de Gruyter Lehrbuch
van der Meulen · Kühling
Programmieren in ALGOL68 I

Programmieren in ALGOL68

I. Einführung in die Sprache

von

Sietse G. van der Meulen

und

Peter Kühling

Walter de Gruyter · Berlin · New York 1974

Sietse G. van der Meulen

Dozent und wiss. Mitarbeiter am Fachbereich Informatik der Universität Utrecht/Nederland,
Mitglied der IFIP-Working Group on ALGOL (W.G. 2.1)

Peter Kühling

Wiss. Assistent der Informatik-Forschungsgruppe Informationsverarbeitung II an der Technischen Universität Berlin

Satz: Walter de Gruyter, Berlin. Druck: Karl Gerike, Berlin.
Bindearbeiten: Dieter Mikolai, Berlin.
Library of Congress Catalog Card Number: 74-79157
ISBN 3 11 004698 9

Vorwort

Der vorliegende erste Band entstand aus Vorlesungen, die in den Wintersemestern 1972/73 und 1973/74 an der Technischen Universität Berlin und an der Universität Utrecht (Nederland) gehalten wurden.
Neben diesem ersten Band ist ein zweiter in Vorbereitung. Im ersten Band wird eine Einführung in die revidierte Sprache ALGOL68 gegeben, wie sie im "Revised Report on the Algorithmic Language ALGOL68" definiert worden ist. Dabei werden der Transput (Ein- und Ausgabe) und die Konstruktionen zur Parallelverarbeitung noch nicht berücksichtigt. Diese werden zusammen mit der Sprachdefinition und größeren Programmbeispielen für praktische Anwendungen Gegenstand des zweiten Bandes sein.

Dieser erste Band ist didaktisch geplant und dennoch systematisch angeordnet. Er besteht aus zwei Teilen:

In Teil I (Die 'ALGOL65-Stufe') findet man ungefähr alles, was man von ALGOL68 wissen muß, um vorhandene ALGOL60- (oder auch FORTRAN-, ALGOL W- usw.) Programme in ALGOL68 zu formulieren.

In Teil II (Bausteine der Orthogonalität) werden sehr eingehend die drei Hauptpfeiler von ALGOL68 behandelt:
die mode- und identifier-declaration,
die impliziten Anpassungsoperationen und die "united-modes" sowie
die Operator-Definitionen.

In dieser Aufteilung wird sich nicht nur der Anfänger zurechtfinden (vorausgesetzt, daß er mit ALGOL60 oder einer anderen maschinenunabhängigen Programmiersprache vertraut ist), sondern auch der Fachmann, der sich schnell orientieren will und genauer wissen möchte, welche Möglichkeiten ALGOL68 bietet.
Einige Beispiele haben wir dem Buch [4] "Informal Introduction to ALGOL68" von C. H. Lindsey und S. G. van der Meulen entnommen.

Für kritische Bemerkungen und Verbesserungsvorschläge sind wir den Kollegen W. Koch, C. H. A. Koster und E. Wegner sowie den vielen Studenten in Berlin und Utrecht sehr dankbar.

Ferner danken wir dem Verlag für gute und unkomplizierte Zusammenarbeit.

Berlin, im Juli 1974

S. G. van der Meulen
P. Kühling

Inhalt

Teil II Bausteine der Orthogonalität

0. Einführung

ALGOL68 ist eine Programmiersprache, die von anderen Programmiersprachen her bekannte Konzepte verallgemeinert und systematisiert. Darüberhinaus werden in ALGOL68 völlig neue Konzepte definiert.

Die Sprache dient dem Verständnis der Algorithmik:
sie trägt zur Klärung der Grundbegriffe bei,
ermöglicht die Formulierung der elementaren Konzepte und
gestattet die Vermittlung von Algorithmen.

Die Sprache ermöglicht außerdem eine effiziente Ausführung von Algorithmen auf einer Vielzahl verschiedener Rechner.

0.1 Historisches

Nach dem Erscheinen des "Report on the Algorithmic Language ALGOL" (1959) übergaben die Autoren dieses Reports die weitere Verantwortung für die Sprache an die "International Federation for Information Processing" (IFIP). Sie selbst gründeten innerhalb der IFIP die "Working-Group on ALGOL" (WG 2.1). Die erste Aufgabe dieser Gruppe bestand darin, einen "Revised Report on the Algorithmic Language ALGOL60" herauszugeben (1963).

In den Jahren 1960–1967 wurden, innerhalb und außerhalb der Working-Group, viele Anstrengungen unternommen, die Sprache ALGOL60 zu erweitern und zu verbessern. Dabei entstanden viele ALGOL-ähnliche Sprachen, die bekanntesten davon sind SIMULA67 und ALGOL W.

In der Working-Group wurden diese Sprachen und viele andere Vorschläge diskutiert. Als Ergebnis dieser Arbeit entstand in den Jahren 1966–1968 der "Report on the Algorithmic Language ALGOL68".

An vielen Orten wurde damit begonnen, ALGOL68 zu implementieren. Schon im Jahre 1971 fand eine dieser Implementationen ihren vorläufigen Abschluß (Royal Radar Establishment, Malvern: ALGOL68-R).

Von 1969–1973 wurden die Erfahrungen dieser Implementationen sowie sehr viele kritische Bemerkungen und Vorschläge eingehend diskutiert. Dies führte dann Ende 1973 zu dem endgültigen "Revised Report on the Algorithmic Language ALGOL68" (Los Angeles, September 1973).

In diesem Buch werden wir uns mit der durch diesen "Revised Report" definierten Sprache ALGOL68 beschäftigen.

Den "Revised Report" werden wir im folgenden kurz als Report (R) bezeichnen.

0.2 Entwurfsziele

Im Report wird im Abschnitt 0.1: "Aims and principles of design" zu den Entwurfszielen der Sprache ALGOL68 in einzelnen Punkten Stellung genommen. Wir wollen diese Punkte hier, sehr frei und subjektiv übersetzt, kurz anführen:

R0.1.1. Completeness and clarity of description

Der Report ist in der Tat erdrückend vollständig. Was die 'Klarheit' anbelangt, so spielen Fragen des unterschiedlichen Standpunktes (hier Benutzer, dort Compiler-Schreiber) sowie Fragen des unterschiedlichen Geschmacks (hier Praktiker, dort Theoretiker) des einzelnen eine wichtige Rolle. Allerdings, nicht in Frage gestellt werden kann die Genauigkeit des Reports.

R0.1.2. Orthogonal design

Ein Minimum an voneinander unabhängigen "primitive concepts" wurde entwickelt, die fast ohne Einschränkung beliebig miteinander kombiniert werden können. Hierin liegt das Neue und auch die Mächtigkeit der Sprache.

R0.1.3. Security

Die Sprache ist so beschaffen, daß die syntaktischen und viele andere Fehler so rechtzeitig erkannt werden, daß sie nicht zu katastrophalen Ergebnissen führen können. So werden unlogische Konstruktionen und insbesondere Flüchtigkeitsfehler sowie einfache Schreibfehler fast alle schon vom Compiler erkannt und entsprechend angezeigt.

R0.1.4. Efficiency

ALGOL68 folgt dem wichtigen Prinzip, daß ein Programmierer, der von aufwendigen Konstruktionen der Sprache keinen Gebrauch macht, auch nicht für die mögliche Ineffizienz dieser Sprachkonstruktionen (die er ja gar nicht benutzt) 'bezahlen' muß (Bauer-Samelson-Prinzip).

R0.1.4.1. Static mode-checking

"Mode" stellt die Verallgemeinerung des von ALGOL60 her bekannten Begriffs 'Typ' dar. Die Syntax von ALGOL68 erlaubt die Definition beliebig vieler "modes". Schon der Compiler kann den "mode" aller in einem Pro-

gramm auftretenden Größen feststellen. Die Ausführung ('execution') des Programms wird also damit nicht belastet.

Der **union**-"mode" stellt hierbei die einzige Ausnahme dar: der tatsächliche "mode" einer Größe kann hier erst zur 'runtime' festgestellt werden.

R0.1.4.2. Mode-independent parsing

Die im Report verwendete Zwei-Stufen-Grammatik erlaubt dem Compiler, ein Programm ohne Berücksichtigung der "modes" der in ihm vorkommenden Größen teilweise zu analysieren.

Verschiedene 'scans' ('passes') des Compilers können das Programm unter verschiedenen Aspekten betrachten, was einer verständlichen Fehler-Diagnose zugute kommt.

R0.1.4.3. Independent compilation

Angestrebt wurde eine Syntax, die es erlaubt, Hauptprogrammteile sowie einzelne Prozeduren unabhängig voneinander zu kompilieren. Dies kann ohne Effizienzverlust für das Endprodukt (Objektprogramm) geschehen.

R0.1.4.4. Loop optimization

Iterative Prozesse ('**for**-statements') sind derart in die Sprache eingebettet, daß auf sie bekannte Optimierungstechniken angewendet werden können.

R0.1.4.5. Representations

Die Repräsentation der in ALGOL68 vorkommenden "symbols" ist so gewählt, daß man mit einem minimalen Zeichenvorrat auskommen kann (48 Zeichen reichen schon aus). Gleichzeitig ist aber die Möglichkeit gegeben, einen größeren Zeichenvorrat zu benutzen, falls dieser zur Verfügung steht.

0.3 Notation

Für den laufenden (deutschen) Text in diesem Buch werden die Klein-Buchstaben

a b c d e f g h i j k l m n o p q r s t u v w x y z

und die Groß-Buchstaben

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

sowie die Ziffern

0 1 2 3 4 5 6 7 8 9

verwendet.

Darüberhinaus werden im Text noch die Zeichen

! ? . , ; : - ‘ ’ “ ” +

benutzt.

Begriffe aus der Terminologie des Reports werden (ohne Übersetzung) übernommen und durch “. . .” im laufenden Text gekennzeichnet. Im Gegensatz dazu geschieht die Hervorhebung bzw. Absetzung einzelner Wörter vom übrigen Text mit Hilfe von ‘. . .’ .

Sowohl im laufenden Text als auch in Programmbeispielen werden syntaktische Begriffe der Sprache ALGOL68 zur besseren Unterscheidung in den beiden Alphabeten

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

angegeben. Diese Begriffe sind im Report durch syntaktische Produktionsregeln formal definiert, so z. B.

identifier , symbol , indication , string , bold-tag

Man wird sich (hoffentlich) leicht an diese sogenannten NOTIONS gewöhnen.

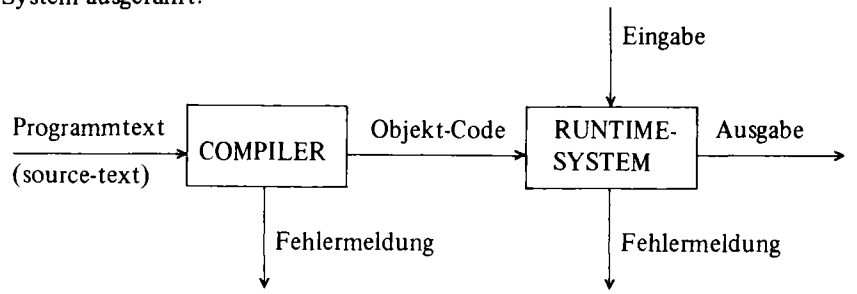
Teil I
Die 'ALGOL65-Stufe'

1. Grundbegriffe und Beispiele

1.1 Basis und Erweiterungsprinzip

1.1.1 Programmtext (Ausgangsbeispiel)

Wie bei vielen Programmiersprachen, so wird i. a. auch in ALGOL68 ein Programmtext zunächst von einem Compiler in einen maschinenabhängigen Objekt-Code übersetzt. Dabei gibt der Compiler Fehlermeldungen aus, wenn der Programmtext syntaktisch nicht korrekt ist. Ist der Programmtext syntaktisch in Ordnung, so wird der Objekt-Code von einem 'runtime'-System ausgeführt:



Nach der 'compilation' (Übersetzung) folgt also die 'execution' (Ausführung). Dabei werden Daten eingegeben, auf welchen das Objekt-Programm operiert. Die Ergebnisse der 'execution' werden vom 'runtime'-System ausgegeben (z. B. auf einem Schnelldrucker ausgedruckt). Stößt das 'runtime'-System während der 'execution' auf Mißstände (z. B. Division durch Null), so kann es passende Fehlermeldungen ausgeben.

Im Abschnitt 1.2 wird beschrieben, wie eine idealisierte 'ALGOL68-Maschine' einen Programmtext interpretiert. Es wird dabei nicht zwischen einem Compiler und einem 'runtime'-System unterschieden: die 'ALGOL68-Maschine' umfaßt beides.

Zur "representation" von ALGOL68 (zur Darstellung von Programmtexten in ALGOL68) benutzen wir u. a. zwei Zeichenmengen:

1) Das Alphabet
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   und die Ziffern
   0 1 2 3 4 5 6 7 8 9

2) Das bold-Alphabet
a b c d e f g h i j k l m n o p q r s t u v w x y z
und die bold-Ziffern
0 1 2 3 4 5 6 7 8 9

Mit Hilfe der Zeichen jeweils einer Menge können Wörter gebildet werden. Dabei ist ein Wort über der Menge 1) oder 2) eine freiwählbare Folge von Zeichen, die stets mit einem Buchstaben beginnt, z. B.

word , **word** , *word1* , **word1** , *w0rd* , **w0rd**

nicht aber

1word , **1word**

Ein Wort über der Zeichenmenge 1) wird als ein identifier bezeichnet, z. B.

x , *y* , *z* , *sum* , *squm* , *next* , *a1* , *a2*

In identifiers dürfen 'blanks' vorkommen:

this is one identifier , *this is another one*

Ein Wort über der Zeichenmenge 2) wird als ein bold-tag bezeichnet. In einem bold-tag dürfen keine 'blanks' vorkommen:

thisisoneboldtag , **these are five bold tags**

Bold-tags werden für die Repräsentation von symbols und indications verwendet. Symbols sind 'reserved-words', Wörter also mit einer festen Bedeutung, die nicht für andere Zwecke verwendet werden können, so z. B.

begin , **end** , **if** , **then** , **else** , **fi** , **case** , **in** , **out** , **esac**

Indications sind (wie auch die identifiers) Wörter, deren Bedeutung man selbst in seinem Programm definieren (deklarieren) oder neu definieren kann.

So sind z. B. die indications

round , **entier** , **abs** , **repr** , **plusab**

schon standardmäßig deklariert worden; wer will, kann sie jedoch in seinem Programm neu definieren.

Alle noch nicht deklarierten indications muß man selbst definieren:

man , **woman** , **human** , **amode** , **amode1** , **amode2**

Wir geben nun einen Überblick über alle symbols (oder auch tokens), die in ALGOL68 eine feste Bedeutung haben, die der Programmierer in seinem Pro-

gramm also nicht ändern kann. Die symbols:

+ - × /
< ≤ > ≥ ↑ ↓

kommen in dieser Liste nicht vor. Das ist kein Versehen; denn selbstverständlich hat man auch in ALGOL68 diese Zeichen mit standardmäßiger Bedeutung zur Verfügung. Jedoch kann man sie selbst neu definieren oder ihnen eine weitere Bedeutung beigeben. Diese Zeichen gehören also nicht in eine Liste von symbols, deren Bedeutung von der Verwendung im Programmtext unabhängig ist.

ALGOL68-symbols mit fester Bedeutung:

( ) [ ] " • 10 : $ | |:

= (is-defined-as-token)
:= (becomes-token)
:=: (is-token) **is** :≠: (is-not-token) **isnt**

; (go-on-token) , (and-also-token)
empty true false

declaration-symbols:

long short ref loc heap struct flex proc union op prio mode int real bool char format void compl bits bytes string sema file

syntactic-symbols:

begin end exit par if then else elif fi case in ouse out esac at of goto go to skip
nil for from by to while do od

pragment-symbols:

comment co ¢ # pragmat pr

In ALGOL68 hat der Programmierer die Möglichkeit, einen erläuternden Kommentar an fast jeder Stelle innerhalb seines Programms einzufügen. Jedoch in identifiers, bold-tags und denotations (vgl. weiter unten und 1.1.2) ist dies nicht erlaubt. Dabei wird ein Kommentar eingeschlossen in

	comment	und	**comment**
oder	**co**	und	**co**
oder	¢	und	¢
oder	#	und	#

Naiv betrachtet, besteht ein Programmtext in ALGOL68 stets aus einer Folge von symbols (tokens), denotations und indicators:

- Die symbols (tokens) sind oben angegeben.
- Die denotations dienen zur Bezeichnung von Werten ("values") und können aufgebaut werden aus Ziffern, Buchstaben, gewissen bold-tags und den symbols " · $_{10}$ *e* $
  Auch denotations haben eine feste Bedeutung, die man nicht ändern kann.
- Wir unterscheiden drei Arten von indicators:
  1) identifiers,
  2) operator-indications
     zur Repräsentation von Operatoren, für die kein geeignetes symbol zur Verfügung steht,
  3) mode-indications

Außer für die standardmäßig vorgegebenen mode-indications

void , bool , int , real , char , compl , bits , bytes , string

kann der Programmierer die Bedeutung für indicators (also nicht nur identifiers, sondern auch operator-indications und mode-indications) selbst bestimmen. Dies geschieht durch declarations.

Feste Bedeutung haben also nur die symbols und denotations sowie die oben angegebenen standard-mode-indications. Alle anderen Zeichen und 'Wörter' (indicators) kann man selbst (neu) definieren.

Betrachten wir als erstes Beispiel für einen Programmtext ein einfaches Programm, mit Hilfe dessen zwei Zahlen gelesen und sodann ihr Mittelwert sowie ihre Standard-Abweichung ausgedruckt werden. Selbstverständlich wird solch ein Programm erst dann sinnvoll, wenn mehrere Zahlen gelesen und behandelt werden (vgl. dazu 1.4.2 und 1.4.3).

```
begin   co mean and standard-deviation of two real numbers co
        real x, y, sum, squm, sd;
        read(x); read(y);
        sum  := x + y ;
        squm := x ↑ 2 + y ↑ 2 ;
        sd   := sqrt (2 × squm - sum ↑ 2)/2 ;
        print (sum/2) ; print (sd)
end
```

Hierbei erhalten die identifiers *x*, *y*, *sum*, *squm* und *sd* ihre Bedeutung durch die declaration

real *x, y, sum, squm, sd*

An diesem Beispiel (und auch an den interessanteren Beispielen in 1.4) werden wir in den folgenden Paragraphen einige Grundbegriffe und Prinzipien der Sprache ALGOL68 kennenlernen.

1.1.2 Die Grundmengen

ALGOL68 kann man mit einem LEGO-Spiel vergleichen: Als Basis sind Bausteine verschiedener Form und gewisse Anbaumöglichkeiten fest vorgegeben. Es gibt nur wenige, voneinander verschiedene Formen von Bausteinen, und die Anbaumöglichkeiten sind unabhängig von der Form.

In ALGOL68 sind die fest vorgegebenen Bausteine Werte ("values") von nur wenigen verschiedenen Typen. Diese Werte werden also aufgeteilt in eine kleine Anzahl von Grundmengen. Die Grundmengen heißen:

boolean , character ,
integral , real ,
format .

Wir werden später sehen, daß man die leere Menge (**void**) auch noch als eine 'Grundmenge' betrachten kann: eine Menge also, die keine Werte enthält.

Die Elemente der Grundmengen (also: die Grundwerte) können im Programm durch denotations bezeichnet werden, wie z. B.

empty ,
true , *"a"* ,
37 , *3.1415926536* ,
$ *+ d. 13 de 1 d* $

und die Mengen selbst durch mode-indications:

void ,
bool , **char** ,
int , **real** ,
format .

Die Grundmengen, ihre indications und die Bezeichnung (denotation) ihrer Werte sind in der nachfolgenden Tabelle angegeben.

mode-indications	"values"	denotations
void	leer	**empty**
bool	die Menge, bestehend aus den beiden logischen Werten 'wahr' und 'falsch'	**true** **false**
char	eine Mindestmenge von Zeichen: ein Alphabet von Buchstaben, die Ziffern *0* bis *9*, einige Sonderzeichen; abhängig von der jeweiligen Implementierung sind eventuell noch andere Zeichen (z. B. ein weiteres Alphabet) vorhanden	*"a" "b" "c" "d" ...* *... "x" "y" "z"* *"0" "1" "2" ... "9"* *"." "," "+" "-" "="* *"(" ")" ":" ...* *"A" "B" "C" "D" ...* *... "X" "Y" "Z"* *"/" "?" "!" "x" "&"* *"$_{10}$" ";" "%" ...*
int	ein Teilbereich der ganzen Zahlen (dessen Grenzen implementationsabhängig gegeben sind durch *max int*)	*0 1 2 3 4 5 6 7 8 9* * *10 11 12 13 14 ...* *100 101 ...* *... 67108863*
real	eine Teilmenge der reellen Zahlen, gegeben in Gleitpunkt-Annäherung; die kleinste positive reelle Zahl, die man noch mit Effekt zu *1.0* addieren kann, ist gegeben durch *small real*; die größte darstellbare reelle Zahl durch *max real* (beide sind implementationsabhängig).	*3.14159265* * *$0.314159265_{10}1$* *$0.000314159265_{10}4$* *31415.9265_{10} - 4* *314159265_{10} - 8* *0.0 1.0 0.1* *$10_{10}10$ 1_{10} - 100* *0.314159265e1* *314159265e - 8*
format	eine Menge von 'strings' mit eigener Syntax für die Beschreibung von Ein- und Ausgabedaten.	$ + d. 13 de 1 d $ $ "dm" 8 d. 2 d $

* Für die negativen Werte gibt es keine denotations, man erhält sie aus den entsprechenden positiven Werten durch Anwendung des monadischen Operators '-'.

1.1.3 Der Begriff "orthogonal-design"

Mit Hilfe der wenigen (in 1.1.2 genannten) Grundbausteine kann man größere Bau-Elemente bilden, z. B.

Reihen (eine gewisse Anzahl numerierter Elemente eines bestimmten Typs),

Strukturen (ein Gebilde von Elementen, möglicherweise verschiedenen Typs),

Routinen (mit oder ohne Parameter versehene Programmteile, die dann nach ihrer Ausführung einen bestimmten Baustein liefern),

Adressen (Werte, die sich auf andere Bausteine beziehen), etc.

Aus diesen so zusammengesetzten Bausteinen können wiederum 'höhere' Bausteine gebildet werden (Reihen von Strukturen, Strukturen mit Reihen als Komponenten, Reihen von Routinen, Routinen, die jeweils eine Reihe oder ein Struktur liefern, etc.). Dieses Erweiterungsprinzip kann man beliebig oft und in jeder Kombination anwenden.

Die Verallgemeinerung des ALGOL60-Begriffs Typ heißt in ALGOL68: "mode". Einen gewünschten neuen "mode" kann man aus den Grund-"modes" (**bool**, **char**, **int**, **real** und **format**) mit Hilfe von gewissen symbols (die wir oft als 'mode-maker' bezeichnen werden) bilden.

'Mode-maker' sind u. a.:

[ ]	zur Konstruktion einer Reihe,
struct	zur Konstruktion einer Struktur,
long	zur Konstruktion der Ausdehnung bestimmter Mengen (u. a. **int**, **real**, **compl**)
short	zur Konstruktion der Einschränkung bestimmter Mengen (u. a. **int**, **real**, **compl**)
ref	zur Konstruktion einer Adresse,
proc	zur Konstruktion einer Routine,
union	zur Konstruktion einer Vereinigung mehrerer "modes"

Die "modes" (also: die Mengen) werden dabei durch sogenannte declarers spezifiziert. So spezifiziert z. B.:

[*l* : *n*] **real**

den "mode" row-of-real, d. i. der "mode" einer Reihe von **reals**.

Eine neue mode-indication (für die Bezeichnung einer Menge) kann man durch eine mode-declaration definieren: rechts von dem is-defined-as-token ' = ' steht der declarer.

Beispiele:

```
mode boolean = bool
mode logical = bool
```

definieren **boolean** oder **logical** als alternative mode-indications für **bool**.

```
mode vector = [1 : n] real
mode matrix = [1 : n, 1 : n] real
```

definieren **vector** und **matrix** als ein- bzw. zweidimensionale Reihe von **reals**.

Die mode-indications **compl** und **string** sind standardmäßig in der sogenannten standard-prelude definiert. Dort wird durch

```
mode compl = struct (real re, im)
```

compl als eine Struktur definiert, deren zwei **real**-Komponenten man mit den field-selectors *re* und *im* auswählen kann. Dagegen ist **string** als eine **char**-Reihe mit flexiblen Grenzen definiert (vgl. 3.1.2).

```
mode giant = long int
```

definiert **giant** als den "mode" von ganzen Zahlen aus einer Menge, welche die der **int**s umfaßt.

```
mode man   = struct (string name, int year,
                     ref woman wife),
mode woman = struct (string name, int year,
                     ref man husband,
                     ref [ ] human child ),
mode human = union (man, woman)
```

definieren ein naives Modell der Menschheit, wobei die elementare Tatsache, daß eine Frau Kinder gebären kann, ein Mann jedoch nicht, zum Ausdruck gebracht worden ist. Ein **human** ist entweder ein **man** oder eine **woman**.

```
mode fun = proc (real) real
```

definiert **fun** als eine reelle Funktion auf der Menge der reellen Zahlen.

```
mode matmat = proc ([,] real) [,] real
```

definiert **matmat** als eine Funktion, die Matrizen in Matrizen überführt; oder auch:

```
mode matmat = proc (matrix) matrix
```

Eine mode-indication kann man als eine Abkürzung für einen zusammengesetzten declarer ansehen, den man jedoch auch selbst im Programm benutzen kann. Allerdings sind mode-declarations notwendig, wenn "modes" miteinander verknüpft sind, wie es bei **man**, **woman** und **human** der Fall ist.