Variabel lange Zeichenketten in Fortran90

1. Einleitung

1.1 Problematik von variabel langen Zeichenketten

Bekanntermassen kennt der Standard fuer ANSI FORTRAN 77 /1/ nur Daten vom Typ CHARACTER mit fester Laenge. Es hat sich in der Praxis aber herausgestellt, dass die Behandlung von variabel langen Zeichenketten haeufig benoetigt wird, aber im Standard ANSI FORTRAN 77 recht muehsam ist.
Typischerweise muessen alle Zeichenketten in einen maximal langen Speicher abgelegt werden, und von Fall zu Fall muss das jeweilige Ende der Zeichenkette (z.B. das letzte nicht leere Zeichen) gesucht werden.

Manche Compiler(z.B. der Compiler FOR1 des Siemens-Nixdorf-Betriebssystems BS2000) hatten darum die Sprache Fortran um den Datentyp CHARACTER VARYING erweitert. Daten vom Typ CHARACTER VARYING enthalten immer den Inhalt der letzten Zuweisung. Insbesondere wird auch bei Ausgabe nur der zuletzt zugewiesene Text transferiert. Zur Unterstuetzung dieses neuen Datentyps wurden die Operationen und Funktionen, die der Standard fuer Zeichenketten vom Typ CHARACTER vorsieht, durch Erweiterungen in Compiler und Laufzeitsystem auf den Datentyp CHARACTER VARYING ausgedehnt.
Der Datentyp CHARACTER VARYING erweist sich in der Praxis als sehr nuetzlich.

1.2 Enstehung des String-Moduls

Es gab darum schon frueh den Vorschlag, den Datentyp CHARACTER VARYING als neuen Datentyp in den Standard fuer Fortran90 aufzunehmen. Trotz Forderungen von verschiedenen Seiten ist dies aber nicht geschehen. Grund dafuer war im wesentlichen die Meinung, dass der Fortran90-Standard nicht mit diesem zusaetzlichen Datentyp belastet werden sollte, zumal die Funktionalitaet von CHARACTER VARYING ja mit Standard-Mitteln von Fortran90 (mit wenig Komfort und einigen Einschraenkungen) abzudecken sei.

Da aber gerade im deutschsprachigen Raum der Wunsch nach Unterstuetzung variabel langer Zeichenketten besonders stark war, wollte DIN den neuen Standard nur akzeptieren, wenn wenigstens eine begrenzte Unterstuetzung variabel langer Zeichenketten in Fortran90 vorhanden waere. Der Weg, der mit Unterstuetzung von ISO/IEC JTC1/SC22 WG5 Working Group, welche sich mit Normungsfragen der Sprache Fortran befasst, schliesslich beschritten wurde, bestand in der Definition eines geeigneten Standard-Moduls /3/ der unter Federfuehrung von J.L Schonfelder unter Mitarbeit des DIN-Arbeitskreises Fortran, in den Jahren 1989 bis 1992 entworfen wurde. Das entsprechende Dokument liegt jetzt als sogenannter 'Committe Draft' vor.

Das Dokument enthaelt erstens die Spezifikation von Schnittstellen fuer die Bearbeitung von variabel langen Zeichenketten, mit den Mitteln des Fortran90 Standards.

Zweitens enthaelt das Dokument eine Prototyp-Implementierung eines Modules, das die vorgegebene Spezifikation erfuellt.

Im folgenden wird der Modul der Kuerze halber 'String-Modul' und der in ihm definierte Datentyp fuer variabel lange Zeichenketten 'String' genannt.

1.3 Absicht des kollateralen Standards

Das oben erwaehnte Dokument wird zur Zeit mit dem Ziel oeffentlich diskutiert, es als 'zu Fortran90 kollateralen Standard' zu verabschieden.

Die Zielsetzung dieses vorgeschlagenen kollateralen Standards ist es, das Schreiben von portablen Programmen zu ermoeglichen, die Zeichenketten variabler Laenge verarbeiten. Ein standard-konformes Fortran90-System muss nicht die Schnittstellen des kollateralen Standards anbieten, da es eine gueltige Fortran90-(Prototyp)-Implementierung gibt. Aber wenn ein Fortran90-System Unterstuetzung zur Bearbeitung von variabel langen Zeichenketten anbietet, z.B. durch speziell fuer das jeweilige System optimierte Funktionen, so soll es sich an die Spezifikation des kollateralen Standards halten. In diesem Sinne garantiert der kollaterale Standard die Portabilitaet solcher Benutzerprogramme.

Es sei darauf hingewiesen, dass ausschliesslich die Schnittstellen zum Benutzer Gegenstand der Standardisierung durch den kollateralen Standard sind. Die Interna der Realisierung, z. B. die interne Struktur des Datentyps , der die variabel langen Zeichenketten darstellt, die Art der Implementierung der Funktionen und Operatoren, die verwendeten Algorithmen, die internen Namen der Funktionen, sind vom kollateralen Standard ausdruecklich offen gelassen worden. Damit ist die Moeglichkeit gegeben, einen Modul zu erstellen, der fuer ein spezielles System voellig andere, optimierte Algorithmen verwendet, und doch standardkonform ist.

Die Prototyp-Implementierung soll somit nicht verbindlich werden. Ihre Bedeutung liegt jedoch in folgendem :

Sie zeigt, dass die Implementierung durch reine Fortran90-Mittel moeglich ist, so dass ein neuer intrinsischer Datentyp CHARACTER VARYING nicht notwendig ist.

Sie eroeffnet die Moeglichkeit, die Brauchbarkeit der definierten Schnittstellen zu testen.

Sie liefert ein Beispiel, das im Default-Falle oder als Grundlage fuer eine eigene Implementierung verwendet werden kann.

Schliesslich bietet das entstandene Modul ein schoenes Beispiel, die Maechtigkeit der neuen Fortran90-Sprachmittel zu demonstrieren.

2. Die Schnittstellen des String-Moduls

2.1 Einleitung

In diesem Kapitel sollen die zu normierenden Schnittstellen des String-Moduls dargestellt werden. Dabei sollen die verwendeten Fortran90-Konzepte (Modulkonzept, Overloading) deutlich gemacht werden.
Zur besseren Anschaulichkeit werden am Beispiel des Assign-Operators die Fortran90-Konzepte verdeutlicht.

2.2 Spracherweiterungen durch den String-Modul

Der kollaterale Standard schreibt fuer den String-Modul den Namen

ISO_VARYING_STRING

vor. Wenn ein Fortran90-Anwender mit der Anweisung

USE ISO_VARYING_STRING

den Zugriff auf den String-Modul ermöglicht, stehen ihm über den Fortran90-Standard hinaus folgende Sprachmittel zur Verfügung:

2.2.1 Der Typ VARYING_STRING

Mit diesem Typ ist es möglich, variabel lange Zeichenketten im Fortran-Programm zu definieren.
Im Gegensatz zu variablen Zeichentypen in anderen Programmiersprachen (z.B PL/1, Ada, FOR1 CHARACTER VARYING) ist es dabei nicht notwendig, eine Maximallänge zu spezifizieren, bis zu der die aktuelle Zeichenlänge variieren kann. Die Maximallänge wird lediglich durch die Größe des jeweiligen Rechners bestimmt.
Es handelt sich also um einen String-Typ mit echt dynamischer Länge.

2.2.2 Operatoren fuer Datentypen VARYING_STRING

Die folgenden Fortran90-Operatoren können auch auf Operanden des Typs VARYING_STRING angewendet werden:

Zuweisung :	=
Konkatenation :	//
Vergleiche :	==
	/=
	<
	<=
	>
	>=

Es sind auch Kombinationen von Operanden der Typen VARYING_STRING und CHARACTER zugelassen.
Aus Konsistenzgründen werden für VARYING_STRING die gleichen Operatoren wie für CHARACTER verwendet, diese Operationen werden mit Hilfe des Fortran90-Overloading-Konzepts organisch erweitert.

2.2.3 Intrinsic-Funktionen des Fortran90-Standards mit erweiterter Semantik

Die folgenden generischen Fortran90-Intrinsic-Prozeduren sind semantisch erweitert worden und können auch auf Operanden des Typs VARYING_STRING angewendet werden:

LEN	Länge des Strings
ICHAR	interne Darstellung des Zeichens(Prozessor-abhängig)
IACHAR	interne Darstellung des Zeichens(ASCII)
TRIM	beseitigt nachfolgende Leerzeichen
LEN_TRIM	Länge ohne nachfolgende Leerzeichen
ADJUSTL	linksbündig ausrichten
ADJUSTR	rechtsbündig ausrichten
REPEAT	wiederholte Verkettung
LLT	
LLE	Vergleichsfunktionen
LGE	
LGT	
INDEX	Position einer Teilzeichenfolge
SCAN	Position eines von mehreren Zeichen
VERIFY	Position eines Zeichens, das nicht in einer Menge von Zeichen ist

Für die Vergleichsfunktionen sind auch Kombinationen von Operanden der Typen VARYING_STRING und CHARACTER zugelassen. Wie die Operatoren werden die CHARACTER-Funktionen durch die Overloading-Technik konsistent erweitert.

2.2.4 Konversionsfunktionen

Zusätzlich stehen die generischen Intrinsic-Funktionen

VAR_STR
CHAR

zur Verfügung, die CHARACTER-Argumente fester Laenge in den Typ VARYING_STRING konvertieren, bzw. umgekehrt.

2.2.5 Funktionen zur Ein-/Ausgabe

Für die Ein-/Ausgabe von VARYING_STRING-Daten gibt es zusätzliche generische Prozeduren:

READ_STRING	Lesen eines Satzes (oder eines Satzteils) in einen String
WRITE_STRING	Schreiben eines Strings ans Ende eines Satzes
WRITE_LINE	Schreiben eines Strings ans Ende eines Satzes und Beenden des Satzes

Anders als bei CHARACTER-Operationen und CHARACTER-Intrinsic-Funktionen ist bei den Ein-/Ausgabe-Prozeduren keine konsistente Erweiterung auf VARYING_STRING möglich.
Dies wird weiter unten noch diskutiert.

Die VARYING_STRING-Ein-/Ausgabefunktionen lassen folgende Argumente zu:

READ_STRING:	unit	Eingabe-Unit, Standard-Eingabe, falls Argument fehlt
	string	String, in den eingelesen wird vom Typ VARYING_STRING
	maxlen	Maximalzahl der einzulesenden Zeichen
	set	Menge der Zeichen, die Leseabbruch hervorruft
	iostat	Status-Variable des Datentransfers
WRITE_STRING, WRITE_LINE:	unit	Ausgabe-Unit, Standard-Ausgabe, falls Argument fehlt
	string	String, der ausgegeben werden soll, dieser kann sowohl vom Typ VARYING_STRING als auch vom Typ CHARACTER sein
	iostat	Status-Variable des Datentransfers

2.2.6 Neue Funktionen zur SUBSTRING-Behandlung

Für die Substring-Manipulationen von VARYING_STRINGs stehen folgende zusätzliche generische Prozeduren zur Verfügung:

INSERT	Substring einfügen
REPLACE	Substring ersetzen
REMOVE	Teil eines Strings löschen
EXTRACT	Teil eines Strings extrahieren

Folgende Argumente sind in diesen Prozeduren zugelassen:

INSERT:	string	String, in den eingefügt wird, vom Typ VARYING_STRING
	start	Position der Einfügung
	substring	einzufügender Substring : dieser kann sowohl vom Typ VARYING_STRING als auch vom Typ CHARACTER sein
REPLACE:	string	String, in dem ersetzt wird, vom Typ VARYING_STRING
	start	Startposition der Ersetzung
	finish	Endeposition der Ersetzung
	substring	einzufügender Substring : dieser kann sowohl vom Typ VARYING_STRING als auch vom Typ CHARACTER sein
REMOVE:	string	String, in dem gelöscht wird, vom Typ VARYING_STRING
	start	Startposition der Löschung
	finish	Endeposition der Löschung
EXTRACT:	string	String, aus dem extrahiert wird, vom Typ VARYING_STRING
	start	Startposition der Extraktion
	finish	Endeposition der Extraktion
	Das Funktionsergebnis ist der extrahierte String, vom Typ VARYING_STRING.	

2.3 Nutzung des Fortran90-Overloading

Wie oben gesehen werden Operatoren und Funktionen (soweit dies sinnvoll ist) so definiert, dass sie sowohl fuer Daten vom Typ VARYING_STRING als auch gemischt mit Daten des Typs CHARACTER verwendet werden koennen. Das heisst, die fuer den Benutzer sichtbaren Namen sind generische Namen.

Das Operator- und Funktionsoverloading von Fortran90 wird hier genutzt, um Funktionen und Operatoren generischer Natur zu realisieren.

Der String-Modul enthaelt INTERFACE-Bloecke fuer die generischen Funktionen und Operatoren. Nur deren Namen sind fuer den Benutzer sichtbar. Der aktuelle Aufruf einer generischen Funktion oder die Anwendung eines solchen Operators fuehren dann zum Aufruf der im Modul ISO_VARYING_STRING jeweils zustaendigen versteckten lokalen Funktion, deren Namen im INTERFACE-Block spezifiziert ist. Diese lokale Funktion ist jeweils die spezialisierte Funktion fuer eine jeweilige Kombination von Argumenttypen, mit speziellem Namen.

2.4 Struktur des String-Moduls

Anhang I zeigt schematisch die Struktur des String-Moduls.
Eine vollstaendige Liste ist in Ref. /3/ zu finden.

Die generelle Anweisung PRIVATE bewirkt, dass die Voreinstellung fuer alle Deklarationen PRIVATE, also nicht fuer den Benutzer sichtbar, ist.

In einer expliziten PUBLIC-Anweisung sind die Namen der nach aussen sichtbaren generischen Funktionen enthalten.

Die Definition des Datentyps VARYING_STRING selbst ist explizit PUBLIC, aber seine interne Komponente ist ausdruecklich als PRIVATE deklariert.
Dies entspricht der Absicht des vorgeschlagenen kollateralen Standards, keine Interna der Realisierung offenzulegen, um die Portabilitaet der Benutzerprogramme zu gewaehrleisten.

Die generischen Namen in den INTERFACE-Bloecken fuer das Funktions- und Operator-Overloading sind PUBLIC, die internen Funktionen aber sind wieder PRIVATE.

Anhang I enthaelt auch den Code fuer die Assign-Funktionen und fuer eine I/O Funktion.
Das Beispiel der Assign-Funktion und der I/O-Funktion zeigt, dass bei der expliziten Codierung natuerlich Interna des Datentyps VARYING_STRING verwendet werden, naemlich die Tatsache, dass der Datentyp intern einen CHARACTER-Array namens chars mit Attribut Pointer enthaelt.
Diese Interna sollen natuerlich nicht standardisiert werden, und entsprechend werden auch die Algorithmen dieser Funktionen nicht durch den Standard vorgeschrieben. Die Funktionen koennten auch anders codiert und gegebenenfalls fuer ein vorgegebenes System optimiert werden. Die zugrundeliegende Problematik wird im folgenden diskutiert.
Bemerkenswert an der Assign-Funktion ist, dass sie eine explizite ALLOCATE-Anweisung fuer die Variable der linken Seite enthaelt, aber keine DEALLOCATE-Anweisung.

2.5 Speicherverwaltung fuer VARYING_STRING-Daten

Bei der Diskussion der Assign-Funktion haben wir bemerkt, dass fuer die Variable der linken Seite mit ALLOCATE explizit Speicher angelegt wird, in den der neue Text hineingeschrieben wird, dass aber kein Speicher freigegeben wird.
Die oeffentliche Diskussion der Prototyp-Implementierung hat gezeigt, dass gar keine andere Implementierung fuer einen wirklich dynamisch anzulegenden Datentyp VARYING_STRING moeglich ist als eine, bei der in der Assign-Funktion des String-Moduls einmal angeforderter Speicher nicht wieder freigegeben wird.
Dies liegt daran, dass innerhalb der Assign-Funktion nicht entschieden werden kann, ob es sich um die erste Zuweisung an die Variable handelt oder nicht. Falls der Variablen noch kein Wert zugewiesen wurde, ist ihr Status undefiniert, und gemaess Fortran90-Standard ist die Wirkung einer DEALLOCATE-Anweisung nicht vorhersehbar. Damit kann innerhalb der Assign-Funktion keine DEALLOCATE-Anweisung erfolgen. Nicht mehr benutzter Speicher sammelt sich an.

Dieser Defekt ist auch erkannt worden. Man kann ihn durch die Forderung loesen, dass ein Fortran90 System eine Garbage Collection haben soll, so dass bei Speicher-Ueberlauf nicht mehr benoetigter Speicher wiederverwendet wird. Der Fortran90-Standard fordert aber solch eine Garbage Collection nicht, und die meisten Systeme werden sie wohl auch nicht haben.

Eine weitere Loesung waere, zu fordern, dass der Benutzer selber vor der ersten Zuweisungan an eine String-Variable diese explizit initialisieren muss, wodurch z.B. einen Minimalspeicher zugewiesen wird. Tatsaechlich wird ueber eine Erweiterung der Benutzer-schnittstellen nachgedacht, die benutzereigene Speicherverwaltung erlauben.
Die Notwendigkeit, jede Variable vor Benutzung explizit zu initialisieren, waere aber wohl ein Nachteil des Datentyps VARYING_STRING gegenueber dem frueheren Datentyp CHARACTER VARYING.

2.6 Diskussion der Funktionen zur Ein-/Ausgabe

Fuer Ein-/Ausgabe von Daten des Typs VARYING_STRING sind einige zusaetzliche Funktionen definiert worden, die fuer den CHARACTER-Datentyp nicht existieren.
Der Grund ist, dass Standard CHARACTER Variable vom Fortran-Laufzeitsystem unterstuetzt werden durch den Format-Bezeichner A. Die Ausgabe eines beliebigen benutzerdefinierten Datentyps, (und der Datentyp VARYING_STRING ist hinsichtlich des Fortran90 Standards ein solcher), wird aber nicht unterstuetzt. Standard Fortran kennt keinen Format-Bezeichner zur Ausgabe eines Datums vom Typ VARYING_STRING.

Ohne Kenntnis der internen Struktur des Datentyps VARYING_STRING kann der Benutzer noch nicht einmal eigene Ausgabefunktionen erstellen. Diese Kenntnis darf er aber nicht verwenden, wenn er die Portabilitaet seiner Programme sicherstellen will.
Man hat sich darum entschlossen, spezielle Ein-/Ausgabefunktionen zu definieren, und sie (soweit sinnvoll) auch fuer Standard CHARACTER-Variable zu erweitern.

3. Anwendungsbeispiel

Das Beispiel in Anhang II soll demonstrieren, wie man in einem Fortran90-Programm mit variabel langen Zeichenketten arbeiten kann. Es ist dem Committee Draft /3/ ueber das String-Modul entnommen.

In dem Programm wird ein in einer Datei gespeicherter Text analysiert und eine Liste der vorkommenden Wörter sowie deren Auftrittshäufigkeit erstellt.
Zu bemerken ist, daß dieses Beispiel nicht in erster Linie eine besonders effiziente Implementierung der gestellten Aufgabe bieten soll, sondern den Komfort, den der Datentyp VARYING_STRING und die Funktionen des String-Moduls bieten, bei der Lösung der Aufgabe zeigen soll.

Das Anwendungsprogramm ist selbst ein Modul mit einem Hauptprogramm und zwei internen Prozeduren.

Das Hauptprogramm enthaelt zwei geschachtelte Schleifen.
In der aeusseren wird jeweils eine variabel lange Zeile gelesen.
In der inneren wird diese analysiert. Mit SCAN wird ein Wort, separiert durch entweder "," oder "." oder "!" oder "?" , gesucht.
Mit EXTRACT und REMOVE wird es aus der Zeile herausgeloest.

Dann wird es in den Array vocab umgespeichert und seine Haeufigkeit gezeahlt. Dies geschieht innerhalb der internen Prozedur update_vocab_lists.

Falls in letzterer Prozedur die Grenze der Listen vocab und freq erreicht ist, werden diese Listen verlaengert durch den Prozeduraufruf extend_lists. extend_lists enthaelt die automatischen Arrays vocab_swap und freq_swap. Diese werden bei jedem Eingang in extend_lists mit der aktuellen Laenge von list_size angelegt. list_size ist eine host-assoziierte Variable, die jeweils neue Werte annimmt.

In die Arrays vocab_swap und freq_swap werden die Inhalte von vocab und freq mit einfachen Array-Zuweisungen zwischengespeichert. vocab und freq werden dann deallokiert und mit neuer Laenge allokiert. Zuletzt werden die vorigen Inhalte aus den automatischen Arrays vocab_swap und freq_swap mittels Array-Zuweisungen wieder restauriert.

In dem Beispiel finden folgende String-Modul-Bestandteile Verwendung:

```
VARYING_STRING
=  (Zuweisung)
== (Vergleich)
READ_STRING
WRITE_LINE
CHAR
LEN
SCAN
EXTRACT
REMOVE
```

Das Beispiel zeigt, dass sich die Verarbeitung der neuen Datentypen VARYING_STRING harmonisch in die Fortran90-Sprache einfuegt. Durch die Erweiterung der Semantik von bekannten Funktionen und Operatoren ist die String-Verarbeitung in aehnlich kompakter und selbsterklaerender Weise moeglich wie bei Standard-Datentypen.

Das Beispiel zeigt auch, dass die Schnittstelle zum String-Modul ISO_VARYING_STRING trotz der vollen Unterstuetzung des neuen Datentyps schmal ist : es entfallen langwierige private Deklarationen, und die Funktionen und Operatoren sind vertraut, da sie sich syntaktisch und semantisch eng an die Standard-Funktionen des Fortran90-Standards anlehnen.
Schliesslich zeigt das Beispiel den Komfort von einer Menge neuer Fortran90 Sprachmittel (Benutzer-Module, Interne Prozeduren, Automatische Datentypen, Allocatable Arrays, Arrayzuweisungen).

3. Status des String-Moduls

Das String-Modul hat, wie schon erwaehnt, den Status eines Committee Draft der ISO/IEC JTC1/SC22 WG5 Working Group. Es ist zur Zeit wieder in einer 'public comment' Phase mit dem Ziel, es als einen zu Fortran90 kollateralen Standard zu verabschieden.

Diskussionspunkte sind unter anderem die Punkte, die hier erwaehnt wurden, z.B. das Problem der Speicherverwaltung. Der aktuelle Vorschlag des DIN-Arbeitskreises fuer Fortran geht dahin, zusaetzliche Schnittstellen zur Speicherverwaltung durch den Benutzer zu definieren.

Neben ueberwiegender Zustimmung gibt es allerdings auch die Meinung, dass die Schnittstellen nicht standardisiert werden sollten, da sie noch nicht genuegend erprobt seien.

Die ueberwiegende Mehrheit der Kritiker und auch der DIN-Arbeitskreis sind aber der Ansicht, dass die Schnittstellen sinnvoll, hilfreich fuer portable Programmierung und komfortabel sind, und dass die Schnittstellen nach moeglicherweise leichter technischer Ueberarbeitung standardisiert werden sollten.

Dr. Thomas Kahl
Siemens-Nixdorf AG.
8000 Muenchen 83
Otto-Hahn-Ring 8

Literaturhinweise

/1/ American National Standard Programming Language FORTRAN
ANSI X3.9-1978

/2/ American National Standard Programming Language FORTRAN
ANSI X3.198-1991

Fortran 90
ISO/IEC 1539:1991, May 1991

/3/ International Standards Organization
Varying Length Character Strings in Fortran
ISO/IEC 1539-1
(collateral standard to ISO/IEC 1539:1991)
(Draft 8-Mar-92)

ANHANG I

Schema der Prototypimplementierung des Moduls
zur Bearbeitung von Zeichenketten variabler Laenge

```
MODULE ISO_VARYING_STRING

! Written by J.L.Schonfelder
! Incorporating suggestions by C.Tanasescu, C.Weber, J.Wagener and W.Walter,
! and corrections due to L.Moss, M.Cohen, P.Griffiths, B.T.Smith
! and many other members of the committee ISO/IEC JTC1/SC22/WG5

! Version produced (10-Mar-92)

!-------------------------------------------------------------------------!
! This module defines the interface and one possible implementation for a !
! dynamic length character string facility in Fortran 90. The Fortran 90  !
! language is defined by the standard ISO/IEC 1539 : 1991.                !
! The publicly accessible interface defined by this module is conformant  !
! with the collateral standard, ISO/IEC 1539-1 : 1993.                    !
! The detailed implementation may be considered as an informal definition of !
! the required semantics, and may also be used as a guide to the production !
! of a portable implementation.                                           !
! N.B. Although every care has been taken to produce valid Fortran code in !
!      construction of this module no guarantee is given or implied that this !
!      code will work correctly without error on any specific processor.  !
!-------------------------------------------------------------------------!

PRIVATE

!-------------------------------------------------------------------------!
! By default all entities declared or defined in this module are private to !
! the module. Only those entities declared explicitly as being public are !
! accessible to programs using the module. In particular, the procedures and !
! operators defined herein are made accessible via their generic identifiers !
! only; their specific names are private.                                 !
!-------------------------------------------------------------------------!

TYPE VARYING_STRING
 PRIVATE
 CHARACTER,DIMENSION(:),POINTER :: chars
ENDTYPE VARYING_STRING
```

```
!-------------------------------------------------------------------------!
! The representation chosen for this definition of the module is of a string !
! type consisting of a single component that is a pointer to a rank one array !
! of characters.                                                          !
! Note: this Module is defined only for characters of default kind. A similar !
! module could be defined for non-default characters if these are supported !
! on a processor by adding a KIND parameter to the component in the type  !
! definition, and to all delarations of objects of CHARACTER type.        !
!-------------------------------------------------------------------------!

CHARACTER,PARAMETER :: blank = " "
INTEGER,PARAMETER   :: ichar0 = ICHAR("0")

!----- GENERIC PROCEDURE INTERFACE DEFINITIONS ----------------------------!

!----- LEN interface ------------------------------------------------------!
INTERFACE LEN
  MODULE PROCEDURE len_s   ! length of string
ENDINTERFACE

     .
     .
     .

!----- ASSIGNMENT interfaces ----------------------------------------------!
INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE s_ass_s, &   ! string    = string
                   c_ass_s, &   ! character = string
                   s_ass_c      ! string    = character
ENDINTERFACE

!------ Equality comparison operator interfaces-----------------------------!
INTERFACE OPERATOR(==)
  MODULE PROCEDURE s_eq_s, &  ! string==string
                   s_eq_c, &  ! string==character
                   c_eq_s     ! character==string
ENDINTERFACE

     .
     .
     .

!----- Output procedure interfaces ----------------------------------------!
INTERFACE WRITE_LINE
  MODULE PROCEDURE putline_d_s, & ! string to default unit
                   putline_u_s, & ! string to specified unit
                   putline_d_c, & ! char to default unit
                   putline_u_c    ! char to specified unit
ENDINTERFACE

     .
     .
     .
```

```
!----- specification of publically accessible entities ----------------------!
PUBLIC ::
          VARYING_STRING,VAR_STR,CHAR,LEN,READ_STRING,WRITE_STRING,          &
          WRITE_LINE,INSERT,REPLACE,REMOVE,                                  &
          REPEAT,EXTRACT,INDEX,SCAN,VERIFY,LLT,LLE,LGE,LGT,ASSIGNMENT(=),    &
          OPERATOR(//),OPERATOR(==),OPERATOR(/=),OPERATOR(<),OPERATOR(<=),   &
          OPERATOR(>=),OPERATOR(>),LEN_TRIM,TRIM,IACHAR,ICHAR,ADJUSTL,ADJUSTR

CONTAINS

!----- LEN Procedure ---------------------------------------------------------!
 FUNCTION len_s(string)
  type(VARYING_STRING),INTENT(IN) :: string
  INTEGER                         :: len_s
  ! returns the length of the string argument or zero if there is no current
  ! string value
  IF(.NOT.ASSOCIATED(string%chars))THEN
    len_s = 0
  ELSE
    len_s = SIZE(string%chars)
  ENDIF
 ENDFUNCTION len_s

     .
     .
     .

!----- ASSIGNMENT Procedures -------------------------------------------------!
 SUBROUTINE s_ass_s(var,expr)
  type(VARYING_STRING),INTENT(OUT) :: var
  type(VARYING_STRING),INTENT(IN)  :: expr
  !  assign a string value to a string variable overriding default assignement
  !  reallocates string variable to size of string value and copies characters
  ALLOCATE(var%chars(1:LEN(expr)))
  var%chars = expr%chars
 ENDSUBROUTINE s_ass_s

 SUBROUTINE c_ass_s(var,expr)
  CHARACTER(LEN=*),INTENT(OUT)    :: var
  type(VARYING_STRING),INTENT(IN) :: expr
  ! assign a string value to a character variable
  ! if the string is longer than the character truncate the string on the right
  ! if the string is shorter the character is blank padded on the right
  INTEGER                         :: lc,ls
  lc = LEN(var); ls = MIN(LEN(expr),lc)
  DO i = 1,ls
   var(i:i) = expr%chars(i)
  ENDDO
  DO i = ls+1,lc
   var(i:i) = blank
  ENDDO
 ENDSUBROUTINE c_ass_s
```

```
SUBROUTINE s_ass_c(var,expr)
 type(VARYING_STRING),INTENT(OUT) :: var
 CHARACTER(LEN=*),INTENT(IN)      :: expr
 !  assign a character value to a string variable
 !  disassociates the string variable from its current value, allocates new
 !  space to hold the characters and copies them from the character value
 !  into this space.
 INTEGER                          :: lc
 lc = LEN(expr)
 ALLOCATE(var%chars(1:lc))
 DO i = 1,lc
   var%chars(i) = expr(i:i)
 ENDDO
ENDSUBROUTINE s_ass_c

    .
    .
    .

!------ Equality comparison operators ----------------------------------------!
 FUNCTION s_eq_s(string_a,string_b)  ! string==string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL                         :: s_eq_s
  INTEGER                         :: la,lb
  la = LEN(string_a); lb = LEN(string_b)
  IF (la > lb) THEN
     s_eq_s = ALL(string_a%chars(1:lb) == string_b%chars) .AND. &
              ALL(string_a%chars(lb+1:la) == blank)
  ELSEIF (la < lb) THEN
     s_eq_s = ALL(string_a%chars == string_b%chars(1:la)) .AND. &
              ALL(blank == string_b%chars(la+1:lb))
  ELSE
     s_eq_s = ALL(string_a%chars == string_b%chars)
  ENDIF
 ENDFUNCTION s_eq_s

 FUNCTION s_eq_c(string_a,string_b)  ! string==character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*),INTENT(IN)     :: string_b
  LOGICAL                         :: s_eq_c
  INTEGER                         :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) /= string_b(i:i) )THEN
      s_eq_c = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. ANY( string_a%chars(lb+1:la) /= blank ) )THEN
    s_eq_c = .FALSE.; RETURN
  ELSEIF( la < lb .AND. blank /= string_b(la+1:lb) )THEN
    s_eq_c = .FALSE.; RETURN
  ENDIF
  s_eq_c = .TRUE.
 ENDFUNCTION s_eq_c
```

```
 FUNCTION c_eq_s(string_a,string_b)  ! character==string
  CHARACTER(LEN=*),INTENT(IN)      :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL                          :: c_eq_s
  INTEGER                          :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) /= string_b%chars(i) )THEN
      c_eq_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank )THEN
    c_eq_s = .FALSE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) )THEN
    c_eq_s = .FALSE.; RETURN
  ENDIF
  c_eq_s = .TRUE.
 ENDFUNCTION c_eq_s

      .
      .
      .

!----- Output string procedures ---------------------------------------------!
SUBROUTINE putline_d_s(string,iostat)
 type(VARYING_STRING),INTENT(IN) :: string
                                    ! the string variable to be appended to
                                    ! the current record or to the start of
                                    ! the next record if there is no
                                    ! current record
                                    ! uses the default unit
 INTEGER,INTENT(OUT),OPTIONAL    :: iostat
                                    ! if present used to return the status
                                    ! of the data transfer
                                    ! if absent errors cause termination
! appends the string to the current record and then ends the record
! leaves the file positioned after the record just completed which then
! becomes the previous and last record in the file.
 INTEGER           :: ist
  WRITE(*,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
  IF( ist /= 0 )THEN
   IF(PRESENT(iostat))THEN
    iostat = ist; RETURN
   ELSE
    WRITE(*,*) " Error No.",ist, &
               " during WRITE_LINE of varying string on default unit"
    STOP
   ENDIF
  ENDIF
 IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE putline_d_s


      .
      .
      .

ENDMODULE ISO_VARYING_STRING
```

ANHANG II

Anwendungsbeispiel eines Programms zur Bearbeitung
von Daten von Typ VARYING_STRING

```
PROGRAM vocabulary_word_count
!-------------------------------------------------------------------------------!
! Counts the number of "words" contained in a file. The words are assumed to    !
! be terminated by any one of:                                                  !
! space,comma,period,!,? or the EoR                                             !
! The file may have records of any length and the file may contain any number   !
! of records.                                                                   !
! The program prompts for the name of the file to be subject to a word count    !
! and the result is written to the default output unit                          !
! Also builds a list of the vocabulary found and the frequency of occurance     !
! of each different word.                                                       !
!-------------------------------------------------------------------------------!
USE ISO_VARYING_STRING
type(VARYING_STRING) :: line,word,fname
INTEGER              :: ierr,nd,wcount=0
!-------------------------------------------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will     !
! be extended dynamically in steps of 100 as the used vocabulary grows          !
!-------------------------------------------------------------------------------!
type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab
INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq
INTEGER                                       :: list_size=100,list_top=0
!-------------------------------------------------------------------------------!
! Initialize the lists and determine the file to be processed                   !
!-------------------------------------------------------------------------------!
ALLOCATE(vocab(1:list_size),freq(1:list_size))
WRITE(*,ADVANCE='NO',FMT='(A)') " Input name of file?"
CALL READ_STRING(STRING=fname)  ! read the required filename from the default
                                ! input unit assumed to be the whole of the
                                ! record read
OPEN(UNIT=1,FILE=CHAR(fname))   ! CHAR(fname) converts to the type
                                ! required by FILE= specifier
file_read: DO ! until EoF reached
  CALL READ_STRING(1,line,ierr)       ! read next line of file
  IF(ierr /= 0) EXIT file_read
  word_scan: DO ! until end of line
    nd=SCAN(line," ,.!?")  ! scan to find end of word
    IF(nd == 0) THEN  ! EoR is end of word
      nd = LEN(line)
      EXIT word_scan
    ENDIF
    IF(nd > 1) THEN  ! at least one non-terminator character in the word
      wcount=wcount+1
      word = EXTRACT(line,1,nd-1)
      CALL update_vocab_lists
    ENDIF
    CALL REMOVE(line,1,nd)  ! strips the counted word and its terminator
                            ! from the line reducing its length before
                            ! rescanning for the next word
  ENDDO word_scan
```

```
  IF(nd > 0) THEN  ! at least one character in the word
    wcount=wcount+1
    word = EXTRACT(line,1,nd-1)
  ENDIF
ENDDO file_read

IF(ierr < 0) THEN
  WRITE(*,*) "No. of words in file =",wcount
  WRITE(*,*) "There are ",list_top," distict words"
  WRITE(*,*) "with the following frequencies of occurance"
  print_loop: DO i=1,list_top
    WRITE(*,FMT='(1X,I6)',ADVANCE='NO') freq(i)
    CALL WRITE_LINE(STRING=vocab(i))
  ENDDO print_loop
ELSE
  WRITE(*,*) "Error in READ_STRING in vocabulary_word_count ",ierr
ENDIF

CONTAINS

SUBROUTINE extend_lists
!-------------------------------------------------------------------------!
! Accesses the host variables:                                            !
!  type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab                 !
!  INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq                  !
!  INTEGER                                       :: list_size             !
! so as to extend the size of the lists preserving the existing vocabulary !
! and frequency information in the new extended lists                     !
!-------------------------------------------------------------------------!
type(VARYING_STRING),DIMENSION(list_size) :: vocab_swap
INTEGER,DIMENSION(list_size)              :: freq_swap
INTEGER,PARAMETER :: list_increment=100
INTEGER           :: new_list_size,alerr
vocab_swap = vocab  ! copy old list into temporary space
freq_swap = freq
new_list_size = list_size + list_increment
DEALLOCATE(vocab,freq)
ALLOCATE(vocab(1:new_list_size),freq(1:new_list_size),STAT=alerr)
IF(alerr /= 0) THEN
  WRITE(*,*) "Unable to extend vocabulary list"
  STOP
ENDIF
vocab(1:list_size) = vocab_swap  ! copy old list back into bottom
freq(1:list_size) = freq_swap    ! of new extended list
list_size = new_list_size
ENDSUBROUTINE extend_lists
```

```
SUBROUTINE update_vocab_lists
!-------------------------------------------------------------------------------!
! Accesses the host variables:                                                  !
!  type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab                       !
!  INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq                        !
!  INTEGER                                       :: list_size,list_top          !
!  type(VARYING_STRING)                          :: word                        !
! searches the existing words in vocab to find a match for word                 !
! if found increments the freq if not found adds word to                        !
! list_top + 1 vocab list and sets corresponding freq to 1                      !
! if list_size exceeded extend the list size before updating                    !
!-------------------------------------------------------------------------------!
list_search: DO i=1,list_top
  IF(word == vocab(i)) THEN
    freq(i) = freq(i) + 1
    RETURN
  ENDIF
ENDDO list_search
IF(list_top == list_size) THEN
  CALL extend_lists
ENDIF
list_top = list_top + 1
vocab(list_top) = word
freq(list_top) = 1
ENDSUBROUTINE update_vocab_lists

ENDPROGRAM vocabulary_word_count
```