Einführung in Object Pascal

©by Uwe Schächterle
www.Corpsman.de

21. August 2013

Dieses Tutorial erhebt keinerlei Anspruch auf Vollständigkeit und oder Korrecktheit.

Die behandelten Themen sollen in erster Linie den Einstieg in die Programmiersprache Object Pascal (Delphi/FPC) ermöglichen.
Es werden hauptsächlich die zur Bearbeitung der gestellten Aufgaben benötigten Grundlagen erläutert.

Die Aspekte der Objektorientierung sind hierbei nicht notwendig und werden daher nicht eingeführt.

Inhaltsverzeichnis

1 Datentypen

Ein Computer arbeitet mit Nullen und Einsen, das ist alles was ein Computer versteht und womit er arbeiten kann. Ein Zeichen im Computer kann also genau 2 Werte annehmen: 0 und 1. Je nach Architektur werden diese Zeichen zu Gruppen a 32 beziehungsweise 64 Zeichen zusammengefasst (ältere Architekturen, oder Sonderbaureihen können hiervon natürlich stark abweichen).
Alles was im PC passiert, wird durch diese Zeichenketten dargestellt.
Für das vorliegende Programm ¨QProgrammer¨ nutzen wir nur eine sehr kleine Auswahl dieser Datentypen. Diese Auswahl ist vollkommen ausreichend zur Lösung der Aufgaben, bzw. zur Vermittlung der notwendigen Grundlagen.
Der einfachste und kleinste Datentyp ist der **Boolean** Datentyp. Dieser besteht genau aus 2 Werten, kann also mit nur einem Zeichen dargestellt werden. **Boolean** besteht aus den Werten **true** = wahr und **false** = falsch. Wie leicht zu ersehen ist, wird mithilfe von **Boolean** alles dargestellt was eine Aussage bezüglich Ja / Nein, Wahr / Falsch oder sonstige Entscheidungen mit nur 2 Wahlmöglichkeiten betreffen. Dem Computer ist hierbei völlig egal wie der **Boolean** Wert interpretiert wird. Für ihn ist der Wert von **Boolean** stets immer nur wahr oder falsch bzw. in der internen Darstellung 0 oder 1.
Den 2. und letzen Datentyp den wir für unsere Implementierungen noch benötigen ist der **Integer** Datentyp. In ¨QProgrammer¨entspricht dieser Datentyp einer Kette von 32 Zeichen. Diese Kette wird als vorzeichenbehaftete Binärzahl mit 32 Stellen betrachtet. Somit entspricht der Datentyp **Integer** einem Wertebereich von $-2^{31} = -2147483648$ bis $2^{31} - 1 = 2147483647$ (die minus 1 kommt daher, daß man die 0 als positive Zahl wertet) aus der mathematischen Menge $\mathbb{Z}$ (Wen es interessiert wie die Werte -2^{31} und $2^{31} - 1$ zustandekommen, dem seien die Grundlagen der Binärzahlen, so wie deren 2er Komplement Darstellung ans Herz gelegt).
Selbstverständlich gibt es in der Welt der Computer noch deutlich mehr Datentypen. Als Beispiele seien hier die Fließ kommazahlen (**real**,**double**,**extended**), die Verbunde (**record**) sowie Felder (**array**) und Klassen (**class**) genannt.

2 Variablen

Nachdem wir uns nun mit den Datentypen beschäftigt haben, wollen wir diese auch benutzen. Um dies zu können, benötigen wir Variablen. Hierbei ist der Begriff Variable genau derselbe wie in der Mathematik. Das heisst, will man in einem Programm irgendwo eine Variable benutzen, muss diese vorher mit ihrem Namen und ihrem zugrundeliegenden Datentyp definiert werden. In Object Pascal nutzen wir hierzu das Schlüsselwort **var**. Variablen dürfen aber nicht an jeder beliebigen Stelle im Code eingefügt werden. In ¨QProgrammer¨ gibt es aufgrund der vielen Einschränkungen hierfür nur eine mögliche Stelle.

- vor dem Schlüsselwort **begin** innerhalb einer Funktion, dies gilt auch für genestete Funktionen (siehe Kapitel 7)

Jedoch darf man den Namen einer Variablen nur bedingt frei wählen. Eine Variable darf z.B. nicht mit einer Zahl beginnen, auch darf man keine Variablen definieren die gleich einem Schlüsselwort (eine Liste der in ¨QProgrammer¨benutzen Schlüsselworte befindet sich am Ende dieses Dokumentes) heissen. Die Groß -Kleinschreibung ist in Object Pascal nicht von Relevanz. Über die geeignete Wahl der Variablennamen gibt es ettliche Styleguids und andere Dokumente. Aus diesem Grund sei hier nur folgender Rat : ¨Geben Sie ihren Variablen eindeutige Namen, welche ungefähr beschreiben, wofür sie diese Variable einsetzen. Ein Variablennamen wie : ¨x,y,a1,a2..¨ ist nicht unbedingt der Richtige.¨.
Einige gültige Variablenamen könnten somit lauten : ¨Nenner, Zaehler, Laufvariable ..¨. Da Object Pascal aus dem englischen Sprachraum stammt, sind Zeichen wie ¨ä¨ und ¨ß ¨ ebenfalls nicht erlaubt. Am Besten ist es, wenn die Variablennamen aus der Menge $\{a..z, A..Z, 0..9\}$ gewählt werden. Die Sprachdefinition von Object Pascal lässt hierbei noch ein paar Zeichen mehr zu. Für genauere Informationen sei auf die Spezifikation von FPC verwiesen. Der Aufbau einer Variablendeklaration ist immer derselbe. Zuerst kommt das Schlüsselwort **var** gefolgt von einer Liste der zu definierenden Variablennamen, getrennt durch ein ¨**,**¨ und abgeschlossen durch ein ¨**:**¨ gefolgt vom jeweils gewählten Datentyp, hiernach muss ein ¨**;**¨ folgen. Es folgen einige Beispiele :

```
var Variable1, Variable2:Integer;
    BoolscherAusdruck:Boolean;
```

3 Kommentare

Kommentare sind nicht direkt ein Mittel der Programmierung, denn sie werden vom Compiler wieder entfernt. Dennoch sind Kommentare ein sehr wichtiges Instrument der Programmierung, denn ohne sie müsste eine spätere Nachbearbeitung des Quellcodes stets eine entsprechend aufwendige Analyse vorangehen. Diese ist aufwendig, fehlerbehaftet und zeitraubend. Somit tut man sich selbst und allen anderen, die den Code lesen einen groß en Gefallen, wenn ein Code ausreichend und gründlich dokumentiert wird. Hierbei ist natürlich zu beachten, dass nur sinnvolle Kommentare gegeben werden. Eine Überkommentierung kann ebenfalls negativ sein. Jeder Programmierer entwickelt dabei im Laufe der Zeit seinen eigenen Stil. Object Pascal unterstützt 3 Verschiedene Arten der Kommentierung.

- ¨//¨ ist ein Einzeiliger Kommentar, ab da wo dieses Zeichen steht, bis zum Ende der aktuellen Zeile, wird alles automatisch zum Kommentar
- ¨ (* *) ¨ist ein Kommentar der über beliebig lange Textstellen geht, oder auch nur ein Zeichen beinhalten kann.
- ¨{}¨ siehe ¨ (* *) ¨

4 Zuweisungen

Nun wollen wir natürlich auch mit unseren Variablen arbeiten. Hierzu widmen wir uns dem Sprachaufbau von Object Pascal.
Ein Programm besteht hierbei immer aus verschiedenen Grundblöcken

- Variablendeklaration
- Zuweisung
- Schleifen, bedingte Codeausführungen
- Funktionen und deren Deklarationen

Jeder dieser Grundblöcke muss durch ein ¨**;**¨ getrennt werden.
Der einfachste Grundblock, den es neben der bereits bekannten Variablendeklaration gibt, ist der Zuweisungsblock. In diesem Block kann einer Variable ein konstanter, oder ein arithmetischer Wert zugewiesen werden. Eine Zuweisung gestaltet sich immer gleich. Links steht die Variable, der der Wert zugewiesen wird dann folgt das Zeichen ¨**:=**¨ gefolgt vom zuzuweisenden Ausdruck. Die Zuweisung wird dann mittels ¨**;**¨ abgeschlossen. Als Ausdruck kann alles in Frage kommen, was als Resultat den selben Variablentyp hat wie die zugewiesene Variable. Im Ausdruck sind selbstverständlich auch andere Funktionsaufrufe erlaubt. Kommt die zugewiesene Variable im rechten Teil ebenfalls vor, so wird hier der Wert den die Variable vor der Zuweisung hat genommen. Object Pascal unterstützt auch die entsprechende Klammerung. Die Bindungsstärke der einzelnen Operatoren ist hierbei vergleichbar der der Mathematik. Bei Unklarheiten sollte aber auf jedenfall Klammern gesetzt werden (Zu viele Klammern können hier nicht schaden). Wichtig bei den Ausdrücken ist natürlich, dass man weiss welcher Operator welche Typen unterstützt bzw. zurückgibt. In ¨QProgrammer¨ werden drei verschiedene Operatorentypen unterschieden :

- Operatoren die Boolean auf Boolean (z.B. **and**, **or**) abbilden
- Operatoren die Integer auf Boolean (z.B. $<$, $>=$) abbilden
- Operatoren die Integer auf Integer (z.B. $+$, **mod**) abbilden

Bevor wir uns die genauen Definitionen der einzelnen Operatoren ansehen: Hier erst ein mal ein paar allgemeine Beispiele :

```
x := y;
x := y + f(g)*x;//f(g) ist hierbei ein Funktionsaufruf
x := x + 1;      // Erhoehe x um 1
```

Es folgen die Wertetabellen der einzelnen Operatoren. Diese sind nur der Vollständigkeit wegen angegeben, und können von erfahrenen Mathematikern auch gerne übersprungen werden.

4.1 Operatoren die Boolean auf Boolean abbilden

Die boolschen Operatoren sind die jenigen die einen Boolean-Wert wieder auf einen Boolean-Wert abbilden. Da Boolean nur 2 Werte besitzt, ist es am einfachsten die Ergebnisse der Operatoren entsprechend als Wahrheitstabellen anzugeben. Hierbei entspricht der Wert ¨0¨ falsch und ¨1¨ dem Wert wahr. Q steht für das Ergebniss und X_i für die Parameter des jeweiligen Operators.

not

X	Q
0	1
1	0

and

X_1	X_0	Q
0	0	0
0	1	0
1	0	0
1	1	1

or

X_1	X_0	Q
0	0	0
0	1	1
1	0	1
1	1	1

xor

X_1	X_0	Q
0	0	0
0	1	1
1	0	1
1	1	0

4.1.1 Short circuit Operatoren

Das in diesem Abschnitt behandelte Thema, stellt eine Optimierung dar und sollte von einem Absoluten Beginner erst ein mal übersprungen werden.

In der Programmentwicklung geht es nicht nur darum ein Programm zu schreiben, welches eine Vorgegebene Aufgabe bewältigen kann. Es sollte auch stets versucht werden diese Aufgabe möglichst schnell und effizient zu lösen. Aus diesem Grund gibt es die sogenannten Short circuit Operationen. Wie der Name schon suggeriert wird hierbei eine “Abkürzung” genommen. Dies bedeutet, wenn der Ausführende Code feststellt, dass zur Evaluierung eines Boolschen Ausdrucks, keine weiteren Berechnungen mehr notwendig sind, weil das Endergebniss bereits feststeht, so kann dieser die Berechnung der noch ausstehenden Teilausdrücke komplett überspringen. Dies darf der Compiler, da er davon ausgehen darf, dass die Übersprungenen aufgerufenen Funktionen frei von Nebeneffekten sind (der interessierte Leser schlage dies im Drachenbuch nach).

In QProgrammer gibt es 2 Operatoren, welche Short circuit Operatoren sind. Das Boolsche **and** und **or**. Bei der Short circuit Evaluation gelten jedoch strenge Regeln, derer sich ein Programmierer bewust sein mus. Im Groben sollen diese hier nun erläutert werden.
Ein Short circuit Operator ist Textuel definiert als

op1 **relop** op2

Hierbei steht op1 und op2 für den jeweils zu evaluierenden Boolschen Teilausdruck und **relop** für den Operator (im speziellen bei ¨QProgrammer¨um **and** und **or**).
Der Operator op1 wird immer berechnet und ausgewertet, anhand seines Wertes kann der ausführende Code dann entscheiden, ob die Berechnung von op2 noch notwendig ist, oder nicht. Zur Verdeutlichung sei hier anhand des **and** Operators ein Beispiel gegeben:

Der aus zu Wertende Ausdruck sei hierbei "x **:=** $(4 < 3)$ **and** $(5 > 1)$"
Wie man schnell sehen kann wird x der Wert '0' = falsch zugewiesen. Mittels Short circuit Evaluation geschieht dies ebenfalls, allerdings ohne die Berechnung des Ausdruckes $5 > 1$. Dies geschieht, weil der Ausdruck $(4 < 3)$ bereits ein falsch ergibt, und wie man in 4.1 sehen kann, wertet sich ein **and** Ausdruck immer zu falsch aus, wenn bereits einer der Beiden Operatoren einen falsch Wert besitzt. Somit kann die zweite Berechnung gespart werden.

Würde man allerdings stattdessen "x **:=** $(5 > 1)$ **and** $(4 < 3)$"rechnen, so könnte die Ausführung nicht abgekürzt werden, da der erste Operand zu '1' = wahr ausgewertet wurde, und der Gesamtausdruck somit vom 2. Operanden abhängig ist.

Die Auswirkungen des Nebeneffektes sei an folgendem Beispiel demonstriert:

```
Function WasWirdBerechnet(a,b:Boolean):Integer;
var r:Integer;

  function f1(Dummy:Boolean):boolean;
  begin
    result := Dummy;
    r := r + 1;
  end;

  function f2(Dummy:Boolean):boolean;
  begin
    result := Dummy;
    r := r + 2;
  end;

begin
  r :=;
  if(f1(a) and f2(b))then
    r := - r;
  result := r;
end;
```

Wer nun glaubt das diese Funktion stets den Wert 3 oder -3 zurückgibt, je nach Parameter, der hat Short circuit sowie die Nebeneffekte noch nicht verstanden.

Tatsächlich sind die Werte 3,-3, 1 möglich, dies zu evaluieren sei dem Leser selbst überlassen.

4.2 Operatoren die Integer auf Boolean abbilden

Operatoren die Integer auf Boolean abbilden werden im Allgemeinen Vergleichsoperatoren genannt. Sie sind alle binäre Operatoren. Dies bedeutet, dass sie zwei Eingabewerte benötigen und einen Boolean-Ausdruck als Ergebnis liefern. Die Bedeutung der Operatoren entspricht ihrer intuitiven Bedeutung, sodass hier lediglich alle in ¨QProgrammer¨ verfügbaren Operatoren aufgelistet werden.

$$=, <>, >, <, >=, <=$$

Gesprochen: gleich, ungleich, größ er, kleiner, größ ergleich, kleinergleich

Beispiel :
$9 > 5 = \text{true}$

4.3 Operatoren die Integer auf Integer abbilden

Die reinen Integer Operatoren können im Prinzip als die intuitiven Operatoren aus $\mathbb{Z}$ angesehen werden. Neben den Operatoren

$$*, +, -$$

gibt es aber noch zwei weitere Operatoren

div und **mod**.

Die beiden Operatoren **div** und **mod** wollen wir uns ein kleinwenig genauer ansehen, da sie nicht unbedingt jedem geläufig sein dürften.

Der **div** Operator ist hierbei noch der Einfachere. In Worten berechnet der **div** Operator die nach unten (Richtung 0) abgerundete Division zweier Zahlen. In mathematischer Schreibweise entspricht dies : x **div** $y := \lfloor x : y \rfloor$.

Der **mod** Operator berechnet den Rest der sich durch die ganzahlige Division zweier positiven Zahlen ergibt. In mahematischer Schreibweise ist dies : x **mod** $y := x - (x$ **div** $y) * y$. Eine andere gleichweirtige Definition lautet: $x = n * y + r$ dabei ist n stets so zu wählen, dass $r < y$ gilt. Ist dies der Fall so ist $r \equiv x$ **mod** y.

In der Mathematik gibt es noch weitere Definitionen (alle Gleichwertig), der Interessierte Leser suche nach den Begriffen “Faktorraum” bzw. “Restklassenring”.
Die genaue Funktion des **mod** Operators läst sich am einfachsten anhand einiger Beispiele demonstrieren.

10 **div** $5 = 2$, weil die 5 ohne Rest genau 2 mal in 10 passt
10 **mod** $5 = 0$, weil die 5 ohne Rest genau 2 mal in 10 passt
9 **div** $5 = 1$, weil 5 nur 1 mal komplett in die 9 passt, Rest 4
9 **mod** $5 \equiv 4$, weil 9 **mod** $5 \equiv 9 - \underbrace{(9 \textbf{ div } 5)}_{=1} * 5 = 4$

Ergibt sich x **mod** $y \equiv 0$ so erkennen wir, dass y ein Teiler von x ist.

Exkursion:
Ist im oben gezeigten Beispiel y eine Primzahl, so ist x **mod** y ein Körper in mathematischer Schreibweise $\mathbb{Z}/n\mathbb{Z}$

5 Bedingte Codeausführung

Nun, da wir alle mathematischen Grundlagen geschaffen haben, können wir uns dem sogenannten Programmfluss widmen. Als Erstes betrachten wir hierzu die bedingten Codeausführungen. In Object Pascal ist hierfür das **if then else** Konstrukt vorgesehen. Dieses ermöglicht uns in Abhängigkeit eines boolschen Ausdruckes unterschiedlichen Code auszuführen. ¨QProgrammer¨unterstützt verschiedene Formen des **if then else** Konstruktes. Es sei aber angeraten nur die ersten beiden Varianten zu benutzen, da diese auch für unerfahrene Programmierer leicht verständlich sind und Fehler vermeiden (siehe “dangling else”).

o
```
if boolscherAusdruck then begin
  DoSomething
end else begin
  DoSomethingOther
end;
```

o
```
if boolscherAusdruck then begin
  DoSomething
end;
```

o
```
if boolscherAusdruck then
  DoSomething     // Nur 1 Befehl dieser endet mit ;
```

o
```
if boolscherAusdruck then
  DoSomething     // Nur 1 Befehl dieser endet ohne ;
else begin
  DoSomethingOther
end;
```

o
```
if boolscherAusdruck then begin
  DoSomething
end else
  DoSomethingOther// Nur 1 Befehl dieser endet mit ;
```

o
```
if boolscherAusdruck then
  DoSomething      // Nur 1 Befehl dieser endet ohne ;
else
  DoSomethingOther// Nur 1 Befehl dieser endet mit ;
```

Mit den oben gezeigten ersten beiden Varianten können sämtliche anderen Varianten nachgebildet werden. Sie unterscheiden sich lediglich in einer etwas ausführlicheren Schreibweise.

6 Schleifen

Schleifen sind der Teil der Computer-Programme, die die meiste Zeit beanspruchen. Dies bedeutet, dass wenn ein Programm auf einem Computer ausgeführt wird, sich dieses mit sehr hoher Wahrscheinlichkeit gerade in einer Schleife befindet. ¨QProgrammer¨ bietet drei verschiedene Schleifentypen an.

- die **for to** Schleife
- die **for downto** Schleife
- die **while** Schleife

Mit allen drei Varianten kann man im Prinzip dasselbe machen. Sie unterscheiden sich lediglich in kleinen Punkten. Dennoch macht es Sinn hier drei verschiedene Varianten zu haben, denn jede hat einen entscheidenden Vorteil bezüglich ihrer Semantik gegenüber den anderen beiden. Ihnen gemeinsam ist, dass sie dazu dienen einen bestimmten Codeabschnitt mehrfach zu durchlaufen.

6.1 Die for to Schleife

Die **for to** Schleife ist folgendermaß en aufgebaut :
Nach dem einleitenden **for** kommt eine Zuweisung ohne "**;**", welche die Schleifenvariable angibt. Auf diese Schleifenvariablen darf im Schleifenrumpf nur lesend zugegriffen werden und sie muss vom Typ Integer sein. Jeweils zu Beginn eines Schleifendurchlaufes wird die Schleifenvariable neu berechnet und dann der Schleifenrumpf ausgeführt. Die Schleifenvariable ändert sich bei jedem Schleifendurchlauf um eins. Gefolgt von der Zuweisung kommt das Schlüsselwort **to**. Dieses Schlüsselwort gibt an, das die Schleife aufsteigend läuft. Der nun folgende Ausdruck bestimmt die obere Grenze der Schleifenvariable. Übersteigt die Schleifenvariable den Wert dieses Ausdruckes, wird der Schleifenrumpf nicht mehr ausgeführt. Nach dem Schlüsselwort **do** folgt der Schleifenrumpf. Dieser kann ebenfalls wie bei der **If then else** Struktur aus nur einem Befehl bestehen, oder aus mehreren welche dann mittels **begin end** markiert sind. Ein Beispiel für eine **for to** Schleife sei :

```
for Zaehlvariable := x to y do begin
  DoSomething
end;
```

Ist der Wert von $x > y$ wird die Schleife gar nicht ausgeführt. Bei $x = y$ wird die Schleife genau einmal ausgeführt.

6.2 Die for downto Schleife

Die **for downto** Schleife unterscheidet sich von der **for to** Schleife lediglich darin, dass das Schlüsselwort **to** durch **downto** ersetzt wird. Dies hat zur Folge, dass die Schleifenvariable rückwärts läuft. In ¨QProgrammer¨ ist diese Funktion nicht wirklich notwendig. Im allgemeinen Object Paskal gibt es aber auch den

Datentyp Felder und bei diesen ist es in bestimmten Fällen durchaus vom Vorteil sie rückwärts zu durchlaufen. Ein Beispiel für eine **for downto** Schleife sei :

```
for Zaehlvariable := x downto y do begin
  DoSomething
end;
```

Ist der Wert von $x < y$ wird die Schleife gar nicht ausgeführt. Bei $x = y$ wird die Schleife genau einmal ausgeführt. In den meisten Fällen werden die **for to** Schleifen vom Compiler während der Optimierung in **for downto** Schleifen umgewandelt. Der Grund dafür liegt in der Tatsache dass die meisten Architekturen eine Variable nur auf 0 prüfen können, nicht aber auf einen Wert $<> x, mitx \neq 0$. Der Interessierte Leser sei zum Thema Compilerbau auf das Drachenbuch verwiesen.

6.3 Die while Schleife

Wie wir bei der **for to** und der **for downto** Schleife gesehen haben, wird die Zaehlvariable stets immer nur um 1 verändert. Nun gibt es aber auch durchaus Aufgabenstellungen bei denen die konkrete Anzahl der benötigten Wiederhohlungen nicht bekannt sind. Auch kann es sein, dass man die Schrittweite der Schleifenvariable ändern möchte. All dies ermöglicht die **while** Schleife. Sie ist wie folgt definiert :
Beginnend mit dem Schlüsselwort **while** und gefolgt von einem boolschen Ausdruck (der die Abbruchbedingung darstellt) folgt noch das Schlüsselwort **do**. Hiernach steht dann der Schleifenrumpf der wieder aus nur einem Befehl ohne **begin end;**, oder mehreren Befehlen eingeschlossen in einen **begin end;** Block bestehen kann. Der Schleifenrumpf wird so lange ausgeführt bis der boolsche Ausdruck zwischen **while** und **do** zu false ausgewertet werden kann.

!! **Achtung** !!
Der Programmierer muss sicherstellen, dass der boolsche Ausdruck tatsächlich irgendwann einmal zu false ausgewertet wird. Andernfalls läuft das Programm unendlich lange und wird somit nie enden.

Der boolsche Ausdruck wird immer beim Betreten und nach jedem Schleifendurchlauf erneut ausgewertet. So addiert folgendes Beispiel alle geraden Zahlen zwischen 0 und 10:

```
Ergebnis := 0;
Zaehler := 0;
while zaehler < 10 do begin
  Ergebnis := Ergebnis + Zaehler;
  Zaehler := Zaehler + 2;
end;
// Ergebnis hat nach dem Beenden der Schleife den Wert
// 0 + 2 + 4 + 6 + 8 = 20
```

7 Funktionen

Den letzen Grundblock den wir nun betrachten wollen, sind die Funktionen. Im Object Paskal gibt es durchaus noch weitere Grundblöcke wie z.B. Prozeduren. Diese sind aber für unsere Aufgabenstellungen nicht weiters interessant, bzw. in ¨QProgrammer¨nicht implementiert. Eine Funktion in Object Paskal ist im Prinzip genau dasselbe wie die mathematische Definition einer Funktion. Bis auf die in 4.1.1 angesprochenen Nebeneffekte, diese gibt es in der Mathematik nicht. Eine Funktion hat einen Namen, eine Menge von Übergabeparametern und ein Ergebnis, welches sie zurückgibt. Diese Eigenschaften werden natürlich noch von den Object Paskal spezifischen Sprachdefinitionen umgeben. Der Aufbau ist hierbei immer derselbe. Als Erstes kommt das Schlüsselwort **function** gefolgt vom Funktionsnamen. In runden Klammern können dann die entsprechenden Übergabeparameter angegeben werden. Es gibt auch Funktionen ohne Parameter. Hier können dann die runden Klammern komplett weg fallen. Abschließ end folgt ein ¨**:**¨ und der Typ des Rückgabeparameters. Innerhalb des Funktionsrumpfes ist dann stets eine Variable namens ¨result¨ verfügbar. Diese entspricht dem Rückgabewert und muss nicht extra mittels **var** definiert werden. So entspricht die mathematische Formel $f(x) = x^2$ folgendem Code :

```
function f(x:integer):integer;
begin
  result := x * x;
end;
```

Wir sehen an diesem Beispiel auch gleich wie die Übergabeparameter definiert werden. Dies geschieht im Prinzip genau gleich wie die Definition von Variablen. Lediglich das Schlüsselwort **var** braucht nicht extra angegeben werden. In Object Paskal ist das extra Angeben von **var** in der Übergabeparameterliste möglich. Dies hat aber eine spezielle Bedeutung auf die wir hier nicht eingehen wollen. In ¨QProgrammer¨ ist die Definition der Übergabeparameter mittels **var** daher nicht gestattet. Variablen von unterschiedlichen Typen können ebenfalls mittels **;** von einander getrennt und definiert werden. Die Auswirkung von **var** verändert den Übergabemodus, hierzu sei auf die Stichworte “Callbyname” und “Callbyreference” im Drachenbuch verwiesen. Werden noch zusätzliche Variablen benötigt, so können diese zwischen der Deklaration des Ergebnisstypes der Funktion und dem **begin** deklariert werden. Hierbei muss allerdings wieder das Schlüsselwort **var** angegeben werden. Bevor wir uns nun der Rekursion widmen, wollen wir noch geschwind den Begriff der genesteten Funktion erläutern. Es handelt es sich dabei um eine ¨Unterfunktion¨ einer Funktion. Dies Bedeutet, dass eine Funktion selbst noch eine Funktion hat welche im Selben Abschnitt wie Variablendeklarationen steht. In ¨QProgrammer¨ist die Nestung von Funktionen die einzige Möglichkeit mehr als nur eine einzige Funktion im Source zu implementieren.

Hier ein Beispiel für eine genestete Funktion:

```
// Die aeussere Funktion
Function aussen(n:integer):integer;
// die genestete Funktion
  Function ab(a,b:integer):integer;
  begin
    result := a + b;
  end;
// Der Funktionsrumpf von aussen
begin
  if n > 0 then
    result := ab(n, 1)
  else
    result := 1;
end;
```

7.1 Sichtbarkeit von Variablen

Im Kontext der Nestung von Funktionen muß auch die Sichtbarkeit von Variablen angesprochen werden. Unter der Sichtbarkeit einer Variablen ist folgendes zu verstehen :
Eine Variable kann deklariert aber dennoch nicht "Sichtbar" sein. Eine Zuweisung oder ein Lesen einer Variable ist nur erlaubt/ möglich, wenn diese Sichtbar ist. gleichnamige variablen können sich verdecken, dann ist stets die "innerste"/ "nächste" Variable sichtbar. Folgendes Beispiel soll die Sachlage verdeutlichen :

```
Function Ebene_1(x:Integer):Integer;
// Die Variablen v1 und x sind sichtbar
var v1:Integer;
  Function Ebene_2(x:Boolean):Boolean;
// Variablen v1,v2 sind sichtbar
  var v2:Integer;
// Variable x:integer ist nicht sichtbar
// sie wird von x:Boolean Ueberdeckt
  begin
    result := false;
  end;
// Die Variablen v1 und x:integer sind sichtbar
// v2 ist nicht mehr sichtbar, da Ebene_2 beendet wurde
var v3:Integer;
    b :Boolean;
begin
  b := Ebene_2(true);
```

```
  result := 0;
end;
```

Merksatz : Eine Variable gilt immer nur im deklarierten Block, und immer nur ab der Deklaration bis zum Ende des Dokumentes.

7.2 Rekursion

Das für den Programmierneuling wohl ungewöhnlichste Konstruckt, ist die Rekursion. Der Mathematiker kennt es bereits, da auch in der Mathematik sehr häufig rekursive Definitionen vorkommen. In der Informatik, genau wie in Object Paskal sowie ¨QProgrammer¨ ist die Rekursion gleich definiert. Die Umsetzung der Rekursion im Allgemeinen ist allerdings sehr rechenaufwendig, daher sollten ressourcenschonende Programmierer stets versuchen die Rekursion durch eine Schleife zu ersetzen. Wie dies geht füllt allerdings ganze Lehrbücher und kann deswegen hier nicht erläutert werden. Die Rekursion ist ein fortgeschrittenes Thema. Da sie in ¨QProgrammer¨ allerdings umgesetzt wurde, sei sie hier anhand des Beispiels der Fakultätsfunktion einmal gezeigt. Die in der Mathematik verwendete Fakultätsfunktion ist wie folgt für $n >= 0$ definiert :

$$\begin{aligned} 0! &= 1 \\ n! &= (n-1)! * n \end{aligned}$$

Natürlich gibt es auch andere Definitionen für die Fakultätsfunktion, aber wir benötigen für unser Beispiel die Rekursive. Eine entsprechende Umsetzung in Code sieht dann wie folgt aus:

```
function fakultaet(n:Integer):integer;
begin
  if n < 2 then
    result := 1
  else
    result := fakultaet(n-1)*n;
end;
```

Man kann sehen, dass die mathematische rekursive Definition ebenfalls sehr einfach mittels Code darstellbar ist. Genau wie bei der **while** Schleife benötigen wir auch hier eine Abbruchbedingung. Die das Rekursive aufrufen beendet. Geschieht dies nicht, so würde der Computer die Funktion unendlich oft Rekursiv aufrufen und ebenfalls nie beenden können. In der Praxis bedeutet ein rekursiver Funktionsaufruf jedoch, dass im Speicher Platz benötigt wird. Bei einer unendlichen Aufrufkette geht dem Computer dieser Speicher recht schnell aus. Anhand dieser Tatsache stürzt das Programm dann ab und wird auch beendet, nur eben nicht mit dem gewünschten Ergebnis.

8 Debugger

In heutigen Entwicklungsumgebungen gibt es eigentlich auch immer einen Debugger. Ein Debugger ist ein Programmteil der Entwicklungsumgebung und ermöglicht es dem Programmierer Schritt für Schritt den Programmablauf zu verfolgen. In der Regel hat man dabei die Einsicht auf alle Variablen welche gerade benutzt werden (manche Optimierungstufen verhindern ab und an das Einsehen des Variablenwertes). In ¨QProgrammer¨ wurde bewusst kein Debugger integriert. Viele Programmierer lassen sich dazu hinreisen ganze Programme mittels ¨Try and error¨ zu implementieren und diese Programme mit Hilfe des Debuggers auf derren korrekte Funktion hin zu trimmen. Dieser Programmierstiel ist aber absolut nicht zu empfehlen. Aus diesem Grund wurde der Debugger in ¨QProgrammer¨ nicht integriert. Der Programmierer soll sich hier angewöhnen erst zu denken und dann zu programmieren.

9 Zusammenfassung

Die in diesem Dokument vorgestellten Aspekte von Object Paskal gehen zum Teil deutlich über die Anforderungen für die Testaufgaben hinaus. Dieses Dokument kann daher auch als Handbuch von ¨QProgrammer¨betrachtet werden. Für die Lösung der Aufgaben ist das Verständnis von z.B. der Rekursion nicht notwendig. Auch kommen viele Aufgaben ohne die Verwendung von Schleifen aus.

10 Beispiele

Es Folgen einige Codebeispiele. Diese müssen nicht unbedingt eine semantische Bedeutung haben. Sie dienen lediglich als ein kleiner Überblick der syntaktischen Darstellung.

```
// Diese Funktion gibt das Ergebnis von Parameter1 + 1 wider
function Erhoeheum1(Parameter1:integer):integer;
begin
  result := Parameter1 + 1;
end;

// Eine Schleife
for i := 0 to n - 1 do begin
  DoSomething
end;

// Ein if then else
if BoolscherAusdruck then begin
  for i := 0 to 2 do begin
    DoSomething
  end;
end else begin
  for i :=2 downto 0 do begin
    DoSomethingOther
  end;
end;

// eine Funktion
function tutviel(x,y:Integer;b:Boolean):integer;
var Zwischenspeicher :integer;
begin
  if b then begin
    Zwischenspeicher := x - y;
    result := x * Zwischenspeicher;
  end else begin
    result := x * x;
  end;
end;
```

11 Schlüsselworte

Die hier gezeigte Liste zeigt die Schlüsselworte welche in ¨QProgrammer¨benutzt werden. Diese sind nur eine Teilmenge der Schlüsselworte in Object Pascal.

and, **begin**, **div**, **do**, **downto**, **else**, **end**, **for**, **function**, **if**, **mod**, **not**, **or**, **then**, **to**, **var**, **while**, **xor**.

12 Literatur verweise

Drachenbuch	Autoren	Alfred V. Aho, Monica S. Lam, Ravi Sethi
	Verlag	Pearson Studium, 1. Auflage, 2008
	ISBN	3827370973

13 Aufgaben

Es folgen einige einfach zu bearbeitende Aufgaben :

- Schreiben sie den Funktionsrumpf für die Funktion Point_in_rect, Diese Funktion soll "True" zurück geben, wenn der Punkt px,py im durch die Punkte x1,x2,y1,y2 aufspannenden Achsenparallelen Rechteck liegt. Andernfalls soll "False" zurück gegeben werden.
  Kleiner Tipp : Machen sie sich eine Skizze.

- Schreiben sie den Funktionsrumpf für die Funktion Euler1. Diese Funktion soll die Frage 1 auf www.ProjectEuler.net beantworten.
  Diese Frage lautet :
  Bilden sie die Summe aller Zahlen < 1000 , welche durch 3 oder 5 Teilbar sind. Und geben sie ihr Ergebniss über den Result Parameter aus.
  Hinweis :
  Wie man sieht benötigt die Funktion keine Übergabeparemter, also müssen alle Variablen selbst deklariert werden.

- Schreiben sie den Funktionsrumpf für die Funktion Fibonacci. Diese Funktion soll als Übergabeparameter den Wert N erhalten und dann Rekursiv den Wert der N-ten Fibonacci Zahl bestimmen und diesen ausgeben.
  Die gesuchte Fibonacci Folge beginne bei $N = 0$.
  Die ersten Glieder seien : 1,1,2,3,5,8 ...

- Schreiben sie den Funktionsrumpf für die Funktion Primzahl. Diese Funktion soll als Übergabeparameter den Wert N erhalten und dann den Wert true zurückliefern wenn n eine Primzahl ist.
  Hinweis:
  Hierbei ist die Mathematische Definition der Primzahlen gefragt, d.h. die 1 ist keine Primzahl.

- Schreiben sie ein Programm welches die nach unten Abgerundete Wurzel der Eingabe Berechnet.
  z.B.:
  Wurzel(16) = 4
  Wurzel(24) = 4
  Wurzel(25) = 5