

## KAPITEL 5

### PROZEDUREN

In den vorangehenden Kapiteln haben Sie die grundlegenden Kontrollstrukturen der Programmiersprache Component Pascal kennengelernt. Ein zentraler Aspekt - die Prozeduren - ist dabei allerdings nur am Rande erwähnt worden. Component Pascal stützt sich, wie viele moderne Programmiersprachen, wesentlich auf das Prozedurkonzept, sämtliche Aktionen werden von Prozeduren ausgeführt, das in älteren Programmiersprachen übliche Hauptprogramm, mit dem der Anweisungsablauf kontrolliert wurde, gibt es nicht. Dies entspricht der Tatsache, daß moderne Programme nicht mehr wie früher ablaufgesteuert, sondern weitgehend interaktiv sind, nicht der Benutzer soll auf das Programm reagieren müssen, sondern das Programm auf den Benutzer. Sie haben die Konsequenzen dieses Konzepts bereits an zwei Stellen kennengelernt. Zum einen haben Sie alle Aktionen in Component Pascal durch Aktivierung von Kommando-prozeduren ausgelöst, zum anderen haben Sie am Modul *TutKonvert1* gesehen, daß der Anweisungsteil eines Moduls nicht, wie in älteren Programmiersprachen üblich, zur Programmsteuerung, sondern lediglich zur Initialisierung von Daten dient, der Programmfluß selbst wird von Prozeduren kontrolliert. Deren Aufbau und Wirkungsweise lernen Sie in diesem Kapitel genauer kennen.

#### 5.1. Parameterlose Prozeduren

Sehen Sie sich das Programm *Prozed1.odc* an. Es enthält einmal die vertraute Kommandoprozedur *Start*, außerdem drei weitere, identisch strukturierte Prozeduren, *Vorspann*, *Botschaft* und *Nachklapp*. Diese werden jedoch nicht exportiert, sind also keine außerhalb des Moduls aktivierbaren Kommandos. Betrachten Sie *Start* im Detail, sehen Sie, daß diese Prozedur keine Anweisungen im engeren Sinn enthält, sondern lediglich die Namen der drei eben erwähnten Prozeduren, getrennt von einigen *Out.Ln*-Anweisungen.

Wenn Sie das Kommando *Prozed1.Start* ausführen lassen werden Sie feststellen, daß die Bildschirmausgabe des Programms aussieht, als ob die Anweisungsteile der jeweils in *Start* aufgerufenen Prozeduren an Stelle der Namen gestanden hätten. So wie für Sie die Prozedur *Start* ein extern aktivierbares Kommando ist, sind für *Start* die übrigen Prozeduren interne Kommandos des Moduls und können deshalb von *Start* über die Prozedurnamen aktiviert werden. Da es sich um modulinterne Prozeduren handelt, müssen diese jedoch nicht mit ihrem vollständigen, qualifizierten Namen aktiviert werden, der einfache Prozedurname genügt in diesem Fall, so wie Sie in vertrauter Umgebung mit den Vornamen Ihrer Mitmenschen auskommen.

Ein Grund für die Verwendung von Prozeduren liegt darin, daß sie wesentlich zur Verständlichkeit eines Programms beitragen (können). Wenn es dem programm-schreibenden Menschen gelingt, für seine Prozeduren "sprechende" Bezeichner zu finden, die die jeweiligen Zwecke der Prozeduren ausdrücken, so kann einem Leser des Programms bereits durch die Namen ein schnelles, überblickartiges Verständnis des gesamten Programminhalts vermittelt werden, ohne ihm zuzumuten, sich durch eine Unmenge von Einzelheiten zu kämpfen, die beim ersten Kennenlernen eines Programms eher hinderlich sind. Eine solcherart strukturierte Programmierung kann gleichzeitig das Verständnis der Einzelheiten in den Prozeduren erheblich erleichtern, da diese in sich abgeschlossen und (hoffentlich) übersichtlich sind. Ein weiterer Grund besteht darin, daß man in Prozeduren Anweisungsblöcke zusammenfassen kann, die innerhalb eines Programms mehrmals gebraucht werden; statt solche Anweisungsblöcke jedesmal erneut vollständig aufzuschreiben, genügt es, sie durch den Prozedurbezeichner an der gewünschten Stelle aufzurufen, den Rest erledigt der Computer. Schließlich liegt ein dritter nicht unwesentlicher Grund für die Verwendung von Prozeduren darin, daß diese nur für die Dauer ihres Ablaufs Speicherplatz belegen.

Prozeduren der jetzt vorgestellten Art heißen parameterlose Prozeduren, es sind Prozeduren, die nur mit ihrem Namen aufgerufen werden. Eine zweite, sehr viel häufiger gebrauchte Art sind Prozeduren mit Parametern, die Sie in den folgenden Abschnitten kennenlernen werden.

## 5.2. Prozeduren mit einem Parameter

### - Formale und aktuelle Parameter -

Wie Sie in früheren Abschnitten bereits erfahren haben, werden Parameter die Ausdrücke genannt, die einer Anweisung in Klammern beigelegt oder, wie man auch sagt, übergeben werden. Sie wissen ebenfalls, daß solche Ausdrücke vor Ausführung der Anweisung stets soweit entwickelt werden, daß sie einen Wert liefern, der einem bekannten Typ zuordenbar ist. Da an einer Programmiersprache nichts Geheimnisvolles ist, können Sie sich inzwischen wahrscheinlich denken, daß es sich generell bei den in Anweisungen verwendeten Operatoren um vorgefertigte Prozeduren handelt, die mit den eben erwähnten Werten als Parametern arbeiten.

Die erste nicht systeminterne Prozedur, welche ebenfalls mit Parametern arbeitet, lernen Sie in dem Programm Prozed2.odc kennen. Am besten übergehen Sie zum Kennenlernen des Programms die beiden ersten Prozeduren und wenden sich der Kommandoprozedur *Start* zu. Sie finden dort eine einfache *FOR*-Schleife mit der Kontrollvariablen *Temp*. Sie erinnern sich, daß die Kontrollvariablen von *FOR*-Schleifen in keinem Fall innerhalb der Schleife verändert werden sollten. Da aber der jeweils aktuelle Wert von *Temp* sich innerhalb der Schleife ändern soll, wird er im nächsten Schritt in die Variable *Index* kopiert.

Dann wird die Prozedur *Lassen* aufgerufen und ihr als Parameter *Index* übergeben. Der Prozedurkopf von *Lassen* enthält in Klammern als Parameter die Variable *Hund*, wobei ebenso wie bei Variablendeklarationen nach einem Doppelpunkt ein Typ, *INTEGER*, angegeben ist. Die Ganzzahlvariable *Hund* ist auf diese Weise für die Laufzeit der Prozedur *Lassen* deklariert und sie ist nur innerhalb der Prozedur bekannt. Nach dem Beenden der Prozedur kann auf diese Variable nicht mehr zugegriffen werden.

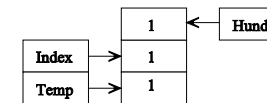


Abbildung 5 a  
Variablen zu Beginn der Prozedur  
*TutProzed2.Lassen*

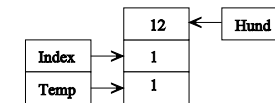


Abbildung 5 b  
Variablen am Ende der Prozedur  
*TutProzed2.Lassen*

Die Einführung von *Hund* als Parameter im Prozedurkopf hat nun neben der dadurch erfolgten Deklaration als Variable die zusätzliche Wirkung, daß beim Aufruf der Prozedur *Lassen* der Wert des übergebenen Parameters *Index* in *Hund* kopiert wird. Deshalb ist es möglich, ohne eine gesonderte Wertzuweisung in der Prozedur als erstes den Wert von *Hund* auszugeben. Anschließend wird *Hund* ein neuer Wert zugewiesen und dieser dann ebenfalls ausgegeben, womit die Prozedur *Lassen* beendet ist. Zurück in *Start* wird der Wert von *Index* ausgegeben und Sie werden, wenn Sie das Programm ausführen lassen, sehen, daß *Index* unabhängig von den Veränderungen des Wertes von *Hund* weiterhin den Wert "1" enthält, mit dem die Prozedur *Lassen* aufgerufen wurde (s. Abb. 5 a und 5 b).

Im zweiten Schritt wird die Prozedur *Ändern* aufgerufen, ebenfalls mit *Index* als Parameter. Wenn Sie die beiden Prozeduren *Lassen* und *Ändern* vergleichen, werden Sie vielleicht beim ersten Hinsehen der Meinung sein, es gebe keinen Unterschied; es gibt jedoch einen, der aber tatsächlich so unscheinbar ist, daß man ihn leicht übersehen kann. Betrachten Sie die Parameterliste der Prozedur *Ändern* genauer, werden Sie bemerken, daß außer der Änderung des Variablennamens von *Hund* in *Katz* das Schlüsselwort *VAR* vor *Katz* steht. Der daraus folgende Unterschied ist allerdings bemerkenswert. Diese Form des Parameters sorgt nämlich dafür, daß *Katz* nicht mehr mit einer Kopie des Wertes von *Index* arbeitet, sondern mit dem Original. Der Variablenname *Katz* existiert zwar während der Laufzeit der Prozedur zusätzlich zu *Index* und er existiert nach dem Prozedurende ebensowenig weiter, wie es *Hund* bei der Prozedur *Lassen* tat, aber alle Veränderungen des Wertes von *Katz* während des Ablaufs der Prozedur *Ändern* werden in *Index* gespeichert, der letzte Wert von *Katz* ist deshalb nach dem Ende der Prozedur in *Index* verfügbar. Sie sehen dies bei der Programmausführung daran, daß der prozedurintern der Variablen *Katz* zugewiesene Wert "-1" nach dem Prozedurende auch als Wert von *Index* ausgegeben wird (s. Abb. 5 c und 5 d).

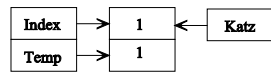


Abbildung 5 c  
Variablen zu Beginn der Prozedur  
*TutProzed2.Ändern*

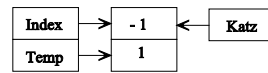


Abbildung 5 d  
Variablen am Ende der Prozedur  
*TutProzed2.Ändern*

Zur Veranschaulichung der (Nicht-)Änderungen der Parameterwerte wird zum Schluß *Lassen* mit dem aktuellen Wert von *Index* erneut aufgerufen; Sie können sich an den Ergebnissen der Aufrufe noch einmal die Unterschiede zwischen den beiden Parameterarten klarmachen. Parameter, die unter Verwendung des Schlüsselwortes *VAR* als Adressen der Originalvariablen übergeben werden, bezeichnet man als *VAR*- oder Referenz-Parameter, während Variablen, deren Wert (engl. value) in Kopie übergeben wird, *VAL*- oder Wert-Parameter genannt werden.

Die Tatsache, daß für eine bereits definierte Variable im Kopf der Prozedur ein neuer Name erscheint, ist für den Anfänger oft verwirrend. Bedenken Sie aber bitte Folgendes. Zum einen soll es möglich sein, eine Prozedur mehrmals innerhalb eines Programms zu verwenden. Ebenso soll es möglich sein, Prozeduren "auf Vorrat", als fertige Werkzeuge zu schreiben. Sie kennen solche vorgefertigten Werkzeuge bereits, z.B. die Prozedur *Out.Int*. Könnte man Prozeduren nicht ohne interne, von der Umgebung unabhängige Variablenamen im Parameter Teil der Prozedurdeklaration schreiben, wäre die Verwendung einer fertigen Prozedur durch einen Benutzer unmöglich.

Der Kopf einer Prozedur wird auch ihre Signatur genannt, die in ihm stehenden Parameter heißen "Formale Parameter". Dagegen heißen die bei einem Prozeduraufruf übergebenen Werte (dies können Werte von Variablen oder Konstanten sein) "Aktuelle Parameter". Verdeutlichen Sie sich bitte, daß es wegen der Namensunabhängigkeit der formalen und aktuellen Parameter einer Prozedur ausschließlich auf die Typkompatibilität der Parameter ankommen kann, daher die (unbedingt notwendige) Typangabe bei den formalen Parametern im Prozedurkopf.

### 5.3. Prozeduren mit mehreren Parametern

#### - Die Reihenfolge der Parameter -

Einer Prozedur kann man nicht nur einen einzelnen, sondern auch mehrere Parameterwerte übergeben. Was dabei zu beachten ist und wie dies genau geschehen muß, sehen Sie an dem Programm *Prozed3.odc*. In der Prozedur *Start* werden vier Variablen deklariert, drei von ihnen wird zu Beginn des Anweisungsteils ein Wert zugewiesen, die vierte, *Obst*, bleibt unbelegt. Danach wird die Prozedur *ObstAddieren* mit allen vier Variablen als Parametern aufgerufen. Die Prozedur selbst ist sehr einfach, sie besteht lediglich aus

### 5.3. Prozeduren mit mehreren Parametern

einer einzigen Anweisung:  $Variable := Wert1 + Wert2 + Wert3$ . Selbstverständlich hätte dies mit weniger Aufwand in der Prozedur *Start* untergebracht werden können, hier sollen jedoch mit dem Prozeduraufruf die folgenden wichtigen Tatsachen erläutert werden.

Im Prozedurkopf von *ObstAddieren* ist *Variable* als *VAR*-Parameter deklariert; der in *Variable* gespeicherte Wert steht demnach in der aufrufenden Umgebung zur Verfügung. Wie Sie aus der Reihenfolge der Parameter in der Deklaration ersehen, steht *VAR Variable* an dritter Stelle. Dies ist entscheidend dafür, welche der Variablen des Prozeduraufrufs einen Wert an *Variable* übergeben und den Wert, den die Variable *Variable* in der Prozedur *ObstAddieren* erhält, nach dem Ende von *ObstAddieren* weiterverwenden kann, es ist die im Aufruf ebenfalls an dritter Stelle stehende Variable *Obst*. Da an die formalen Prozedurparameter nicht die Namen, sondern nur die in den Variablen gespeicherten Werte als Kopie bzw. die Speicheradressen (im Fall von *VAR*-Parametern) übergeben werden, spielt für die Zuordnung von aktuellen und formalen Parametern ausschließlich die Reihenfolge der Parameter im Aufruf eine Rolle. Somit werden in diesem Fall *Apfel* in *Wert1*, *Pflaume* in *Wert2* kopiert, *Variable* erhält die Adresse von *Obst* und *Birne* wird in *Wert3* kopiert. Nach Beendigung der Prozedur *ObstAddieren* ist demnach *Obst* mit dem *Variable* zugewiesenen Wert belegt und kann ohne Gefahr eines unkontrollierbaren Ergebnisses in *Start* von der *Out.Int*-Anweisung ausgegeben werden.

In einem "richtigen" Programm würde man selbstverständlich die Anordnung der formalen Parameter im Prozedurkopf in einer intuitiv leichter erfaßbaren Weise vornehmen, etwa als *PROCEDURE ObstAddieren (VAR Variable: INTEGER; Wert1, Wert2, Wert3: INTEGER)* und der Prozeduraufruf würde entsprechend als *ObstAddieren (Obst, Apfel, Pflaume, Birne)* erfolgen. Es soll hier gezeigt werden, daß die Reihenfolge bei der Deklaration beliebig ist, aber beim Aufruf der Prozedur genau beachtet werden muß, da nur diese Reihenfolge für die Zuordnung der aktuellen zu den formalen Parametern eine Bedeutung hat.

### 5.4. Sichtbarkeitsbereiche von Variablen

Im Programm *Prozed3.odc* wurde keiner der Variablenamen aus der Prozedur *Start* innerhalb der Prozedur *ObstAddieren* verwendet, dennoch war es dem mit *VAR* deklarierten formalen Parameter *Variable* möglich, die aus dem *VAR* folgende Veränderung des aktuellen Parameters, der Variablen *Obst*, vorzunehmen. Im Programm *Prozed4.odc* erfahren Sie genauer, welche Variablen unter welchen Umständen in welchen Teilen eines Programms sichtbar, also zu verändern und zu benutzen sind. Mit diesem Programm sollen die im Kapitel 4.5 erwähnten Begriffe "globale" und "lokale" Bezeichner genauer erklärt werden, außerdem die Bedingungen, unter denen Bezeichner in den verschiedenen Teilen eines Programms "sichtbar" sind.

Zu diesem Zweck finden Sie einige Variablen direkt im Rumpf des Moduls *TutProzed4* deklariert, andere innerhalb der Prozedur *Prozedur*. Wenden Sie sich dem Anweisungsteil der Prozedur *Start* zu. Dort werden drei der vier Variablen, die im Deklarationsteil des Moduls deklariert worden sind, mit Werten versehen, die vierte, *Index*, ist als Kontrollvariable der *FOR*-Schleife bereits initialisiert. Anschließend wird die Prozedur *Schreib* aufgerufen, die nichts anderes tut, als einen Text auszugeben, dessen Inhalt von einer Bedingung abhängig ist. Die Verwendung der Boole'schen Variablen *Ident* demonstriert die Möglichkeit, eine Prozedur mit wenig Aufwand für mehrere Einsatzzwecke zu schreiben, derartige Prozeduren sind oftmals zeit- und arbeitssparend.

In der Prozedur *Schreib* wird als erstes ausgegeben, daß die nachfolgenden Werte nicht aus der Prozedur *Prozedur* stammen (*Ident* wurde auf *FALSE* gesetzt). Anschließend werden der Reihe nach die Werte ausgegeben, die den im Modulrumpf deklarierten Ganzzahlvariablen in *Start* zugewiesen wurden. Diese Variablen sind also in den beiden Prozeduren *Schreib* und *Start* sichtbar, sie können geschrieben und gelesen (und sie könnten verändert) werden. Der Text "Zähler global", der ebenfalls ausgegeben wird, besagt, daß es sich bei *Zähler* wie den anderen Variablen, die im Modulrumpf deklariert worden sind, um globale, in allen Prozeduren (wie auch in einem potentiellen Anweisungsteil des Moduls selbst) sichtbare Variablen handelt.

Die Sichtbarkeit gilt allerdings nur solange, wie keine besonderen Umstände eintreten. Einige dieser Umstände erklärt Ihnen die Prozedur *Prozedur*, weitere werden Sie in den folgenden Abschnitten dieses Kapitels kennenlernen. In der Prozedur *Prozedur* werden zwei Variablen deklariert, *Laus* und *Zähler*, die somit lokal zu dieser Prozedur sind. Anschließend werden diesen beiden sowie den globalen Variablen *Ident* und *Andere* Werte zugewiesen und die Prozedur *Schreib* wird aufgerufen. Da die Variable *Ident* in der Prozedur *Prozedur* auf *TRUE* gesetzt worden ist, werden der *THEN*-Teil der Alternative und wiederum die Werte von *Index*, *Andere* und *Zähler* ausgegeben. Beachten Sie, daß auch jetzt nicht nur der Text "Zähler global" geschrieben wird, sondern tatsächlich auch der entsprechende Wert. Grund dafür ist, daß der Prozedur *Schreib* die in der Prozedur *Prozedur* deklarierten Variablen nicht bekannt sind, denn diese sind zu *Prozedur* lokal, außerhalb davon also unsichtbar. Andererseits ist *Andere* (wie *Ident*, *Index*, *Zähler*) eine globale Variable, denn sie wurde im Modulrumpf deklariert, ihr Wert ist beiden Prozeduren bekannt, er kann gelesen und geschrieben werden, wie es in der Prozedur *Prozedur* geschehen ist.

Jetzt werden Sie sagen, auch *Zähler* sei in *Prozedur* verändert worden und das stimmt. Hier kommt jedoch eine merkwürdige und nützliche Eigenschaft von Component Pascal zum Tragen, die es gestattet, global definierte Variablen vor der Veränderung durch Prozeduren zu schützen, sie zu verstecken. Da in der Prozedur *Prozedur* die Variable *Zähler* als lokale Variable definiert wurde, ist die globale Variable gleichen Namens in dieser Prozedur nicht existent, es kann nicht auf sie zugegriffen werden, die Prozedur kennt nur die lokale Variable *Zähler*. Der Wert der globalen Variablen *Zähler* kann deshalb innerhalb der

Prozedur *Prozedur* nicht verändert werden und die Prozedur *Schreib* gibt für *Zähler* den ihr bekannten, eingangs in *Start* zugewiesenen Wert "1" der globalen Variablen aus.

Nach der Rückkehr aus *Schreib* in die Prozedur *Prozedur* werden von dieser die Werte der lokalen Variablen *Zähler* und *Laus* ausgegeben. Machen Sie sich bitte klar, daß beide Werte nur hier geschrieben werden können, da sie als Werte lokaler Variablen außerhalb der Prozedur nicht bekannt sind. Außerdem steht die Prozedur *Prozedur* hinter *Schreib*, womit sie als Ganzes dieser *Prozedur* unbekannt ist, denn auch für Prozeduren gilt die Regel, daß nur verwendet werden kann, was bekannt ist, also vor der verwendenden Stelle deklariert wurde.

### 5.5. Stil ist eine Geschmacksfrage

Ein gutes Beispiel für schlechten Programmierstil finden Sie im Programm *Prozed5.odc*. Sollte es Ihnen bekannt vorkommen, so irren Sie sich nicht, es entspricht weitgehend dem Programm *Prozed3.odc*, ja, es erzeugt genau die gleiche Bildschirmausgabe. Der Anweisungsteil der Prozedur *Start* ist, bis auf einige sich selbst erklärende Zeilen in Kommentarklammern, mit dem von *Start* aus *Prozed3.odc* identisch. Einen Unterschied gibt es bei den Variablendeklarationen. Sie sehen, daß die Variablen *Apfel*, *Birne*, *Obst*, *Pflaume* im Modulrumpf deklariert werden, sie sind also für das gesamte Modul mit allen Prozeduren global (dies ist kein guter Programmierstil, da Variablen soweit möglich lokal gehalten werden sollten, aber dies Programm trägt seinen Titel schließlich absichtsvoll).

Unterschiedlich zu *Prozed3.odc* ist auch die Deklaration der Prozedur *ObstAddieren*. Dort finden Sie als formale Parameter zwei der Variablenamen aus dem Modul wieder, *Apfel* und *Pflaume*. Denken Sie bitte daran, daß die Namensgleichheiten keine Rolle spielen, da alle im Prozedurkopf von *ObstAddieren* deklarierten Variablen ebenso zur Prozedur lokal sind wie die Variable *Boskop* und als formale Parameter je nach Deklaration als *VAL*-Parameter die Kopien der Werte oder als *VAR*-Parameter die Adressen der aktuellen Parameter des Prozeduraufrufs übergeben bekommen.

Für den Leser des Programms ist es möglicherweise verwirrend, daß diese Namen erneut in *ObstAddieren* auftauchen, wobei die Doppelung von *Apfel* und *Apfel* dem menschlichen Verstand noch akzeptabler erscheinen mag als die Tatsache, daß nach dem Aufruf von *ObstAddieren* die lokale Variable *Pflaume* den Wert der global deklarierten Variablen *Birne* enthält, während der Wert der globalen Variablen *Pflaume* in der lokalen Variablen *Wert2* wiederzufinden ist. Dies alles ist aber nur für den menschlichen Leser des Programms verwirrend, für den Compiler bedeutet es kein Problem, da für ihn nur die im Prozeduraufruf festgelegte Reihenfolge der Parameterzuordnungen bedeutsam ist.

Die Folge der teilweisen Namensgleichheiten ist, daß *ObstAddieren* zwei der vier globalen Variablen, *Apfel* und *Pflaume*, nicht kennt, diese sind verborgen; nur auf die Variablen *Birne* und *Obst* kann innerhalb

der Prozedur *ObstAddieren* zugegriffen werden (die Tatsache, daß die globale Variable *Birne* weiterhin sichtbar ist, findet sich wieder in der Zuweisung des Wertes von *Birne* an die lokale Variable *Pflaume*).

Bei der Variablen *Obst* ist die Zugriffsmöglichkeit unbedenklich, da diese Variable als VAR-Parameter übergeben wurde. Dagegen ist es im Fall der Variablen *Birne* Sache des Programmierenden, ob und wie weit die globale Variable verborgen werden soll. Die Entscheidung hängt von der Art des Programms ab, auf alle Fälle aber sollte man zu Mißverständnissen führende Konstruktionen vermeiden, was in diesem Programm mit Sicherheit nicht der Fall ist.

### 5.6. Sichtbarkeitsbereiche von Prozeduren

Die beiden letzten Programme haben Ihnen gezeigt, unter welchen Umständen Variablennamen von verschiedenen Programmteilen aus sichtbar sind und wie man die Sichtbarkeiten verändern kann. Das Programm *Prozed6.odc* zeigt Ihnen, daß die Sichtbarkeitsregeln sich nicht auf Variablen beschränken, sondern generell für alle deklarierten Bezeichner gelten, also auch für Prozedurnamen. Außerdem werden Sie erfahren, welchen Einschränkungen und Bedingungen Sichtbarkeiten unterliegen.

Sie finden innerhalb des Programms *Prozed6.odc* außer der Kommandoprozedur *Start* insgesamt sechs Prozeduren, die teilweise ineinander geschachtelt sind. Die Prozedur *Start* selbst ruft in ihrem Anweisungsteil die Prozeduren *Proc1* und *Proc3* auf, die ebenso wie *Start* direkt im Deklarationsteil des Moduls stehen. Darüber hinaus finden Sie in *Start* den in Kommentarklammern stehenden Aufruf der Prozedur *Proc5* mit dem Hinweis auf die Fehlermeldung *undeclared identifier*. Diese Meldung würde bei fehlenden Kommentarklammern ausgegeben, weil *Proc5* nicht im Deklarationsteil des Moduls steht und auf der Ebene des Moduls für den Compiler nicht feststellbar ist, was sich innerhalb dieser beiden Prozeduren befindet, also auch nicht die Existenz der Prozedur *Proc5*.

Sie können sich die Sichtbarkeiten der verschiedenen Prozeduren vielleicht am besten als eine Serie von Schachteln vorstellen, die in einander stecken. Die äußerste Schachtel ist das Modul selbst, sozusagen die nullte SichtbarkeitsEbene. Machen Sie diese auf, gehen also in das Modul hinein, haben Sie zum einen die Kommandoprozedur *Start* vor sich, außerdem die Prozeduren *Proc1* und *Proc3*, die alle zum Modul lokal sind, wie Schachteln, die nebeneinander innerhalb der Aussenschachtel (dem Modul) stecken. Nur diese Prozeduren können auf dieser Ebene benutzt werden, da nur sie hier bekannt - sichtbar - sind.

Machen Sie eine der Schachteln auf - z.B. *Proc1* - begeben Sie sich auf die nächsttieferen, erste Ebene. Dort finden Sie eine weitere Schachtel, die Prozedur *Proc2*, die ihrerseits zu *Proc1* lokal ist, weshalb sie für alle auf der höheren, nullten Ebene des Moduls existierenden Prozeduren unsichtbar bleibt, denn diese können nicht in *Proc1* hineinsehen. *Proc2* ist die letzte Schachtel dieses Pakets, aber anders sieht es aus, wenn Sie statt *Proc1* die dazu parallele Prozedur *Proc3* öffnen. In dieser stecken zwei weitere Prozeduren,

*Proc4* und *Proc5*, beide lokal zu *Proc3* und daher von der darüber liegenden Ebene nicht sichtbar. Analog gilt für die in *Proc5* steckende Prozedur *Proc6*, die sich damit vom Modul aus auf der zweiten absteigenden Ebene befindet, daß sie lokal zu *Proc5* ist, also nur von dieser Prozedur aufgerufen werden kann.

Wichtig ist allerdings, daß die Beschränkung der Sichtbarkeit nur absteigend gilt. Auf jeder Ebene sind sämtliche Bezeichner aller in direkter Linie darüber liegenden Ebenen benutzbar, sie sind für die jeweilige Ebene ihrer Deklaration lokal, für alle tiefer liegenden global.

Sichtbarkeiten existieren in absteigender Richtung also nur auf der selben Ebene. Dagegen sind in aufsteigender Richtung alle Bezeichner der oberhalb liegenden Ebenen sichtbar, allerdings nur in direkter Linie. So liegt bei einem (potentiellen) Aufruf von *Proc2* innerhalb von *Proc6* zwar die SichtbarkeitsEbene von *Proc2* von der Zahl her höher, aber es handelt sich nicht um Ebenen derselben Hierarchie, der Compiler reagiert (potentiell) mit einer Fehlermeldung. Umgekehrt wäre auch der Aufruf der Prozedur *Proc6* von *Proc2* aus unmöglich, obwohl *Proc6* auf der zweiten Ebene deklariert ist und damit nominell eine Ebene tiefer als *Proc2*, aber um von *Proc2* zu *Proc6* zu gelangen, müßte man zuerst nach *Proc1* und in das Modul aufsteigen, um über *Proc3*, *Proc4* und *Proc5* abzustiegen bis zu *Proc6*, ein Weg, der den Zugriff unmöglich macht. Aus dem selben Grund ist der Aufruf von *Proc5* aus der Prozedur *Start* heraus nicht möglich, es handelt sich um einen Aufstieg von der ersten auf die nullte mit anschließendem erneutem Abstieg auf die erste Ebene.

Eine Besonderheit bei Sichtbarkeiten muß hier erwähnt werden, die für Bezeichner der nullten Ebene gilt. Grundsätzlich sind diese Bezeichner wie alle anderen zu ihrer Ebene lokal, sie sind außerhalb des Moduls nicht sichtbar. Wie Sie aber bereits wissen, lassen sich diese Bezeichner exportieren, Sie haben Kommandoprozeduren wie *Start*, oder im Programm *Konvert1.odc* auch Variablen und Konstanten der nullten Ebene mit Exportmarken außerhalb des Moduls sichtbar gemacht. Diese Möglichkeit ist allerdings auf die nullte Ebene beschränkt, Bezeichner der ersten oder noch tiefer liegender Ebenen können weder exportiert noch sonstwie auf einer der höher liegenden Ebenen sichtbar gemacht werden.

Ein letzter, zu beachtender Punkt findet sich in dem (in Kommentarklammern stehenden) Aufruf von *Proc3* innerhalb von *Proc6*. Dieser Aufruf ist formal möglich, denn Sichtbarkeitsbereiche sind, wie erwähnt, nur in absteigender Hierarchie beschränkt, alle Bezeichner der in direkter Linie darüber liegenden Ebenen sind grundsätzlich sichtbar. Dies bedeutet, daß der Aufruf der Prozedur *Proc3* in *Proc6* ausgeführt würde. *Proc3* ruft ihrerseits *Proc5* auf, von wo *Proc6* (erneut) aufgerufen wird, die *Proc3* aufruft, von wo *Proc5* ... u.s.w. Es handelt sich um eine Folge von Aufrufen, die nie aufhören kann, eine Endlosschleife. Ein solcher rekursiv genannter Aufruf, also ein direkter oder indirekter Aufruf einer Prozedur durch sie selbst, kann ein nützliches Programmierwerkzeug sein, wie Sie an dem Programm *Recursio.odc*, das Sie im nächsten Abschnitt finden, sehen werden; es erfordert allerdings eine besondere Aufmerksamkeit des Programmierenden, da ein rekursiver Aufruf nur durch eine ausdrücklich formulierte Bedingung beendet wird.

### 5.7. Rekursion - Sich selbst aufrufende Prozeduren

Die folgenden Programme - *Recursio.odc* und *Recurs.odc* - sind weitgehend selbsterklärend, allerdings kommt es gerade bei dem Verständnis der Rekursion auf Feinheiten an. Versuchen Sie deshalb bitte, das Prinzip der Rekursion zuerst an der Datei *Recursio.odc* zu verstehen und sehen Sie anschließend im Vergleich das Programm *Recurs.odc* an.

Im Programm *Recursio.odc* finden Sie in der Kommandoprozedur *Start* den Aufruf der Prozedur *Selbstaufrufe*, der der Parameterwert 7 (der Wert der Konstanten *Anfang*) übergeben wird. Er ist hier wegen der Deutlichkeit mit einer explizit benannten Konstanten als Parameter formuliert worden, um noch einmal die Sichtbarkeit des global deklarierten Bezeichners *Anfang* in der Prozedur zu zeigen. Der Parameterwert kann der Prozedur nicht als VAR-Parameter übergeben werden, da es sich bei *Anfang* nicht um eine Variable handelt. Wäre im Kopf der Prozedur *Selbstaufrufe* der Parameter als Variablenparameter deklariert, gäbe es eine Fehlermeldung des Compilers, denn *Anfang* ist eine Konstante, während der Compiler bei einem VAR-Parameter eine Variable erwartet (Sie können dies ausprobieren, indem Sie die Kommentarklammern um VAR im Prozedurkopf entfernen). Dagegen kann ein Konstantenwert einem VAL-Parameter zugewiesen werden, da in diesem Fall lediglich eine Kopie des Wertes erzeugt und in der Parametervariablen gespeichert wird.

Wichtigster Punkt in der Prozedur *Selbstaufrufe* ist nun die IF-Abfrage, in der die Prozedur erneut aufgerufen wird. Sie erinnern sich, daß man einer Prozedur durch die Verwendung identischer Variablenamen in ihr und in übergeordneten Programmteilen den Zugriff auf die Variablen der höheren Programmebenen unmöglich machen kann. Dieses Prinzip gilt nicht nur für Variablen, sondern generell für alle Bezeichner, also auch für Prozedurnamen. Ein erneuter Aufruf derselben Prozedur innerhalb einer Prozedur ist somit für den Compiler die Eröffnung einer neuen Ebene, auf der alle in ihr deklarierten Bezeichner gleiche Namen der übergeordneten Ebenen verdecken.

Beachten Sie, daß die erneuten - rekursiven - Aufrufe der Prozedur *Selbstaufrufe* zwar jeweils als letzte Anweisung innerhalb der übergeordneten Prozedur, aber vor deren Beendigung erfolgen. Es handelt sich bei der Rekursion also um eine endlose Folge von ineinander geschachtelten identischen Prozeduren, ganz so wie die bekannten Folgen von Bild im Bild im Bild im Bild ... Anders aber als dort findet in diesem Fall keine Verkleinerung statt, die tiefer liegende Bilder ab einer gewissen Stelle unsichtbar macht. Es ist daher unbedingt nötig, daß die Rekursion durch eine explizit gesetzte Bedingung abbricht. Im Fall des vorliegenden Programms wird die Beendigung dadurch erreicht, daß vor jedem erneuten Durchlauf von *Selbstaufrufe* die Kontrollvariable *Index* vermindert, ihr aktueller Wert geprüft und der Neuaufwurf der Prozedur von diesem Wert abhängig gemacht wird, wobei die Rekursion für den Indexwert Null beendet ist.

Wichtig für das Verständnis von Rekursion ist weiter die Tatsache, daß bis zum Erreichen der Abbruchbedingung zwar eine ganze Reihe von Prozeduren geöffnet, aber noch keine einzige geschlossen wurde. Erst mit dem Eintreten der Abbruchbedingung wird das erste Mal das die Prozedur(en) schließende *END Selbstaufrufe* erreicht, und zwar von der als letzte geöffneten Prozedur zuerst. Ab jetzt werden, da der rekursive Aufruf am Ende jeder Prozedur erfolgte, von Innen nach Außen, in umgekehrter Abfolge zum Öffnen, die einzelnen Prozeduren geschlossen, die zuerst geöffnete also zuletzt.

Diese Tatsache ist wesentlich bei der Frage, an welche Stelle innerhalb der Prozedur Sie den rekursiven Aufruf setzen. Versuchen Sie, sich die unterschiedliche Wirkungsweise der Platzierung des rekursiven Aufrufs an den Bildschirmausgaben der Module *TutRecursio* und *TutRecurs* klar zu machen. Die als letzte geöffnete Prozedur wird auch im Modul *TutRecurs* zuerst geschlossen, was aber dazu führt, daß anders als im Modul *TutRecursio*, in dem die Ausgabeanweisungen der Prozedur *Selbstaufrufe* vor dem Rekursionsaufruf stehen, in diesem Fall die Ausgabe der zuletzt geöffneten Prozedur als erste geschrieben wird, denn bei *TutRecurs* stehen die Ausgabeanweisungen hinter dem rekursiven Aufruf. Somit werden die Bildschirmausgaben bei *TutRecursio* in der Reihenfolge der rekursiven Aufrufe geschrieben, im Modul *TutRecurs* jedoch in umgekehrter Reihenfolge. Auch müssen die Ausgabeanweisungen für die Werte von *Index* unterschiedlich sein, denn in beiden Programmen erfolgt die Dekrementierung von *Index* gleichermaßen vor dem Rekursionsaufruf, was zum Erreichen der Abbruchbedingung unumgänglich ist, jedoch geschieht sie im ersten Fall nach der Bildschirmausgabe, im zweiten vor ihr. Im Modul *Recurs* wird daher *Index* vor der Dekrementierung in die lokale Variable *Index1* kopiert, deren (unveränderter) Wert für die Bildschirmausgabe auch nach dem Rekursionsaufruf zur Verfügung steht.

Wie Sie sehen, ist das Grundprinzip der Rekursion stets gleich, Schwierigkeiten liegen nicht so sehr in diesem Prinzip als vielmehr darin, bei einem Algorithmus, der eine rekursive Problemlösung ergeben soll, den jeweils geeigneten Ort des Rekursionsaufrufs zu bestimmen. Machen Sie sich deshalb ganz deutlich klar, daß alles, was vor einem Rekursionsaufruf steht, beim Eintritt in jede einzelne Prozedur in der Reihenfolge der Prozeduraufrufe, dagegen alles, was nach dem Rekursionsaufruf innerhalb der rekursiven Prozedur steht, beim Verlassen der Prozeduren in umgekehrter Reihenfolge der Prozeduraufrufe stattfindet.

Einen weiteren wichtigen Aspekt der Rekursion können Sie kennenlernen, wenn Sie bei dem Programm *Recurs.odc* im Kopf der Prozedur *Selbstaufrufe* die Kommentarklammern um VAR entfernen. Da in diesem Fall *Anfang* als aktueller Parameter der Prozedur *Start* und *Index* als jeweils formaler und aktueller Referenz-Parameter aller *Selbstaufrufe* globale Variablen sind, werden Sie in der Bildschirmausgabe dieser Version des Programms einige Merkwürdigkeiten finden (versuchen Sie bitte, sich den Effekt der Änderung gedanklich klar zu machen, Sie können daran erkennen, ob Sie die nicht ganz einfachen Neuigkeiten dieses Kapitels verstanden haben). Das Programm erzeugt in diesem Fall eine gänzlich andere (und nicht gewünschte) Ausgabe als bei einem VAL-Parameter *Index*, da jede Veränderung des Wertes von *Index* stets dieselbe, nur ein Mal existierende Variable *Anfang* betrifft.

Bei einem VAL-Parameter dagegen besitzt jede neu geöffnete Prozedur *Selbstaufrufe* ihre eigene Variable *Index*, die einerseits zur jeweiligen Prozedur lokal, das heißt, in den zuvor geöffneten Prozeduren ebenso wie im Modul unsichtbar ist und daher keine Rückwirkungen auf den Wert der Variablen *Anfang* hat. Durch den Überdeckungseffekt, bei dem an sich sichtbare globale Variablennamen höherer Programmeebenen durch identische lokale Variablennamen in nachgeordneten Prozeduren unsichtbar werden, ist jede Variable *Index* aber auch in nachfolgend geöffneten Prozeduren, für die sie an sich global wäre, unsichtbar, sie ist also nur innerhalb ihrer jeweils eigenen Prozedur existent.

Wenn daher die Übergabe des Werts von *Anfang* an einen VAL-Parameter *Index* erfolgt, die Prozedur also lediglich eine Kopie des Wertes von *Anfang* erhält, bleibt in der Zeile *Aufruf* := *Anfang* + 1 - *Index* der Wert von *Anfang* bei jedem erneuten Aufruf der Prozedur konstant, die lokale Variable *Aufruf* hat deshalb in jeder geöffneten Prozedur *Selbstaufrufe* einen anderen Wert. Handelt es sich bei *Index* jedoch um einen VAR-Parameter, wird als "Nebeneffekt" der Wert von *Anfang* bei einer Änderung des Wertes von *Index* ebenfalls verändert, da *Anfang* und *Index* identisch sind. Somit erfolgt zwar der rekursive Aufruf der Prozedur *Selbstaufrufe* jeweils mit dem richtigen Wert von *Index*, was Sie daraus ersehen können, daß auch jetzt sieben Ausgabezeilen erscheinen, jedoch enthalten diese jedesmal denselben Wert von *Aufruf*, da sämtliche Ausgabeanweisungen erst erzeugt und geschrieben werden, nachdem alle rekursiven Aufrufe erfolgt sind. Sie sehen, daß bei der Rekursion kleine, scheinbar unwichtige Nebensächlichkeiten große Auswirkungen haben können.

Eine weitere Form der Rekursion finden Sie in dem Programm *Wechsel.odc*, in dem nicht eine Prozedur sich selbst erneut, sondern zwei verschiedene, *Hund* und *Floh*, sich wechselseitig aufrufen. Die Prozedur *Start* enthält im wesentlichen nur den Aufruf der Prozedur *Hund*, die eine *WHILE*-Schleife mit innenliegender *FOR*-Schleife enthält. Am Ende der *WHILE*-Schleife wird nun die Prozedur *Floh* aufgerufen, die innerhalb des Modul-Deklarationsteils der Prozedur *Hund* folgt, dieser daher nach Component Pascal Logik unbekannt ist. Selbstverständlich könnte man die Reihenfolge von *Hund* und *Floh* vertauschen, das Dilemma wäre aber nicht beseitigt, da die Prozedur *Floh* ihrerseits *Hund* aufruft.

Die Auflösung des Problems liegt in der "Vorwärts"-Deklaration von *Floh*, die Sie unmittelbar vor der Deklaration der Prozedur *Hund* finden. Formal besteht eine solche FORWARD-Deklaration aus der gesamten Signatur (dem Prozedurkopf) der vorwärts zu deklarierenden Prozedur, deren vollständiger Programmcode erst später deklariert wird. Die FORWARD-Deklaration erfolgt durch einen dem Schlüsselwort *PROCEDURE* nachgestellten symbolisierten Pfeil, das Zeichen "^" (Latin1-Zeichen 94). Dadurch wird die Prozedur *Floh* der vor ihr stehenden Prozedur *Hund* (sowie natürlich weiteren eventuell dort stehenden Prozeduren) zugänglich gemacht, kann also von *Hund* aufgerufen werden, obwohl ihr eigentlicher Inhalt an dieser Stelle noch unbekannt ist. Dies ist deshalb möglich, weil der Compiler wegen der FORWARD-Deklaration der Prozedur in der Lage ist, eine Speicheradresse einzurichten, die in dem Moment mit Inhalt gefüllt wird, in dem die eigentliche Prozedurdeklaration innerhalb des Programmablaufs auf-

taucht (einen anderen Datentyp, der ähnlich definiert ist, werden Sie in einem späteren Kapitel mit den sogenannten Zeigern kennenlernen).

Beachten Sie bitte, daß die Variable *Durchlauf* eine globale Variable ist. Guter Programmierstil "erfordert" es, eine Variable lokal zu machen, sie in derjenigen Prozedur zu deklarieren, in der sie gebraucht wird, da auf diese Weise Speicherplatz nicht unnötig belegt bleibt und die Programme übersichtlicher und weniger fehleranfällig werden, denn lokale Variablen sind sicherer gegen unbeabsichtigte Wertänderung als globale. Da die Variable *Durchlauf* zwar nur in der Prozedur *Floh* geändert, mit ihrem jeweils aktuellen Wert aber auch in der Prozedur *Hund* verwendet wird, muß sie in diesem Programm als globale Variable deklariert werden.

Von einer Prozedur verursachte Veränderungen globaler Daten können, wie bei der Variablen *Durchlauf*, erwünscht und notwendig sein, treten sie jedoch unbeabsichtigt auf, sind die Folgen möglicherweise fatal, Sie konnten derartig unbeabsichtigte Nebeneffekte bei der Änderung des VAL-Parameters *Index* der Prozedur *Selbstaufrufe* im Programm *Recurs.odc* in einen VAR-Parameter erleben.

Wie bei jeder Rekursion, muß auch im Modul *TutWechsel* eine Abbruchbedingung formuliert und deren Erreichen gesichert sein. In diesem Programm erfolgt der Abbruch, wenn die Variable *Durchlauf* den Wert Null annimmt. Experimentieren Sie ein wenig mit der Stellung der Anweisung *DEC(Durchlauf)*, indem Sie diese z.B. nach *Hund* verschieben und versuchen Sie, sich die Ergebnisse der Positionsänderung(en) klar zu machen. Ebenso sollten Sie die zu erwartenden Änderungen der Bildschirmausgabe handschriftlich erarbeiten, bevor Sie die in Kommentarklammern stehenden mit (\* a \*) bzw. (\* b \*) gekennzeichneten Anweisungen ausführen lassen.

## 5.8. Funktionen - Prozeduren mit Wertrückgabe

Eine spezielle Art von Prozeduren sind Funktionsprozeduren, manchmal auch nur Funktionen genannt, die Sie im Programm *Funktion.odc* kennenlernen. Kurz gesprochen sind Funktionen Prozeduren, die nach ihrem Ablauf einen Wert an die aufrufende Stelle zurückgeben. Ein solcher Wert kann selbstverständlich einer Variablen übergeben werden, ganz wie Sie es inzwischen gewohnt sind. Sie sehen diese Wertübergabe in der Prozedur *Start* in der Zeile *Tatzen* := *Viertatzler(Hunde, Katzen)*. Die Funktionsprozedur *Viertatzler* berechnet einen Wert, der von den Startwerten der Parameter *Hunde* und *Katzen* abhängt, dieser Wert wird anschließend der Variablen *Tatzen* übergeben. Nun ist *Tatzen* eine Variable des Typs *INTEGER*, der folglich nur *INTEGER*-Werte zugewiesen werden können. Diese Tatsache kommt darin zum Ausdruck, daß die Deklaration einer Funktionsprozedur eine wesentliche Änderung gegenüber der Deklaration einer "normalen" Prozedur enthält. In der Deklaration einer Funktionsprozedur steht hinter der schließenden

Klammer der Parameterliste die Deklaration des Typs, den der zurückgegebene Wert hat, genauso, wie dies bei einer Variablendeklaration der Fall ist.

Eine Funktionsprozedur ist im übrigen aufbaubar wie andere Prozeduren, kann also einen Deklarations- und einen Anweisungsteil mit allen darin möglichen Einzelheiten enthalten. Die hier vorgestellte Funktion *Viertatzler* ist sehr einfach gestaltet. Sie enthält im Anweisungsteil außer einigen Schreibanweisungen die Zeile `RETURN 4 * (Nummer1 + Nummer2)`. Wenigstens eine Anweisung dieser Art ist bei jeder Funktionsprozedur unbedingt erforderlich, denn die aufrufende Stelle erwartet eine Wertrückgabe. Die Bereitstellung dieses Wertes erfolgt innerhalb der Funktion durch das Schlüsselwort `RETURN`, gefolgt von dem explizit oder implizit angegebenen Wert, der an die aufrufende Stelle zurückgegeben werden soll. Nun können innerhalb der Funktion mehrere verschiedene Werte berechnet werden, von denen in Abhängigkeit von anderen Bedingungen dieser oder jener zurückgegeben werden soll. Dementsprechend können mehrere `RETURN`-Anweisungen in der Funktionsprozedur stehen, aber Sie müssen als Programmierende durch geeignete Auswahlkriterien dafür sorgen, daß stets nur einer der möglichen Fälle eintritt, denn es kann an die aufrufende Stelle selbstverständlich nur ein einziger Wert zurückgegeben werden.

Das Schlüsselwort `RETURN` hat eine weitere wesentliche Funktion, die nicht nur im Zusammenhang mit Funktionen, sondern generell für Prozeduren gilt, es beendet (vorzeitig) die Ausführung einer Prozedur. Sie können dies an der Bildschirmausgabe von *TutFunktion* sehen, bei der weder die letzte Anweisung der Funktionsprozedur *Viertatzler* noch die dem `RETURN` folgenden Anweisungen der Prozedur *Start* ausgeführt werden. Ändern Sie in der Funktion *ok()* die Zeile `RETURN TRUE` in `RETURN FALSE`, ändert sich die Bildschirmausgabe der Prozedur *Start* entsprechend, die Funktion *Viertatzler* wird immer noch ausgeführt, das heißt, der entsprechende Text wird geschrieben und der Rückgabewert "28" wird der Variablen *Tatzen* übergeben, aber die Ausgabe des Ergebnisses ("*Insgesamt laufen hier 28 Tatzen herum.*") erfolgt nicht, da jetzt in *Start* nicht der `THEN`-Teil der Verzweigung durchlaufen wird, sondern der `ELSE`-Teil, in dem `RETURN` die Ausführung der Prozedur beendet.

Bei der Verwendung von `RETURN` liegt der Unterschied zwischen einer Funktion und einer (eigentlichen) Prozedur darin, daß in einer Funktion `RETURN` einen Wert zurückgibt, während bei einer Prozedur `RETURN` ohne Zusatz die weitere Ausführung der Prozedur beendet. Wollten Sie also aus irgendeinem Grund einmal eine Prozedur mit einem "Nothalt" verlassen, was allerdings bei einem sauberen Programmierstil unnötig sein sollte, können Sie dazu `RETURN` verwenden.

Einen weiteren wichtigen mit Funktionsprozeduren verbundenen Aspekt sehen Sie an der Funktion *ok()*, das im Prozedurnamen *ok()* stehende leere Klammerpaar. In Component Pascal benötigt jede Funktionsprozedur eine Parameterliste, die gegebenenfalls leer sein kann, aber auch in diesem Fall durch die Klammern repräsentiert sein muß. Wie Sie außerdem in der Prozedur *Start* an der Bedingung `IF ~ok() THEN RETURN END` sehen, ist für den Compiler eine (leere oder nichtleere) Parameterliste sowohl in der Signatur einer Funktion als auch bei ihrem Aufruf nötig. Sollten Sie beim Aufruf einer Funktion die Para-

meterliste vergessen, was natürlich hauptsächlich bei einer leeren Liste geschehen kann, erhalten Sie beim Kompilerversuch eine entsprechende Fehlermeldung.