 wird am Anfang des Tuple x2 eingefügt
Andere Argumentreihenfolge als appr !
! → x2 wird zerstört

Domain **appr** *(Domain x1,Element x2)*
Das Element x2 wird am Ende des Tuple x1 angehängt
Andere Argumentreihenfolge als appl !
! → x1 wird zerstört

func1 **binary** *(func2)*
#typedef Domain func1 (Domain,Domain)
#typedef Domain func2 (Domain,Element)
Funktional
Der Name der Argument-Funktion wird mit dem Implementationsparameter FOLDFCT festgelegt; dient zur Realisierung von Funktionen mit zwei Tuples als Argumenten, die wieder ein Tuple liefern, mit Hilfe von Funktionen mit einem Tuple und einem Element als Argumenten.
Beispiel: conc = binary (FOLDFCT=„appr“))

Domain **conc** *(Domain x1,Domain x2)*
Konkatenation: Tuple x2 wird an Tuple x1 angehängt
! → x1 und x2 werden zerstört

Domain **conc_path** *(Domain x1,Domain x2)*
Konstruktion eines Pfadnamen durch Konkatenation: Tuple x2 wird an Tuple x1 gehängt, NACHDEM an x1 ein Pfad-Delimiter angefügt wurde (unter *UNIX* der Slash „/“, unter *MS-DOS* der Backslash „\“)
! → x1 und x2 werden zerstört

void **cpc** *(Domain x1,char⋆ x2)*
Das Tuple x1 wird in einen *C*–String konvertiert, der an der Adresse x2 anfängt; an das Ende des Strings wird ein Null-Byte gehängt
Sinnvoll nur bei Tuples von Zeichen (z.B. Str)
In der Implementation „linklist“ nicht vorhanden

Bool **fapp** *(Domain x1,Str x2)*
Ein Tuple x1 wird an die Datei x2 des Datei-Systems angehängt
Es wird angenommen, daß die Elemente einfache, unstrukturierte Objekte sind.
Bei einem Fehler wird *FALSE* zurückgeliefert, sonst *TRUE*

func **fold** *(func1,func2,func3)*
#typedef Result func (Domain,x1,...,xn)
#typedef Result func1 (Element,x1,...,xk)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result,x1,...,xn)
Funktional
Die Namen der drei Argument-Funktionen werden mit den Implementationsparametern CVFCT, NULLFCT und FOLDFCT festgelegt, der Resultattyp mit dem Parameter RESULT. Die x1,...,xn stellen zusätzliche optionale Parameter dar, die an func1 bzw. func3 übergeben werden; ihre Datentypen können mit den Implementationsparametern X, XX und XXX für func1 sowie Y, YY, YYY, Z, ZZ und ZZZ für func3 definiert werden. Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiel: Distributive Verkettung der Elemente eines Tuples von Tuples
DOMAINS
Tup1 = E1 ⋆
Tup2 = Tup1 ⋆
IMPLEMENTATIONS
Tup2 = ...+ fold.dconc (NULLFCT=„mk_empty“, FOLDFCT=„conc“)

func **foldix** *(func1,func2,func3)*
#typedef Result func (Domain,x1,...,xn)
#typedef Result func1 (Element,Nat0,x1,...,xk)
#typedef Result func2 (void)
#typedef Result func3 (Result,Result,x1,...,xn)
Funktional (ähnlich fold) — hier hat die Convert-Funktion func1 als weiteres Argument einen Element-Index, um z.B. die Verarbeitung nur jedes zweiten Elements zu ermöglichen.
Alles weitere ist identisch mit fold (siehe dort).

void **forall** *(Domain x1,proc x2)*
#typedef void proc (Element)
Für alle Elemente im Tuple x1 wird die Prozedur x2 ausgeführt

void **forindex** *(Domain x1,proc x2)*
#typedef void proc (Element,Nat1)
Für alle Elemente im Tuple x1 wird die Prozedur x2 ausgeführt; diese erhält als zusätzliches Argument den Element-Index

void **forrange** *(Domain x1,proc x2,Nat1 x3,Nat0 x4)*
#typedef void proc (Element)
Für den Indexbereich x3...x4 im Tuple x1 wird die Prozedur x2 ausgeführt

void **forsome** *(Domain x1,predicate x2,proc x3)*
#typedef Bool predicate (Element)

#typedef void proc (Element)
Für alle Elemente a im Tuple x1, für die das Prädikat x2 (a) erfüllt ist, wird die Prozedur x3 ausgeführt

Domain **fread** *(Str x1)*
Ein Tuple wird aus der Datei x1 des Datei-Systems gelesen
Es wird angenommen, daß die Elemente einfache, unstrukturierte Objekte sind.
Bei einem Fehler wird das leere Tuple zurückgeliefert !

Bool **fwrite** *(Domain x1,Str x2)*
Ein Tuple x1 wird in die Datei x2 des Datei-Systems geschrieben
Es wird angenommen, daß die Elemente einfache, unstrukturierte Objekte sind.
Bei einem Fehler wird *FALSE* zurückgeliefert, sonst *TRUE*

Element **hd** *(Domain x1)*
Liefert das erste Element des Tuple

Nat0 **index** *(Domain x1,Element x2)*
Liefert die erste Position des Elements x2 im Tuple x1
Ist das Element NICHT vorhanden, wird 0 geliefert

Bool **is_empty** *(Domain x1)*
Test auf leeres Tuple

Bool **is_ge** *(Domain x1,Domain x2)*
Test, ob x1 $\geq$ x2 ist

Bool **is_gr** *(Domain x1,Domain x2)*
Test, ob x1 $>$ x2 ist

Element **last** *(Domain x1)*
PRECONDITION: x1 $\neq < >$
Das letzte Element im Tuple x1 wird geliefert

Nat0 **len** *(Domain x1)*
Liefert die Länge (Anzahl der Elemente) des Tuple x1

Domain **mk2** *(Element x1,Element x2)*
Konstruktion eines zwei-elementigen Tuple $<$ x1,x2 $>$

Domain **mk3** *(Element x1,Element x2,Element x3)*
Konstruktion eines drei-elementigen Tuple $<$ x1,x2,x3 $>$

Domain **mkc** *(char⋆ x1)*
Der *C*–String, der an der der Adresse x2 anfängt, wird OHNE das abschließende Null-Byte in ein Tuple konvertiert
Sinnvoll nur bei Tuples von Zeichen (z.B. Str)
In der Implementation „linklist" nicht vorhanden

Domain **mklc** *(char⋆ x1)*
Der *C*–String, der an der der Adresse x2 anfängt, wird OHNE das abschließende Null-Byte in ein Tuple konvertiert, dabei wird eine Wandlung in Kleinbuchstaben vorgenommen.
Sinnvoll nur bei Tuples von Zeichen (z.B. Str)
In der Implementation „linklist" nicht vorhanden

Domain **mkn** *(Element x1,Nat0 x2)*
Konstruktion eines Tuple mit x2 identischen Elementen x1

Domain **mk_empty** *(void)*
Das leere Tuple wird geliefert

Domain **mk_one** *(Element x1)*
Das einelementige Tuple $< x1 >$ wird geliefert

Domain **ovwrt1** *(Domain x1,Nat1 x2,Element x3)*
PRECONDITION: len (x1) $\geq$ x2
Das Element x3 wird an der Position x2 in das Tuple x1 eingetragen
! $\rightarrow$ x1 wird zerstört

Domain **ovwrt1d** *(Domain x1,Nat1 x2,Element x3)*
PRECONDITION: len (x1) $\geq$ x2
Das Element x3 wird an der Position x2 in das Tuple x1 eingetragen; der vorherige Eintrag wird zusätzlich mit del gelöscht
! $\rightarrow$ x1 wird zerstört

void **puts** *(Domain x1,FILE⋆ x2)*
Das Tuple x1 wird zeichenweise mit der *C*–Bibliotheksfunktion „fputc" in eine *C*–Datei geschrieben, die mit x2 = fopen (...) geöffnet wurde.
Sinnvoll nur bei Tuples von Zeichen (z.B. Str)
In der Implementation „linklist" nicht vorhanden

Domain **rmv1** *(Domain x1,Nat1 x2)*
PRECONDITION: len (x1) $\geq$ x2
Das Element an Position x2 wird aus dem Tuple x1 entfernt, die anderen Elemente rücken auf
! $\rightarrow$ x1 wird zerstört

Domain **rmv1d** *(Domain x1,Nat1 x2)*
PRECONDITION: len (x1) $\geq$ x2
Das Element an Position x2 wird aus dem Tuple x1 entfernt, die anderen Elemente rücken auf
Das entfernte Element wird zusätzlich mit del gelöscht
! $\rightarrow$ x1 wird zerstört

Nat0 **search** *(Domain x1,predicate x2)*
#typedef Bool predicate (Element)
Das erste Element im Tuple x1, das das Prädikat x2 erfüllt, wird geliefert; ist keins vorhanden, wird 0 geliefert

Element **sel** *(Domain x1,Nat1 x2)*
PRECONDITION: len (x1) $\geq$ x2
Das Element an der Position x2 des Tuple x1 wird geliefert

Domain **sub_tup** *(Domain x1,Nat1 x2,Nat0 x3)*
Das Sub-Tuple mit den Elementen an den Positionen x2...x3 wird geliefert; Bereichsüberschreitungen werden sinnvoll bearbeitet.
! $\rightarrow$ x1 wird zerstört

Domain **tl** *(Domain x1)*
Das Tuple x1 ohne das erste Element wird geliefert
! $\rightarrow$ x1 wird zerstört

Domain **tld** *(Domain x1)*
Das Tuple x1 ohne das erste Element wird geliefert, zuvor wird del (hd (x1)) aufgerufen
! $\rightarrow$ x1 wird zerstört

UNION (unio)

Domain **chg** *(Domain x1,Component-x x2)*
PRECONDITION für chg_u1: is_u1 (x1) = TRUE
Überschreiben des Inhalts der Union x1 durch den Wert x2, der vom gleichen Typ sein muß

Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps.
Das überschriebene Objekt wird NICHT implizit gelöscht

! → x1 wird zerstört (im Gegensatz zu subst)

func **cswitch** *()*
Funktional (ähnlich wie sswitch)
Erzeugt „Switch-Funktionen", die der **switch**-Klausel in *C* gleichen (wie bei discr zu sehen). Beim Aufruf wird der aktuelle Typ der Union bestimmt und abhängig davon eine Bearbeitungs-Funktion (die Sie an anderer Stelle definieren) aufgerufen.
Die Namen der Bearbeitungs-Funktionen ergeben sich aus dem Namen der „Switch-Funktion" durch Erweiterung mit dem Namen des Elementtyps (im Unterschied zu sswitch).

Parameter: Result gibt den Datentyp des Resultats der erzeugten Funktion an (Default: void)
XX,XXX,XXXX,... sind Datentypen von optionalen zusätzlichen Argumenten dieser Funktion
Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.
Beispiel:
DOMAINS
Uni = s_u1 : Nat0 | s_u2 : Str
Xyz = ...
IMPLEMENTATIONS
Uni = ...+ cswitch.case (Result=„Intg", XX=„Xyz")
erzeugt die Switch-Funktion
case_uni (x1,x2) =
(is_u1 (x1) → case_nat0 (s_u1 (x1),x2)
is_u2 (x1) → case_str (s_u2 (x1),x2)
)
TYPE: Uni XyZ ⟶ Intg
Die Bearbeitungs-Funktionen haben folgende TYPE-Klauseln:
TYPE case_nat0 : Nat0 XyZ ⟶ Intg
TYPE case_str : Str XyZ ⟶ Intg

Discriminant-Domain **discr** *(Domain x1)*
#enum Discriminant-Domain {d_xxx,d_yyy,...}
Erzeugt ein mit der **switch**-Klausel in *C* abfragbares Flag, das anzeigt, wel-

chen Datentyp die Union x1 hat
Die Namen der Flags entsprechen den Namen der Komponenten, das führende „s_" wird durch „**d_**" ersetzt.
Beispiel: Uni = s_u1 : E1 | s_u2 : E2

```
Uni u;
  ⋮
switch (discr_uni (u)) {
  case d_u1: ...; break;
  case d_u2: ...; break;
}
```

Bool **is** *(Domain x1)*
Test, welcher Datentyp aktuell ist; der Aufruf der „richtigen" Operation ergibt *TRUE*, jeder andere ergibt *FALSE*.
Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps.

Bool **is_ge** *(Domain x1,Domain x2)*
Test, ob x1 $\geq$ x2 ist

Bool **is_gr** *(Domain x1,Domain x2)*
Test, ob x1 $>$ x2 ist

Domain **mk** *(Component-x x1)*
Erzeugt eine Union aus dem Objekt x1
Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps.

Domain **ovwrt** *(Domain x1,Component-x x2*
Ersetzen des aktuellen Inhalts von x1 durch x2 — dabei kann auch der Typ von x1 verändert werden
Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps.
! $\rightarrow$ x1 wird zerstört

Component-x **s** *(Domain x1)*
PRECONDITION für s_u1: is_u1 (x1) = TRUE
Der Inhalt der Union wird als Objekt vom gewünschten Komponententyp geliefert, wenn die Union gerade diesen Typ hat
Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps. Der Operationsname entspricht damit dem Namen der Komponente, wenn dieser angegeben ist !

func **sswitch** *()*

Funktional (ähnlich wie cswitch)
Erzeugt „Switch-Funktionen“, die der **switch**-Klausel in *C* gleichen (wie bei discr schon gesehen). Beim Aufruf wird der aktuelle Typ der Union bestimmt und abhängig davon eine Bearbeitungs-Funktion (die Sie an anderer Stelle definieren) aufgerufen.
Die Namen der Bearbeitungs-Funktionen ergeben sich aus dem Namen der „Switch-Funktion“ durch Erweiterung mit dem Komponentennamen OHNE führendes „s_“ (im Unterschied zu cswitch).
Die genaue Spezifikation können Sie mit *vdminfo* ermitteln.

Parameter: Result gibt den Datentyp des Resultats der erzeugten Funktion an (Default: void)
XX,XXX,XXXX,... sind Datentypen von optionalen zusätzlichen Argumenten dieser Funktion
Beispiel:
DOMAINS
Uni = s_u1 : Nat0 | s_u2 : Str
IMPLEMENTATIONS
Uni = ...+ cswitch.case ()
erzeugt die Switch-Funktion
case_uni (x1) =
(is_u1 (x1) → case_u1 (s_u1 (x1))
is_u2 (x1) → case_u2 (s_u2 (x1))
)
TYPE: Uni ⟶
Die Bearbeitungs-Funktionen haben folgende TYPE-Klauseln:
TYPE case_u1 : Nat0 ⟶
TYPE case_u2 : Str ⟶

Domain **subst** *(Domain x1,Component-x x2)*
PRECONDITION für subst_u1: is_u1 (x1) = TRUE
Überschreiben des Inhalts der Union x1 durch den Wert x2, der vom gleichen Typ sein muß, nachdem x1 kopiert wurde (der alte Inhalt bleibt erhalten)

Nomenklatur Der Name wird erweitert; wenn angegeben, mit dem Namen der Komponente OHNE das führende s_ , sonst mit dem Namen des jeweiligen Elementtyps.

Anhang C. Die Implementationen

Die verfügbaren Implementationen sind nach den VDM-Basistypen geordnet; beide finden Sie in alphabetischer Reihenfolge. Sie erfahren generelle Eigenschaften, Einschränkungen bezüglich der Datentypen oder Element-Implementationen, die interne Repräsentation sowie die Parameter. Fast alle Parameter sind optional; der Default ist angegeben. Bei Flags bedeutet der Wert 0, daß es nicht gesetzt ist. In der Überschrift finden Sie zusätzlich zum Namen des ADT übrigens noch seine Bezeichnung in der VDM-Bibliothek, die auf die ersten vier Zeichen verkürzt ist.

Strategien zur Speicherallokation

Einen entscheidenden Einfluß auf die Performance hat die zugrundeliegende Speicherallokationsstrategie der auf Pointern basierenden Implementationen (die sehr oft verwendet werden). Sie können sie mit dem Implementationsparameter „ALLOC“ festlegen. Hier eine kurze Übersicht der vorhandenen Möglichkeiten:

- **freelist**: Freispeicherliste; wenn sie überläuft, wird automatisch die Strategie „malloc“ verwendet. Der Parameter „MAXLEN“ legt die maximale Größe fest.
- **malloc**: Verwaltung des Speichers mit den Standard–C-Funktionen malloc und free. Der Parameter „SAVE“ schaltet die Prüfung auf die Verwendung von legalen Pointern ein (sinnvoll nur während des Programmtests).
- **vmalloc**: Verwaltung des Speichers in Einheiten variabler Länge (in Zweierpotenzen) auf Basis von malloc und free. Der Parameter „SAVE“ schaltet die Prüfung auf die Verwendung von legalen Pointern ein (sinnvoll nur während des Programmtests).

Parameter „PROFILE“

Alle drei Strategien kennen den Parameter „PROFILE“, der die Anzahl der Aufrufe der malloc– und free–Funktionen zählt und eine zusätzliche Operation profile bereitstellt, die diese Informationen ausgibt.

ELEMENTARY (elem)

bool

Datentyp(en):
- Bool

Repräsentation:
- Kleinster numerischer Datentyp („unsigned char“)

Parameter:
- keine

Anmerkungen:
- String-Konvertierung liefert „TRUE“ oder „FALSE“
- „hash“-Operation ist die Identität

ptr

Datentyp(en):
- Pointer auf beliebige Typen

Repräsentation:
- Pointer auf den entsprechenden Typ. Damit das referenzierte Datum korrekt verarbeitet werden kann, sollten Sie dessen Datentyp mit dem Parameter „ELEM“ angeben.

Parameter:
- **ELEM**: Der Datentyp der Objekte, auf die der Pointer zeigt (Default: void)

Anmerkungen:
- Geeignete Implementation für TOKEN
- String-Konvertierung liefert hexadezimale Adresse
- „hash“-Operation liefert „unsigned int“

scalar

Datentyp(en):
- Intg
- Unterbereiche der Menge $\mathcal{Z}$ (Ganzzahlen)

Repräsentation:
- Vorzeichenbehaftete Integer (char, short, long)

Parameter:
- **LB**: „Lower Bound“ = Untergrenze des Wertebereichs (Default: kleinstmöglicher Wert)
- **UB**: „Upper Bound“ = Obergrenze des Wertebereichs (Default: größtmöglicher Wert)

Anmerkungen:

- String-Konvertierung liefert die minimale Zeichenanzahl
- „hash“-Operation liefert „unsigned“. Ein optionales Rechtsschieben kann mit dem Parameter „HASHSHIFT“ angegeben werden.
- „new“-Operation liefert bei jedem Aufruf einen vorher nicht benutzten Wert

unsigned

Datentyp(en):

- Nat0
- Nat1
- Unterbereiche der Menge $\mathcal{N}_0$ der natürlichen Zahlen
- Char

Repräsentation:

- Vorzeichenlose Integer (unsigned char, unsigned short, unsigned long)

Parameter:

- **LB**: „Lower Bound“ = Untergrenze des Wertebereichs (Default: 0)
- **UB**: „Upper Bound“ = Obergrenze des Wertebereichs (Default: größtmöglicher Wert)

Anmerkungen:

- String-Konvertierung liefert die minimale Zeichenanzahl
- „hash“-Operation liefert „Nat0“. Ein optionales Rechtsschieben kann mit dem Parameter „HASHSHIFT“ angegeben werden.
- „new“-Operation liefert bei jedem Aufruf einen vorher nicht benutzten Wert

MAP (map)

dir

Datentyp(en):

- Str $\overrightarrow{m}$ Tuple
- Str $\overrightarrow{m}$ Map

Repräsentation:

- Die Map wird als Verzeichnis im Datei-System implementiert. Die Argumentwerte müssen Tuples von Zeichen (Strings) sein; sie entsprechen Dateinamen, wenn der Co-Domain aus Tuples besteht (die in Dateien angelegt werden), oder Verzeichnisnamen, wenn der Co-Domain aus Maps besteht (deren Argument-Domain wiederum „Str“ sein muß). Allgemein müssen für den Co-Domain die Operationen fread und fwrite existieren.

Parameter:

- **LINKLIST**: Name der Map-Implementation, die für die Objekte verwendet wird, die sich temporär im Hauptspeicher befinden (Default: „linklist“)

Anmerkungen:

- Gut geeignet zur Arbeit mit dem Datei-System von UNIX.

heap

Datentyp(en):

- Token $\overrightarrow{m}$ Co-Domain

Repräsentation:

- Die Map-Werte werden auf dem Heap angelegt und durch Pointer dereferenziert. Die Sammlung der Pointer entspricht dem Datentyp „Token“, für den keine Repräsentation vorgegeben ist; demzufolge sind Operationen mit Mengen von Argumentwerten (wie forall, union) **nicht** möglich!

Parameter:

- keine

Anmerkungen:

- Sehr schneller Zugriff auf die Map-Werte.
- Alle Operationen außer den folgenden erzeugen einen Laufzeitfehler: mk_empty ovwrt1 union1 sel sub1 ext
- In der Testphase eines Programms sollte eine andere Implementation (z.B. “heapmap“) benutzt werden, da die Operationen bestimmte Tests nicht durchführen.

heapmap

Datentyp(en):

- Token $\overrightarrow{m}$ Co-Domain

Repräsentation:

- Die Map-Werte werden auf dem Heap angelegt und durch Pointer dereferenziert. Die Sammlung der Pointer entspricht dem Datentyp „Token“, der hier als Set von Pointern implementiert wird. Dadurch sind im Gegensatz zur “heap“-Implementation alle Operationen vorhanden.

Parameter:

- **MACROS**: Operationen werden (wenn möglich) durch Makros realisiert, dadurch wird das Programm schneller, aber größer (Flag, Default: 0)
- **SAVE**: Prüfungen auf legale Zugriffe werden eingefügt (Flag, Default: 0)
- **ALLOC**: Speicherallokations-Strategie für den Heap (Default: „malloc“)
- **DOMIMPL**: Implementation des Set der Argumentwerte (Default: „linklist“)

Anmerkungen:

- Schneller Zugriff auf die Map-Werte — etwas langsamer als “heap“

linklist

Datentyp(en):

- Beliebige Elementtypen

Repräsentation:

- Die Paare werden in einer verketteten Liste gespeichert.

Parameter:

- **ALLOC**: Strategie der Speicherallokation (Default: „vmalloc“)

Anmerkungen:

- Keine Einschränkungen bei den Elementtypen.
- Langsam.

scaldom

Datentyp(en):

- Nat0 $\overrightarrow{m}$ Co-Domain
- Nat1 $\overrightarrow{m}$ Co-Domain
- Intg $\overrightarrow{m}$ Co-Domain
- Char $\overrightarrow{m}$ Co-Domain
- Unterbereich $\overrightarrow{m}$ Co-Domain

Repräsentation:

- [Co-Domain] $\star$
  Die Paare werden in ein Tuple eingetragen (die Tuple-Indizes entsprechen den Argumentwerten der Map). Ein nicht vorhandenes Paar wird durch den Eintrag **nil** im Tuple gekennzeichnet.

Parameter:

- **TUPLIMPL**: Implementation des zugrundeliegenden Tuple (Default: „array“)
- **OPTIMPL**: Implementation des Optionals (Default: „elemonil (NILVAL=-1)“)
- **LB_ELEM1**: Untergrenze des Argument-Domains (Default: für jeden Skalartyp ist eine Konstante „lb_xxxx“ definiert)

Anmerkungen:

- Der Argument-Domain muß eine feste Untergrenze haben. Bei den oben genannten Datentypen ist das erfüllt.
- Wird am häufigsten mit einem Unterbereich als Argument-Domain eingesetzt.
- Die Tuple-Implementation sollte „array“ sein, da der Element-Zugriff dann sehr schnell ist.
- Angabe der Untergrenze bei „Intg“ ist sinnvoll!
- Beispiel:
  DOMAINS
  Domain = Nat0 -m-> Nat1
  IMPLEMENTATIONS
  Domain = scaldom (OPTIMPL=„elemonil (NILVAL=0)“,lb_elem1 = 0)

OPTIONAL (opti)

elemonil

Datentyp(en):
- Beliebiger Elementtyp

Repräsentation:
- **nil** wird durch einen speziellen, vom Anwender zu bestimmenden Wert repräsentiert (für „Nat1" z.B. 0).

Parameter:
- **NILVAL**: Spezieller Wert für **nil** (kein Default, *muß* angegeben sein)
- **MACROS**: Operationen werden (wenn möglich) durch Makros realisiert, dadurch wird das Programm schneller, aber größer (Flag, Default: 0)
- **NOCHECK**: Keine Prüfung bei Selektions-Funktionen auf **nil** (Flag, Default: 0). Wenn MACROS = 1 ist, ist NOCHECK = 1

Anmerkungen:
- Sollte gewählt werden, wenn die Implementation „ptelem" nicht möglich ist (weil der NULL-Pointer einen gültigen Objekt-Zustand darstellt). Vorschlag: „elemonil (NILVAL= -1)"

ptelem

Datentyp(en):
- Elementtyp, der als Pointer implementiert ist

Repräsentation:
- Pointer oder **nil**. Der Element-Domain muß als Pointer implementiert sein, wobei der Wert **nil** durch den NULL-Pointer repräsentiert wird; dieser darf also keinen gültigen Zustand eines Objekts darstellen.

Parameter:
- **MACROS**: Operationen werden (wenn möglich) durch Makros realisiert, dadurch wird das Programm schneller, aber größer (Flag, Default: 0)
- **NOCHECK**: Keine Prüfung bei Selektions-Funktionen auf **nil** (Flag, Default: 0). Wenn MACROS = 1 ist, ist NOCHECK auch = 1

Anmerkungen:
- Kann nur mit Pointer-Implementationen der Elemente gewählt werden („ptunion", „ptstruct"). „linklist" ist z.B. nicht erlaubt, weil der NULL-Pointer den gültigen Wert „leeres Objekt" repräsentiert!

SET (set)

linklist

Datentyp(en):	• Beliebiger Elementtyp
Repräsentation:	• Verkettete Liste der Elemente (unsortiert).
Parameter:	• **ALLOC**: Strategie der Speicherallokation (Default: „vmalloc“)
Anmerkungen:	• Generelle Implementation
	• Im Moment gibt es keine Alternative

TREE (tree)

ptstruct

Datentyp(en):	• Beliebige Elementtypen
Repräsentation:	• Pointer auf C–**struct**
Parameter:	• **ALLOC**: Strategie der Speicherallokation (Default: „vmalloc“)
Anmerkungen:	• Brauchbar bei rekursiven ADT
	• Zugriff benötigt eine Pointer-Indirektion (langsamer als „struct“)
	• Zuweisung erfolgt durch Pointer-Zuweisung (schneller als „struct“)

struct

Datentyp(en):	• Beliebige Elementtypen
Repräsentation:	• C–**struct**
Parameter:	• keine
Anmerkungen:	• Zugriff benötigt keine Pointer-Indirektion (schneller als „ptstruct“)
	• Zuweisung erfolgt durch Kopieren (langsamer als „ptstruct“)

TUPLE (tupl)

array

Datentyp(en):
- Beliebige Elementtypen

Repräsentation:
- Variabel große Tabelle (beim Verändern der Größe wird eine neue Tabelle erzeugt, die alte umkopiert und gelöscht)

Parameter:
- **SIMPLEELEMS**: Ist der Elementtyp statisch (keine dynamischen Speicheroperationen mit Pointern), kann die Operation del beschleunigt werden (Flag, Default: 0)
- **ISCHAR**: Elementtyp ist "Char" — das Tuple repräsentiert einen String (Flag, Default: 0)
- **NOCHECK**: Keine Prüfung bei Selektions-Funktionen, ob der Index größer ist als die Länge des Tuple (Flag, Default: 0)
- **ALLOC**: Strategie der Speicherallokation (Default: „vmalloc")

Anmerkungen:
- Speicherallokation erfolgt in Einheiten von Zweierpotenzen
- Bei häufiger Veränderung der Größe viel Kopieraufwand (da das „array" allerdings in diskreten Größen alloziert wird, ist dies bei kleinen Änderungen in der Regel meist unnötig)
- Elementzugriffe sehr schnell
- Geeignete Implementation für Strings (bereits vordefiniert!)

linklist

Datentyp(en):
- Beliebiger Elementtyp

Repräsentation:
- Verkettete Liste der Elemente (unsortiert)

Parameter:
- keine

Anmerkungen:
- Generelle Implementation
- Bei Zugriffen sehr viel langsamer als „array", bei Veränderungen der Größe sehr viel schneller (die Größenänderung ist bei „array" allerdings nicht oft nötig)

UNION (unio)

ptunion

Datentyp(en):
- Beliebige Elementtypen

Repräsentation:
- Pointer auf *C*–**union** mit einem zusätzlichen „tag-field“ zur Erkennung des aktuellen Inhalts

Parameter:
- **MACROS**: Operationen werden (wenn möglich) durch Makros realisiert, dadurch wird das Programm schneller, aber größer (Flag, Default: 0)
- **NOCHECK**: Keine Prüfung bei Selektions-Funktionen auf korrekten aktuellen Inhalt — das „tag-field“ wird ignoriert (Flag, Default: 0). Wenn MACROS = 1 ist, ist NOCHECK auch = 1
- **ALLOC**: Strategie der Speicherallokation (Default: „vmalloc“)

Anmerkungen:
- Brauchbar bei rekursiven ADT
- Zugriff benötigt eine Pointer-Indirektion (langsamer als „union“)
- Zuweisung erfolgt durch Pointer-Zuweisung (schneller als „union“)

union

Datentyp(en):
- Beliebige Elementtypen

Repräsentation:
- *C*–**union** mit einem zusätzlichen „tag-field“ zur Erkennung des aktuellen Inhalts

Parameter:
- **MACROS**: Operationen werden (wenn möglich) durch Makros realisiert, dadurch wird das Programm schneller, aber größer (Flag, Default: 0)
- **NOCHECK**: Keine Prüfung bei Selektions-Funktionen auf korrekten aktuellen Inhalt — das „tag-field“ wird ignoriert (Flag, Default: 0). Wenn MACROS = 1 ist, ist NOCHECK = 1

Anmerkungen:
- Zugriff benötigt keine Pointer-Indirektion (schneller als „ptunion“)
- Zuweisung erfolgt durch Kopieren (langsamer als „ptunion“)

Anhang D. Der VDM–Compiler

Aufruf des VDM-Compilers

Der VDM-Compiler hat folgendes Aufruf-Format:

vdm [Optionen] *Eingabedatei(en)*

Sie können also mehrere DSL-Spezifikationen bzw. Enrichment-Dateien (die ja auch nur erweiterte DSL-Spezifikationen sind) gleichzeitig verarbeiten lassen, z.B. um die Spezifikation eines Typgenerators in einer Datei mit dem gewünschten Elementtyp in einer anderen Datei zu verbinden.

Übliche Aufrufe sind:

- Erzeugung einer Enrichment-Datei:
  vdm -v -e My_ADT.vdm -o My_ADT
- Erzeugung des ADT-Implementations-Moduls aus einer Enrichment-Datei:
  vdm -v -c -n My_ADT.enr -o My_ADT
- Erzeugung der Header-Datei aus einer Enrichment-Datei:
  vdm -v -h -n My_ADT.enr -o My_ADT
- Erzeugung des ADT-Implementations-Moduls aus einer DSL-Spezifikation:
  vdm -v -c My_ADT.vdm -o My_ADT
- Erzeugung der Header-Datei aus einer DSL-Spezifikation:
  vdm -v -h My_ADT.vdm -o My_ADT

Optionen des VDM-Compilers

Die Option –v

Der VDM–Compiler gibt seine momentane Tätigkeit zur Kontrolle am Bildschirm aus.

Die Option –S$Parameter[= def]$

Der VDM–Compiler setzt einen globalen, für *alle* Implementationen und Operationen gültigen Parameter.

Die Option –B*Lib-Name*

Der VDM–Compiler verwendet die angegebene Datei statt einer fest vorgegebenen Default-Datei als Quellcode-Bibliothek.

Die Option –o *Dateiname*

Damit können Sie den Hauptnamen der erzeugten Datei explizit bestimmen. Die Endung ergibt sich aus der erzeugten Datei.

Die Option –e

Der VDM-Compiler erstellt aus der angegebenen Datei die **Enrichment-Datei**, um die nachfolgenden Aufrufe des VDM–Compilers zu beschleunigen. Die Datei erhält die Endung **.enr** .

Die Option –n

Die angegebene Datei wird nicht „angereichert“, während das ADT-Implementations-Modul oder die Header-Datei erstellt wird — Sie sollten diese Option nur verwenden, wenn Sie bereits explizit die Enrichment-Datei (mit **–e**) erstellt haben.

Die Option –c

Der VDM-Compiler erstellt das ADT-Implementations-Modul. Die Datei erhält die Endung **.c** . Haben Sie die Enrichment-Datei mit **–e** explizit erstellt, sollten Sie zusätzlich **–n** angeben.

Die Option –h

Der VDM-Compiler erstellt die Header-Datei. Die Datei erhält die Endung **.h** . Haben Sie die Enrichment-Datei mit **–e** explizit erstellt, sollten Sie zusätzlich **–n** angeben.

Fehlermeldungen des VDM-Compilers

Die wichtigsten Fehlermeldungen sind hier (wenn möglich) in der Reihenfolge ihres Auftretens geordnet. Zunächst finden Sie die Fehler, die den Ablauf der Übersetzung betreffen, danach die Fehler, die in einer DSL-Spezifikation auftreten können.

Fehler beim Ablauf

! → **can't open** *Dateiname*

Die angegebene DSL-Spezifikations-Datei kann nicht geöffnet werden.

! → **–o: missing output name**

Sie haben zwar mit der Option **–o** angezeigt, daß Sie den Hauptnamen der zu erstellenden Datei selbst festlegen möchten, haben diesen aber nicht angegeben.

! → **vdm database** $DB - Name$ **not found**

Sie haben mit der Option **–B** eine VDM-Bibliothek gewählt, die es nicht gibt.

! → **unknown option** *–opt* **ignored**

Warnung: Sie haben eine unbekannte Option gewählt. Sie wird aber bei der Bearbeitung ignoriert.

Fehler in der DSL-Spezifikation

! → **syntax error near line** n
illegal domain equation oder **illegal implementation equation**

Syntaktischer Fehler in Ihrer DSL-Spezifikation.

! → **missing implementation description for** ADT

Sie haben eine Domain-Gleichung für einen ADT erstellt, ohne seine Implementation ebenfalls festzulegen.

! → **missing domain equation for** ADT

In Ihrer Implementationsbeschreibung kommt ein ADT vor, für den Sie keine Domain-Gleichung angegeben haben.

implementation *impl* **for constr** ADT **not found**

Die für einen ADT gewünschte Implementation gibt es nicht. Haben Sie versehentlich eine Implementation eines anderen VDM-Basistyps ausgewählt ?

! → **enrichment** *op* **for constr** ADT**, implementation** *impl* **not found**

Die angegebene Implementation für den ADT kennt die gewünschte Operation nicht.

! → **unknown flag** *par* **for domain** ADT**, enrichment** *op*

Sie haben bei einer Operation einen Parameter angegeben, der nicht definiert ist.

! → **dependency cycle detected**

Sie haben einen ADT rekursiv definiert, ohne dafür zu sorgen, daß einer der beteiligten VDM-Basistypen als Pointer implementiert ist (siehe Kapitel 6).

! → **inconsistent domain equations for** ADT **in files** $f1$ **and** $f2$

Sie bearbeiten mehrere DSL-Spezifikationen gleichzeitig, haben dabei aber den gleichen ADT auf unterschiedliche Weise spezifiziert. Prüfen Sie, ob Sie sich eventuell verschrieben haben.

! → **ambiguous implementation specification for domain** ADT

Sie bearbeiten mehrere DSL-Spezifikationen gleichzeitig, haben dabei den gleichen ADT mehrfach spezifiziert, aber unterschiedliche Angaben für dessen Implementation gemacht.

! $\rightarrow$ **inconsistent implem.param. for** ADT**,** $par1(par2)$ **in files** $f1$ **and** $f2$

Sie bearbeiten mehrere DSL-Spezifikationen gleichzeitig, haben dabei den gleichen ADT mehrfach korrekt spezifiziert, aber der Implementation unterschiedliche Parameter oder Parameter mit unterschiedlichem Inhalt mitgegeben.

! $\rightarrow$ **unknown domain constructor for** ADT
not implemented domain constructor for ADT

Sie haben mit Hilfe des Konstruktors $\mathcal{A} = \mathcal{B} - name$ einen Domain spezifiziert, ohne daß der Konstruktor $name$ in der VDM-Bibliothek existiert.

Glossar

Abstrakte Syntax: Die Sammlung aller Domain-Gleichungen eines Programms.

Abstraktion: Die Absonderung des Allgemeinen vom Einzelnen, die Begriffsbildung; auch das Ignorieren von unwichtigen Details, stattdessen Konzentration auf das Wesentliche.

ADT: Eine Klasse von Objekten, die vollständig und ausschließlich durch die Operationen, die mit den Objekten vorgenommen werden können, definiert ist.

ADT-Operation: Gelegentlich verwendet, zur Verdeutlichung der Tatsache, daß die Operationen den ADT zugeordnet sind und diese charakterisieren. Meist finden Sie nur den Begriff „Operation".

Applikation: Anwendungsprogramm, Anwendungs-System.

Argument-Domain: Argumentmenge einer Map (zugeordnet ist der Co-Domain).

Argumentwert: Im Zusammenhang mit einer Map der Teil eines Paares aus dem Argument-Domain; ihm ist der Map-Wert zugeordnet.

Attribut: Allgemein eine Eigenschaft eines Objekts — die zweiwertigen Eigenschaften werden allerdings Prädikate genannt.

Bag: Ein abstrakter Datentyp: ein Set, in dem jedes Element mehrfach vorkommen darf.

Binden: Das Zusammenfügen einzelner Objekt-Module zu einem vollständigen, ablauffähigen Programm.

C–String: Eine Zeichenkette wird in *C* üblicherweise als unbegrenztes Zeichen-Array dargestellt, an dessen Ende das Null-Zeichen steht, welches kein gültiges Zeichen repräsentiert. Die Länge des String ist nicht explizit gespeichert — sie kann durch Auszählen der Zeichen bis zum Null-Zeichen ermittelt werden.

Call By Reference: Eine Funktion kann den Inhalt eines ihrer Argumente dauerhaft ändern, da ihr die *Adresse* des Arguments übergeben wurde.

Call By Value: Einer Funktion wird nur der *Wert* eines Arguments übergeben; die Änderung des Arguments durch die Funktion ist nicht möglich.

Co-Domain: Zielmenge einer Map (wird dem Argument-Domain zugeordnet).

Codier-Phase: In dieser Phase der Programmentwicklung erfolgt die Umsetzung der Spezifikation, die in der Design-Phase entstanden ist, in die gewünschte(n) Programmiersprachen.

Datenabstraktion: Abstraktion, bezogen auf die Daten eines Programms und deren Manipulation: Absonderung der Eigenschaften, der „Bedeutung“ der Daten von ihrer physikalischen Realisierung — Beschreibung eines Objekts mit den Operationen, die mit ihm durchgeführt werden können.

Datenelemente: Daten, aus denen ein strukturierter Datentyp zusammengesetzt ist. Sie können wiederum strukturiert oder elementar sein.

Datentyp-Invariante: (auch „Bedingung für die Wohlgeformtheit“:) Grenzt die sinnvollen Zustände von Objekten einer Klasse von den sinnlosen, nach ihrer Syntax aber erlaubten Zuständen ab, ist also eine semantische Bedingung.

Design-Phase: In dieser, auch „Detailentwurf“ genannten Phase werden die Daten, Schnittstellen und Programm-Module genau spezifiziert und Testfälle definiert. Hier setzt VDM an.

Deskriptive Sprachen: Sprachen wie Prolog basieren auf der Definition des gewünschten Resultats einer Funktion *ohne* Angaben über den Algorithmus zur Gewinnung dieses Resultats.

Domain: Ein anderer Ausdruck für „abstrakter Datentyp“, auch dessen englischer Name im VDM-System.

Domain-Gleichung: Die syntaktische Beschreibung eines ADT, d.h. aus welchen Datenelementen er sich mit welcher Struktur zusammensetzt. Mit ihrer Hilfe lassen sich alle tatsächlichen Objekte erzeugen, zudem solche, die unsinnig sind. Diese müssen über semantische Bedingungen ausgegrenzt werden.

Elementarer Datentyp: Unteilbarer (atomarer) Datentyp einer Programmiersprache. In *C* sind dies nur die verschiedenen Zahlentypen und das Zeichen.

Elementary: Einige Datentypen sind auch in VDM vordefiniert: das Zeichen („Char“), verschiedene Zahlentypen und der logische Typ „Bool“.

Elementtyp: Der Datentyp eines Datenelementes in einem strukturierten Datentyp.

Enrichment: Angabe einer benötigten Operation für einen ADT in der DSL-Spezifikation. Der Begriff wird synonym mit „Operation“ verwendet.

Funktional: Funktion, deren Resultat eine Funktion ist. Als Argumente hat es weitere Funktionen, aus denen die resultierende Funktion konstruiert wird.

Funktionale Sprachen: Sprachen wie LISP basieren auf Funktionen und Rekursion, freie Variablen gibt es nicht.

Generischer Name: Name, der in dieser Form noch nicht komplett ist; aus ihm wird durch ein Suffix (oder Präfix) ein gültiger Name generiert.

Implizites Löschen: Bei Operationen, die ein strukturiertes Objekt so ändern, daß Elemente entfernt werden, ist es manchmal wünschenswert, daß diese Elemente dabei gleichzeitig gelöscht werden, da sie auch von anderen strukturierten Objekten nicht mehr referenziert werden; die Operationen sollen also diese Sub-Strukturen implizit löschen. Nähere Informationen finden Sie im Abschnitt 5.1.

Kombinator: Operation, die aus einzelnen Objekten ein neues Objekt ableitet.

Konstruktor: Operation, die ein neues Objekt eines Datentyps erzeugt und dabei meist auch eine Initialisierung vornimmt.

Korrektheitsbeweis: Formaler oder informaler Beweis der Korrektheit eines Programms. Ist die abstrakte Spezifikation als korrekt für die Lösung des Problems bewiesen und jede Konkretisierung als äquivalent zur vorhergehenden Abstraktion, ist auch die Implementation korrekt. Der Programmtest ist dann unnötig. In der Praxis ist der vollständige Beweis nicht möglich.

Map: Sie entspricht einer Funktion mit endlicher, variabler Argumentmenge: jedem Argumentwert ist ein „Map-Wert“ aus der Zielmenge, dem „Co-Domain“, zugeordnet.

Map-Wert: Im Zusammenhang mit einer Map der Teil eines Paares aus dem Co-Domain (analog zum Funktionswert bei Funktionen).

META-IV: ist die formale Sprache von VDM mit einer Kombination aus funktionalen und prozeduralen Sprachelementen. Der Name ist ein Homonym zum englischen „metaphor“. Eine Spezifikation mit META-IV kann recht einfach in eine „echte“ prozedurale Sprache übersetzt werden.

Modellorientierter Ansatz: siehe Abschnitt 2.3

Modularisierung: Aufteilung eines komplexen Programms oder Unterprogramms in eine Menge von einfacheren Unterprogrammen, deren Zusammenspiel die gleiche Funktion hat.

Namenserweiterung: Dies betrifft die *C*-Funktionen, die die Operationen der einzelnen VDM-Basistypen realisieren; sie müssen jeweils an die spezielle „Konfigurierung“ des VDM-Basistyps angepaßt werden (für einen Set von Integer-Werten werden andere Routinen benötigt als für einen Set von Tuples), woraus sich viele „Versionen“ der gleichen Operation ergeben können, die alle unterschiedliche Namen haben müssen. Daraus entsteht die Forderung, daß *alle* von Ihnen definierten Domains einen eigenen Namen haben müssen — die Operationsnamen werden dann automatisch mit diesem Domain-Namen „erweitert“, indem ihnen ein Unterstrich und der Domain-Name angehängt wird. Ausnahmen sind einzelne Tree-, Union- und Optional-Operationen, die komponentenbezogen sind; dann erfolgt die Erweiterung mit dem Namen des Komponententyps (eventuell doch gefolgt vom Domain-Namen) oder mit Teilen der Komponenten-Namen (bei Trees und Unions). Näheres finden Sie jeweils bei der Beschreibung dieser Typen.

nil: ist ein spezieller Zustand eines Objekts und bedeutet „kein gültiger Inhalt“.

Non-Terminal: Name für eine Symbolfolge einer Programmiersprache, die noch weiter in Terminale zerlegbar ist und meist ein Konzept dieser Sprache bezeichnet (z.B. Anweisung, Ausdruck, Bezeichner usw.)

Objekt-Modul: Bereits übersetztes Quellmodul, das aber allein meist nicht ablauffähig ist, da Funktionen oder Variablen verwendet werden, die in einem anderen Objekt-Modul definiert sind. Durch „Binden“ mehrerer Objekt-Module kann ein lauffähiges Programm entstehen.

Operation: In diesem Zusammenhang meist eine elementare Funktion zur Manipulation eines ADT und damit ein Charakteristikum dieses ADT.

Operationale Abstraktion: Im modellorientierten Ansatz die Spezifikation der Operationen und Funktionen, die auf Objekte eines ADT angewendet werden, unabhängig von ihrer Realisierung.

Optional: Es enthält entweder ein gültiges Objekt oder einen speziellen Wert **nil**, der anzeigt, daß das Optional im Moment *keinen* Inhalt hat.

Post-Kondition (postcondition): Im modellorientierten Ansatz die Bedingung, die nach einem erfolgreichen Aufruf einer Funktion erfüllt ist.

Prä-Kondition (precondition): Im modellorientierten Ansatz die Bedingung, die erfüllt sein muß, damit die Funktion aufgerufen werden darf.

Prädikat: Eigenschaft, die ein Objekt haben oder nicht haben kann — im Gegensatz zu Attributen, die mehrwertig sind. Auch die Bezeichnung für eine Operation, die eine solche Eigenschaft überprüft.

Programmentwicklung: Sie wird in mehrere Phasen unterteilt; die Hauptphasen sind Voruntersuchung, Analyse, Lösungskonzeption, Systementwurf, Detailentwurf (Design), Codierung, Test und Installation. Darauf folgen Programmpflege und Wartung in der Einsatzphase.

Prototyp: Die Deklaration einer Funktion ohne Definition ihres Rumpfes ermöglicht die Typprüfung bei ihrem Aufruf in einem anderen Modul; sie besteht aus Funktionsname, Parametertypen und Resultattyp.

Prozedurale Sprachen: Sprachen wie *C* basieren auf Anweisungen, veränderbaren Variablen und (bedingten und unbedingten) Verzweigungen zur Steuerung des Programmablaufes.

Quellcode-Bibliothek: Eine Modul-Bibliothek, die nicht aus vorcompilierten Objekt-Modulen besteht, sondern aus Quellmodulen.

Repräsentationale Abstraktion: Im modellorientierten Ansatz die Spezifikation von Klassen bzw. Objekten abstrakter Datentypen im Hinblick auf ihre Eigenschaften, aber unabhängig von ihrer Repräsentation.

Semantik: „Bedeutung" einer Sache, z.B. einer Funktion, d.h. „das, was die Sache tut" (im Gegensatz zur Syntax, ihrer formalen Beschreibung).

Set: Ein Set entspricht der mathematischen „Menge", ist also durch dieselben Eigenschaften und Operationen gekennzeichnet; sie ist eine endliche, ungeordnete Sammlung von unterschiedlichen Objekten des gleichen Typs.

Schrittweise Verfeinerung: siehe „Modularisierung".

Speicherallokationsstrategie: Die Methode, mit der der dynamisch verfügbare Speicher verwaltet (angefordert und freigegeben) wird.

String: In der Informatik gebräuchlicher Begriff für „Zeichenkette".

Strukturierter Datentyp: Eine strukturierte (geordnete, gegliederte) Sammlung von Datenelementen, z.B. Tabelle von Zahlen, Verbund mit Adreßdaten usw.

Switch-Funktion: Eine Funktion, die abhängig vom aktuellen Datentyp eines Union-Objekts verschiedene Verarbeitungsfunktionen aufruft.

Syntax: Im Gegensatz zur Semantik die formale Struktur einer Sache ohne inhaltliche Bedeutung.

Terminal-Symbol: Symbol (Zeichen oder Zeichenkette), die in einer Programmiersprache eine feste Bedeutung hat und nicht umdefiniert werden kann (Beispiele in *C*: **if**, **switch**, **void** usw.)

Tree: Ein Tree ist eine endliche, geordnete Sammlung verschiedenartiger Objekte und entspricht damit dem *struct* in der Sprache *C*.

Tuple: Es entspricht einer eindimensionalen Tabelle (in der Sprache *C* einem *array*): eine endliche, geordnete Sammlung gleichartiger Objekte, die nicht alle verschieden sein müssen.

Typgenerator: Jeder strukturierte Datentyp, dessen Elementtypen frei gewählt werden können, der also eine Klasse von Datentypen beschreibt, ist ein Typgenerator — er kann verschiedene Datentypen generieren.

Union: Sie enthält genau *ein* Objekt; zu verschiedenen Zeitpunkten kann der Datentyp des Objekts allerdings unterschiedlich sein.

VDM-Basistypen: Einige grundlegende, mächtige Typgeneratoren, die von VDM bereitgestellt werden: Set, Tuple, Tree, Map, Union, Optional.

VDM-Bibliothek: Die Quellcode-Bibliothek, die alle Funktionen für den Umgang mit den VDM-Basistypen enthält.

VDM-String: Die in fast allen Applikationen benötigte Manipulation von Strings ist in VDM mit einem „eigenen“ Datentyp möglich, der sich von den üblichen C-Strings unterscheidet und daher in diesem Handbuch zur Abgrenzung VDM-String genannt wird. Er ist als Tuple von Zeichen modelliert.

Virtuelle Speicherverwaltung: Das Betriebssystem ermöglicht den Austausch von Speicherseiten zwischen Hauptspeicher und Festplatte (Swapping); unbenutzte Seiten werden ausgelagert. Der Hauptspeicher wirkt dadurch fast beliebig groß. Ab einer bestimmten Auslastung muß ständig ausgetauscht werden, das System gerät in den „Thrashing“-Bereich, wo fast die gesamte Rechenzeit vom Betriebssystem beansprucht wird.

Wohlgeformtheit: Ein Objekt einer Klasse ist wohlgeformt, wenn es einen sinnvollen Zustand hat; ein sinnloser Zustand führt in der Regel zu einem Programmfehler.

Literaturhinweise

Die folgenden Bücher und Artikel können zur Vertiefung des Wissens über abstrakte Datentypen und VDM dienen.

Bjørner, Dines/ Jones, Cliff B., 1978: *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science Bd. 61, Berlin, Heidelberg, New York: Springer, 1978

Bjørner, Dines, et. al., 1987: *VDM '87: VDM – A Formal Method at Work*, Lecture Notes in Computer Science Bd. 252, Berlin, Heidelberg, New York: Springer, 1987

Bjørner, Dines/ Hoare, C.A.R./ Langmaack, Hans, 1990: *VDM '90: VDM and Z – Formal Methods in Software Development*, Lecture Notes in Computer Science Bd. 428, Berlin, Heidelberg, New York: Springer, 1990

Bloomfield, Robin/ Marshall, Lynn/ Jones, Roger, 1988: *VDM '88: VDM – The Way Ahead*, Lecture Notes in Computer Science Bd. 328, Berlin, Heidelberg, New York: Springer, 1988

Guttag, John/ Horowitz, Ellis/ Musser, David, 1976: *Abstract Data Types and Software Validation*, Los Angeles: Information Sciences Institute, University of Southern California, 1976

Henhapl, Wolfgang/ Letschert, Thomas, 1987: *VDM — Vienna Development Method* in: Informationstechnik it, 29(4), April 1987, S. 202–210

Liskov, Barbara/ Zilles, Stephen, 1974: *Programming with abstract data types*, in: ACM SIGPLAN Notices, 9(4), April 1974, S. 50–59

Schmidt, Uwe/ Völler, Reinhard, 1987: *Erfahrungen mit VDM bei der Übersetzungskonstruktion* in: Informationstechnik it, 29(4), April 1987, S. 211–216

Abbildungsverzeichnis

3.1 VDM in der Design-Phase 30
3.2 VDM in Detailentwurf und Codierung 32
3.3 VDM: Tätigkeit des VDM-Compilers 33
3.4 VDM: Tätigkeit des C-Compilers 35
3.5 Make-Datei für VDM 38
3.6 Make-Datei mit Erzeugung einer Enrichment-Datei 39

4.1 Beispiel für eine DSL-Spezifikation 46

5.1 Wirkung der Operationen **copy1** und **copy** 49
5.2 Beispiel für Sets: das „coarsest partitioning"-Problem 65
5.3 Beispiel für Tuples: das Pascal-Dreieck 82
5.4 Beispiel für Trees: der Binärbaum 112

8.1 Struktur der VDM-Bibliothek 143
8.2 ADT in der VDM-Bibliothek: Namen, Implementationen 146

Index

Abbildung (= Map) 16
Abstrakter Datentyp 6
ADT-Implementations-Modul **28**, 32, 179, 180
ADT 6
Argumentwert (Map) 16, **83**
Abstrakte Syntax **13**, 41
Abstraktion **5**, 101, 105, 122
Argument-Domain (Map) 16, **83**
Attribut 47

Bag (Datentyp) 129
Beispiel: Coarsest Partitioning 64
Beispiel: Heap-Speicherverwaltung 99
Beispiel: Stücklistenverwaltung 95, 129
Bool (Datentyp) 43, **53**
Bedeutung (Semantik) 10
Beispiel: Binärbaum 109
Beispiel: Pascal-Dreieck 80
Binärbaum **109**, 124

C-Compiler, Aufruf 35
Call by Reference 15
Co-Domain (Map) 16, **83**
Codier-Phase 31
C-Compiler 34
Char (Datentyp) 43, **52**
„Coarsest Partitioning" 64

DSL 25, 27, **41**, 143
Datenabstraktion 6
Datenorganisationsformen 125
Design-Phase 29
Domain 13
Dynamische Datenstrukturen **8**, 25, 39, 48
Datei-System 123
Datenelemente **7**, 9
Datentyp-Invariante 14, **24**, 96
Domain-Gleichung 13, 24, 27, 29, **41**

Elementarer Datentyp 7, 51
Enrichment 27, **44**
Elementary (Datentyp) 47, 142
Enrichment-Datei **33**, 179, 180

FH-Installation **13**, 16, 47
freelist (Speicherallokation) 167
Formale Sprache 9
Funktional 23, **60**

Geheimnisprinzip 6
Generischer Operationsname **26**, 44

Header-Datei **28**, 32, 179, 180
Hilfesystem 44, **139**
Heap-Speicherverwaltung 56, **99**

Implementation 6, 10, 25, **44**, 167
Implementation *bool* 54
Implementation *elemonil* 103
Implementation *heapmap* 86
Implementation *ptr* 56
Implementation *ptstruct* 107
Implementation *array* **69**, 76
Implementation *dir* **85**, 92
Implementation *heap* **85**, 87
Implementation *linklist* 58, 69, 85
Implementation *ptelem* 102
Implementation *ptunion* 115

Implementation *scalar* 53
Implementation *struct* 107
Implementation *unsigned* 52, 53
Implizite Spezifikation **10**, 19
Intg (Datentyp) 43, **53**
Implementation *scaldom* 85
Implementation *union* 115
Implementationsparameter 45
Implizites Löschen 50

Klasse (von Objekten) 13
Komponentenname 42
Konstruktive Spezifikation **10**, 19
Korrektheitsbeweis 2, **9**
Kombinator (Operation) 47
Konkretisierung 10
Konstruktor (Operation) 15, **47**

Liste (= Tuple) 16

*make*36
Make-Datei 36
Map-Wert 16, **83**
Menge (= Set) 16
Modellorientierter Ansatz **8**, 13, 16
Modularisierung 5
make, Aufruf 38
malloc (Speicherallokation) 167
Map (Datentyp) 16, 25, 42, 47, **83**, 142
META-IV 2, **29**
Modellspezifikation 13, 27

Namenserweiterung **27**, 45
Nat1 (Datentyp) 43, **53**
NULL-Pointer 101
Nat0 (Datentyp) 43, **53**
nil 17, 43, **101**

Objekt-Modul **35**, 37
Operationen (VDM-Basistypen) **7**, 10, 14, 25, 41, 44
Optional (Datentyp) 17, 25, 43, 47, **101**, 142
Operationale Abstraktion **10**, 15

Pascal-Dreieck 80
Post-Kondition 10
Prä-Kondition 10
precondition 10
Programmentwicklung 1, **29**
Performance 18, 26, 101, 167
postcondition **10**, 19
Prädikat **17**, 19, 22, 47
Produktion (*make*) 36

Quellcode-Bibliothek 26
Quellmodul 26, **35**

Rekursion 102, 107, 122
Repräsentationale Abstraktion **9**, 15
Rekursion, Implementation von 125

Schrittweise Verfeinerung **5**, 29
Semantik **10**, 19
Set (Datentyp) 16, 25, 42, 47, **57**, 142
Speicherverwaltung **8**, 25, 28
Stücklistenverwaltung 95, 129
Strukturierter Datentyp 7
Switch-Funktion (Union) 118
Syntax 10
Selektorname (Trees) 42
Semantische Funktionen **14**, 18, 27, 29
Speicherallokationsstrategie 25, **167**
Spezifikation 7
String **54**, 69
Str (Datentyp: VDM-String) 43, **54**, 75
Syntaktische Struktur 13

Token (Datentyp) 43, **55**, 99
Tree (Datentyp) 16, 25, 42, 47, **104**, 142

Tuple (Datentyp) 16, 25, 42, 47, **67**, 142
Typgenerator **7**, 8, 42

Union (Datentyp) 16, 25, 43, 47, 91, **113**, 142

Variante (= Union) 16
VDM, Vorteile 38
VDM-Bibliothek 26, 32, **139**
VDM-Compiler, Aufruf 34, **179**
VDM-Compiler, Optionen 33, **179**
VDM-String (Datentyp) **54**, 69, 76
Verbund (= Tree) 16
VDM, Historie 2
VDM-Basistypen **16**, 25
VDM-Compiler **32**, 41
VDM-Compiler, Fehlermeldungen 180
VDM-Europe 2
vdminfo 28, 44, **139**
Verkettete Liste **122**, 139

Wiederverwendbarkeit 3, **14**, 125
Wohlgeformtheit **14**, 17