

KAPITEL 6

MENGEN- UND PROZEDURTYPEN

6.1. Mengentypen

Mengen sind (nicht nur) in der Mathematik definiert als Gesamtheiten gleichartiger oder verschiedener Grundeinheiten, für die keine Ordnung festgelegt ist, die jedoch (prinzipiell) abzählbar sind. Diese Definition ist in Component Pascal weitgehend erhalten geblieben, allerdings gibt es die Einschränkung, daß die Grundeinheiten gleichartig sein müssen, es sich bei ihnen also um Elemente eines einzigen Grundtyps, des Typs *INTEGER*, handeln muß. Menge heißt im Englischen *set*, und die Component Pascal Mengentypbezeichnung lautet folglich *SET*. In Bezug auf die Größe des Typs gibt es eine weitere Einschränkung, die den maximalen Umfang, man sagt dazu auch die Mächtigkeit, einer Menge betrifft. Die Mächtigkeit des Typs *SET* beträgt 32 Elemente. Dies scheint im ersten Moment eine (zu) starke Einschränkung darzustellen, aber diese Einschränkung läßt sich leicht durch Aneinanderreihung mehrerer Mengen aufheben.

Sehen Sie sich das Programm *Mengen.odc* an. Sie finden am Anfang der Prozedur *Start* die Deklaration mehrerer Variablen mit dem Typ *SET* und einer Variablen des Typs *INTEGER*. Im Anweisungsteil finden Sie nach der Ausgabe einer Überschrift als Neuigkeit in den Zuweisungen von Anfangswerten an die Mengenvariablen *Ziffern* und *Zwiffern* Paare von geschweiften Klammern { }, die wie in der Mathematik eine Menge repräsentieren. Zwischen den Mengenklammern werden die in der Menge enthaltenen Elemente in gleicher Weise notiert, wie Sie dies bei der CASE-Abfrage kennengelernt haben, entweder durch explizite Angabe der Elemente, die durch Kommata voneinander getrennt werden, oder durch Angabe eines zusammenhängenden Bereichs, wie Sie es bei der Zuweisung *Ziffern := {0..10}* sehen. Sind die Mengenklammern leer, so ist damit die leere Menge bezeichnet, also eine Menge, die kein Element enthält. Sie finden eine derartige Menge in der Zuweisung *Zwiffern := {}*. Die nächsten beiden Zeilen machen Sie mit zwei nur für Mengen existierenden Component Pascal Operatoren bekannt, *INCL* und *EXCL*, die von den englischen Wörtern *include* und *exclude* abgeleitet sind und das ihren Namen Entsprechende tun, sie fügen der in den Klammern als erster Parameter stehenden Menge das als zweiter Parameter stehende Element zu bzw. entfernen es aus ihr. Die anschließende *FOR*-Schleife macht deutlich, daß diese beiden Operatoren nur mit einzelnen Elementen als zweiten Parametern arbeiten, der in Kommentarklammern stehende Versuch, mittels *INCL(Zwiffern, 8..12)* die Schleife "elegant" zu umgehen, würde vom Compiler abgewiesen.

Eine abgekürzte Aktion zum gleichzeitigen Einschluß mehrerer Elemente in eine Menge ist mit Mengenoperatoren möglich. Für Mengenoperationen werden die bekannten Rechenzeichen "+", "-", "*", "/"

verwendet, die aber nicht genau die gleichen Bedeutungen wie die entsprechenden mathematischen Operationen haben. Während die Operatoren "+" und "-" praktisch das Gewohnte tun, nämlich einer Menge eine andere Menge einzufügen (man nennt dies in der Mengenlehre Mengenvereinigung) bzw. von ihr wegzunehmen (Mengendifferenz genannt), erzeugt der Operator "*" die sogenannte Schnittmenge, das ist diejenige Menge, deren Elemente sowohl in der ersten als auch in der zweiten Menge enthalten sind und der Operator "/" erzeugt die "Symmetrische Differenz" zweier Mengen, womit die Menge derjenigen Elemente gemeint ist, die entweder in der einen oder in der anderen, aber nicht in beiden Mengen enthalten sind. In allen Fällen muß berücksichtigt werden, daß jedes Element nur ein einziges Mal in einer Menge enthalten sein kann, wenn also eine Menge 1 und eine Menge 2 jeweils das Element "22" enthalten, dann ist diese "22" in der Vereinigungsmenge trotzdem nicht zweimal, sondern nur einmal enthalten, so wie die Zahl "22" in der Menge der ganzen Zahlen nur ein Mal existiert.

Sie finden die Anwendung der Mengenoperatoren in den Programmzeilen, in denen die Mengen *Ziffern* und *Zwiffern* durch Vereinigungs- und Differenzbildung vergrößert bzw. verkleinert werden, wobei in der ersten Zeile dieses Anweisungsblocks, *Vereinigung := Ziffern + Zwiffern*, die "Addition" mit benannten Variablen durchgeführt ist, während in der zweiten Zeile *Vereinigung := Vereinigung + {13}* eine einelementige Menge der Menge *Vereinigung* hinzugefügt wird. Diese Zeile läßt sich nicht in der Form *Vereinigung := Vereinigung + 13* schreiben, da *Vereinigung* eine Menge(nvariable) ist, der mit dem Mengenoperator "+" nur eine Menge, nicht aber ein Element zugefügt werden kann.

Die anschließende *FOR*-Schleife erzeugt die Menge *GeradeZahlen*, eine Teilmenge der Menge $\{0..31\}$. In der Zeile *UngeradeZahlen := -GeradeZahlen* sehen Sie, daß der Operator "-" ähnlich wie in der Algebra eine zweite Funktion hat; in der Algebra wird -3 die Gegenzahl zu +3 genannt, man kann das Minuszeichen vor der 3 als Umkehroperator bezeichnen. In der Mengenlehre erzeugt dieser Umkehroperator ebenfalls das Gegenteil der Menge, auf die er angewendet wird, diese Menge wird mathematisch Komplementärmenge genannt und enthält alle Elemente, die in der "umgekehrten" Menge nicht enthalten sind. Zur Menge *GeradeZahlen* ist also die Menge *UngeradeZahlen* die Komplementärmenge.

Der Ausgabeteil des Programms stellt die Ergebnisse der vorangegangenen Operationen im Log-Fenster dar. Sie können sich durch Vergleich der in den Boole'schen Bedingungen stehenden Mengen mit den jeweils nachfolgenden Ergebnissen noch einmal die Funktionsweise der verschiedenen Mengenoperatoren klarmachen; dies gilt insbesondere für die symmetrische Differenz, für die Sie in der Boole'schen Bedingung eine äquivalente Darstellung in der Form $Ziffern / Zwiffern = (Ziffern + Zwiffern) - (Ziffern * Zwiffern)$ finden.

In dem Programmteil, der die symmetrische Differenz im Einzelnen ausgibt, macht Sie die Zeile *IF Index IN SymDiff THEN* mit dem beim Programmieren wahrscheinlich am häufigsten gebrauchten Mengenoperator *IN* bekannt. Mit diesem Operator läßt sich ohne Verwendung von Mengenklammern prüfen, ob ein einzelnes Element (nicht also eine Menge) in einer Menge enthalten ist. Andere Mengenvergleichs-

operatoren sind das "=" Zeichen (Prüfung, ob die rechts und links stehenden Mengen übereinstimmen) und das "#" Zeichen (Prüfung, ob die rechts und links stehenden Mengen nicht übereinstimmen), Sie sehen diesen Vergleichsoperator in der Zeile *IF {1, 2, 3} # {1..4} THEN*. Dagegen gibt es in Bezug auf Mengen nicht die von den Zahlen gewohnten Vergleichsoperatoren "<" (kleiner als, aber nicht gleich), "<=" (kleiner als oder gleich), ">" (größer als, aber nicht gleich), ">=" (größer als oder gleich).

Wie am Anfang bereits erwähnt, sind die Elemente einer Menge nicht geordnet, d.h. sie haben keine festgelegte Reihenfolge, Sie finden diese Tatsache bestätigt in der Zeile *IF {3, 1, 2} = {1, 2, 3} THEN*, in der geprüft wird, ob die Menge $\{3, 1, 2\}$ mit der Menge $\{1, 2, 3\}$ übereinstimmt.

6.2. Prozedurtypen

Prozeduren und Funktionen kennen Sie aus den vorangegangenen Kapiteln als Teile innerhalb eines Programms, die durch Aufruf der jeweiligen Namen (sowie eventuelle Übergabe benötigter Parameter) aktivierbar waren und einen festen Inhalt abgearbeitet haben. Darüber hinaus gibt es in Component Pascal die Möglichkeit, Prozedurvariablen zu deklarieren, die Instanzen entsprechender Prozedurtypen sind. In dem Programm *ProcTyp.odc* finden Sie Einzelheiten zum Umgang mit derartigen Prozedurtypen und Prozedurvariablen.

Die Ihnen bisher bekannt gewordenen Datentypen gehören zum Sprachstandard von Component Pascal, sie können deshalb ohne weiteres in jedem Programm benutzt werden. Anders ist es dagegen mit den sogenannten strukturierten Datentypen. Der einzige in der Sprache vordefinierte strukturierte Datentyp ist der Typ *SET*, den Sie im vorigen Abschnitt kennengelernt haben. Andere strukturierte Typen müssen als benutzerdefinierte Datentypen dem Compiler in jedem Einzelfall durch eine Typdeklaration bekannt gemacht werden.

Eine explizite Typdeklaration ist ähnlich aufgebaut wie eine Konstantendeklaration, sie wird eingeleitet von dem Component Pascal Schlüsselwort *TYPE* gefolgt von einem frei wählbaren Typbezeichner und einem Gleichheitszeichen, dem sich die eigentliche Typdeklaration anschließt. Ein Beispiel finden Sie in der Typdeklaration *TYPE Funktion = PROCEDURE (x: REAL): REAL*. Diese zeigt Ihnen, daß ein Prozedur- bzw. Funktionstyp sich zusammensetzt aus einem vollständigen Funktions- oder Prozedurkopf inklusive formale Parameter, deren Typen und gegebenenfalls einem Rückgabotyp, jedoch ohne Prozedurname.

Interessant sind Prozedurtypen nicht zuletzt deshalb, weil sie wie alle anderen Typen ihrerseits als Typen formaler Prozedurparameter verwendet werden können, wie Sie es in der Deklaration *Ffunktion = PROCEDURE (f: Funktion; x: REAL): REAL* sehen, deren formaler Parameter *f* selbst einen Funktionstyp hat. Schließlich zeigt Ihnen die Deklaration *Textschreiben = PROCEDURE*, daß nicht nur Funktionen,

sondern auch Prozeduren als Typen fungieren können, wobei nicht nur wie im vorliegenden Fall parameterlose Prozedurtypen, sondern auch solche mit Parametern möglich sind.

Im Modul *TutProcTyp* finden Sie als global deklarierte Variablen *Zahl*: *REAL* und *Func*: *Funktion*, wobei mit dieser zweiten Deklaration eine Variable des zuvor deklarierten Prozedurtyps *Funktion* deklariert wird. Außerdem finden Sie die beiden Funktionsprozeduren *Wurzel* und *Quadrat*. Die Funktion *Wurzel* ist einfach gebaut, sie hat eine *REAL*-Variable als Parameter und gibt die Quadratwurzel des Parameterwertes als *REAL*-Wert zurück. Interessanter ist die Funktion *Quadrat*, sie ist wie eine mathematische Funktion universell einsetzbar, da sie für jede beliebige Funktion des Typs *Funktion* das Quadrat einer ebenfalls beliebigen reellen Zahl zurückgibt.

Eine Anwendung sehen Sie im Initialisierungsteil des Moduls, der globalen Variablen *Func* wird die Funktion *Math.Sin* zugewiesen, die von der Parameterliste und dem Rückgabetyt her mit dem Typ *Funktion* kompatibel ist, wie Sie der Schnittstelle des Moduls *Math* entnehmen können. Anschließend können der Funktion *Quadrat* die Funktion *Func* sowie als zweiter Parameter die Zahl $\frac{\pi}{6}$ übergeben und das Ergebnis mit *Out.Real* im Log-Fenster angezeigt werden.

Eine weitere Anwendung von Prozedurvariablen sehen Sie in der Prozedur *Text1*, in der als erstes die Variable *Ffunc* vom Typ *Ffunktion* deklariert wird; diese erhält im Anweisungsteil die Funktion *Quadrat* zugewiesen und der Wert der globalen Variablen *Func*, die bisher die Prozedur *Math.Sin* enthält, wird zu *Wurzel*. Mit diesen Werten wird anschließend eine banale Rechnung durchgeführt, wobei Sie beachten sollten, daß die Funktionen *Wurzel* und *Quadrat* nicht nur den Prozedurvariablen zugewiesen, sondern wie üblich auch direkt verwendet werden können.

In der Kommandoprozedur *Ausgeben* erhält die lokale Variable *Texten*: *Textschreiben* in Abhängigkeit vom Wert der Variablen *Zahl* eine der beiden Prozeduren *Text1* oder *Text2* zugewiesen, die danach abgearbeitet wird. Dies hätte natürlich durch direkten Aufruf der jeweiligen Prozeduren in den Zweigen der IF-Abfrage geschehen können; hier soll der Gebrauch des Prozedurtyps *Textschreiben* demonstriert werden.

In der Kommandoprozedur *Einlesen*, die Ihnen in ihrer Funktionsweise bekannt sein dürfte, lernen Sie in der Zeile *ASSERT(Zahl >= 0.0, 20)* mit der Standardprozedur⁸ *ASSERT* (engl. to assert: zusichern) eine nützliche Möglichkeit kennen, in Component Pascal die Einhaltung bestimmter Bedingungen zu garantieren. *ASSERT* hat zwei Parameter, der erste ist ein Boole'scher Ausdruck und stellt die zu prüfende Bedingung dar, der zweite Parameter ist fakultativ, d. h. er muß nicht angegeben werden - wenn er angegeben wird, muß er vom Typ *INTEGER* sein. *ASSERT* prüft zur Laufzeit des Programms die Bedingung des ersten Parameters und setzt bei deren Zutreffen (*TRUE*) die Programmausführung fort, im anderen Fall

⁸ Eine Zusammenstellung aller vordefinierten Bezeichner und Standard-Prozeduren finden Sie im Anhang C und in der BlackBox Hilfe im Dokument *Language Report*.

wird sie abgebrochen und der zweite Parameterwert wird, sofern er vorhanden ist, an die ausführende Systemumgebung zur Auswertung zurückgegeben.

In der BlackBox Systemumgebung werden diese Parameterwerte nach bestimmten Kategorien eingestuft, die im begleitenden Dokumentationsmaterial (*Programming Conventions*) des Systems zu finden sind. Die beiden wichtigsten dieser Kategorien betreffen die Verletzung von Prä- und Postbedingungen, die Sie im Zusammenhang mit dem Programm *Konvert1.odc* kennengelernt haben. Parameterwerte von 20 bis 59 signalisieren die Verletzung von Vorbedingungen (preconditions), d. h. eine Bedingung wird nicht eingehalten, die zur Ausführung der nachstehenden Anweisung(en) erfüllt sein muß. In analoger Weise signalisieren Parameterwerte von 60 bis 99 die Prüfung von Nachbedingungen (postconditions), d. h. eine Bedingung wird getestet, die nach Ausführung der vorstehenden Anweisung(en) erfüllt sein muß.

Entfernen Sie die Kommentarklammern um die Zusicherung und lassen Sie das Modul neu kompilieren, werden Sie beim Einlesen des Wertes "-8.4", nicht die Fehlermeldung der Prozedur *Text2*, sondern ein Fenster erhalten, dessen erste Zeilen folgendermaßen aussehen (s. Abb. 6)

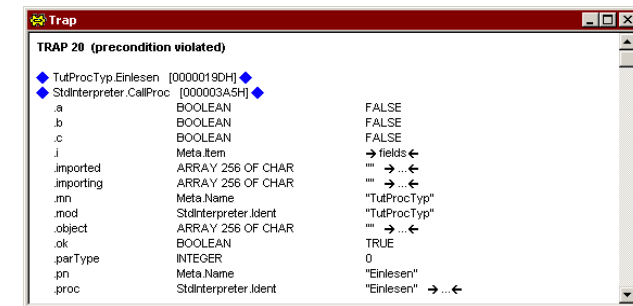


Abbildung 6
Von der *ASSERT*-Prozedur in *TutProcTyp.Einlesen* bei einem Laufzeitfehler erzeugtes Trap-Fenster

Die Überschrift *TRAP 20 (precondition violated)* weist daraufhin, daß eine Vorbedingung mit der Fehlernummer "20" nicht eingehalten wurde (hier war es die Bedingung *Zahl >= 0.0*). Die einzige Zeile, die Sie außer der Überschrift noch beachten müssen, ist die erste, von zwei Rauten begrenzte Zeile *TutProcTyp.Einlesen [0000019DHI]*, die Sie auf die Ursache des Fehlers, die Prozedur *Einlesen* des Moduls *TutProcTyp* aufmerksam macht. Klicken Sie auf die linke der beiden Rauten, öffnet sich das folgende Fenster (s. Abb. 7)

Variable	Typ	Wert
TutProcTyp		Uebste
Func	PROCEDURE	Math.Sin
Zahl	REAL	-8.399999618530273

Abbildung 7
Aktuelle Werte der globalen Variablen des Moduls
zum Zeitpunkt des Laufzeitfehlers

In der linken oberen Ecke steht der Name des verursachenden Moduls. Darunter sind die globalen Variablen des Moduls mit ihren Typen und aktuellen Werten aufgelistet. Die Namen werden von einem Punkt eingeleitet, damit deutlich wird, daß es sich bei `.Zahl` eigentlich um den Bezeichner `TutProcTyp.Zahl` handelt. Da der angegebene Wert von `Zahl` negativ ist, ist Ihre Fehlersuche hier möglicherweise beendet, falls Sie sich erinnern, daß die Zusicherung negative Werte der Variablen `Zahl` ausschließen sollte.

Sollten Sie jedoch an dieser Stelle noch unsicher sein, können Sie in dem Trap-Fenster auf die rechts neben der Zeile `TutProcTyp.Einlesen [0000019DH]` stehende Raute klicken. Sofern sich der Quelltext des Moduls im Verzeichnis `MOD` des entsprechenden Subsystems (`Tut`) befindet, wird er angezeigt und die fehlererzeugende Zeile des Textes ist markiert (s. Abb. 8), anderenfalls fordert BlackBox Sie auf, das Modul selbst zu suchen und zu öffnen.

```

(* ----- Einlesen *)
PROCEDURE Einlesen*;
BEGIN
  Zahl := 0.0;
  In.Open;
  In.Real(Zahl);
  ASSERT(Zahl >= 0.0, 20);
  Out.Ln;
  Out.Ln;
END Einlesen;

```

Abbildung 8
Die den Laufzeitfehler erzeugende Anweisung im Quellcode

Das Zusammenwirken der Informationen in den drei Fenstern der Abbildungen 6 bis 8 sollte in fast allen Fällen ausreichen, dem Fehler auf die Spur zu kommen. BlackBox bietet, wie Sie sehen, nicht nur einen besonders benutzerfreundlichen Compiler, mit den hier gezeigten Möglichkeiten existiert ein in anderen Programmierumgebungen selten zu findendes Informationssystem, um den gegenüber Kompilationsfehlern viel tückischeren Laufzeitfehlern auf die Spur zu kommen. (Weitere Einzelheiten zu den Informationen eines Trap-Fensters finden Sie in der Dokumentation *Dev Subsystem User Manual* des BlackBox Systems.)

Unabhängig von diesen Möglichkeiten sollten Sie bei Ihrer Programmierarbeit die folgenden Regeln beachten

1. Am einfachsten lassen sich Fehler behandeln, die nicht gemacht wurden.
2. Relativ einfach lassen sich Fehler behandeln, die vom Compiler entdeckt werden können.
3. Sehr viel unangenehmer sind Fehler, die zur Laufzeit eines Programms auftreten. Einige können sich monatelang verstecken und treten, wie alle Fehler, immer im unpassendsten Moment auf.
Machen Sie daher oft Gebrauch von Zusicherungen, die Ihnen das Austesten Ihrer Programme erleichtern.
4. Am unerfreulichsten sind Fehler, die der Programmierer oder Verkäufer eines Programms als besondere Funktionalität verkauft (is it a bug or is it a feature?). Behaupten Sie nie, Sie seien ein perfekter Programmierer – es gibt keine. Es gibt schließlich auch keine Wunder!