> T E X T O P I A

Einführung in die Programmierung von Text-Adventures mit
Textopia Version 1.0

```
      ,\=/
 oo___@  !
 {,,,___) \   /\  /\
        (_ \/^ (/^\ (
        (_   \/^\    (
        (_    \ _    (
        (_ \  !! \/\(
       (_( !  !!
      (__( ( )!        \
     /(___(/´´  \      //
 ___( (___(      )____//
;;;__ ;;;_______)_____/
```


Geschrieben von Oliver Berse
eMail: Oliver.Berse@rz.ruhr-uni-bochum.de

Stand: Juni 1999

1 Einleitung

Textopia ist eine Toolbox für die Programmierung klassischer Text-Adventures. Wie andere Systeme auch, die für diese Aufgabe entwickelt wurden, nimmt Textopia Ihnen viel Routinearbeit ab, um die Sie sich kümmern müßten, wenn Sie Ihre Adventures ohne Hilfsmittel in einer Sprache wie C oder Pascal schreiben wollten. Viele Entwicklungssysteme verlangen das Erlernen einer speziellen Programmiersprache und/oder müssen erst an die deutsche Grammatik angepaßt werden. Textopia ermöglicht die Adventureprogrammierung mit Hilfe der vielen Programmierern vertrauten Sprache Pascal und wurde von vornherein für die deutsche Sprache entworfen. Textopia

- arbeitet mit Turbo Pascal unter DOS, mit dem kostenlosen Free Pascal (http://www.tfdec1.fys.kuleuven.ac.be/~michael/fpc/fpc.htm) unter Linux und mit Delphi 4 unter Windows. Mit Textopia geschriebene Spiele müssen daher nur neu kompiliert werden, um sie zwischen DOS, Windows und Linux auszutauschen.

- unterstützt Räume, Türen, Fahrzeuge, Nichtspielercharaktere, Behälter, ein- und ausschaltbare Dinge, Lichtquellen, die Unterscheidung zwischen Licht und Dunkelheit, Zeitschalter und Hintergrundprozesse (Timer & Daemons), die Verwaltung mehrerer Spielstände, Automapping, eine Protokolldatei sowie einen Debugmodus.

- verfügt über einen Parser, der 40 vordefinierte Verben und Kommandos beherrscht und sich einfach erweitern läßt. Der Parser unterstützt die Deklination fast aller deutschen Substantive und erkennt Pluralformen, Adjektive, Reflexivpronomen und natürlich Umlaute.

- ahmt das "Look & Feel" klassischer Text-Adventures der 80er Jahre nach. Die Statuszeile, Bildschirmfarben und Funktionstastenbelegung sind nur einige der Dinge, die Sie in Textopia an Ihre eigenen Vorstellungen anpassen können.

Dieser Text soll Ihnen als Einstieg in Textopia und als Nachschlagewerk bei der Programmierung eigener Adventures dienen. Grundkenntnisse in Pascal sind zu seinem Verständnis sicher von Vorteil. Am besten lesen Sie diesen Text vor Ihrem Computer, um die Beispielprogramme besser nachvollziehen zu können.

Programmiert, getestet und dokumentiert wurde Textopia von Oliver Berse, dem Schreiber dieser Zeilen. Textopia darf selbstverständlich von jedem interessierten Programmierer benutzt und weiterverbreitet werden. Das gilt auch für damit entwickelte Programme. Beachten Sie aber bitte folgende Punkte: (a) Wenn Sie Textopia weiterverbreiten, dann nur mit allen zugehörigen Dateien, auch den Quelltexten. Eigene Programme, die Textopia verwenden, müssen Sie aber nicht im Quelltext weitergeben. (b) Wenn Sie Änderungen (es sollten Verbesserungen sein) an Textopia vornehmen, dürfen Sie natürlich auch Ihren Namen in den Kommentarkopf der Quelltexte setzen. Meinen lassen Sie da bitte stehen. (c) Eine kommerzielle Verwendung, wenn sie denn überhaupt möglich ist, bedarf meiner Zustimmung.

Textopia findet sich im IF-Archiv (ftp.gmd.de) und auf meiner Homepage unter http://homepage.ruhr-uni-bochum.de/Oliver.Berse/comp1.htm. Dort finden Sie immer die aktuellsten Informationen über Textopia. Berichte über Bugs, Kommentare und Anregungen senden Sie bitte an Oliver.Berse@rz-ruhr-uni-bochum.de.

2 Kompilieren der Units

2.1 Der Quelltext

Die Quelldateien für die drei Textopia Units sind tstring.pas, tio.pas und tmain.pas. Diese Dateien sind für jedes Textopia-Programm notwendig und müssen zunächst kompiliert werden.

Textopia wurde mit Turbo Pascal 6.0 unter DOS, mit Delphi 4 unter Windows 95/98 und mit Free Pascal unter Linux programmiert und getestet. Mit Delphi 3 und Windows NT dürfte es auch funktionieren, getestet habe ich das aber nicht. Der Free Pascal Compiler (FPC) unterstützt neben Linux auch OS/2 und das AmigaOS. Mangels entsprechender Soft- und Hardware konnte ich Textopia unter diesen Systemen noch nicht testen. Eine Anpassung der Units an andere TP-kompatible Compiler, wie z.B. das kommerzielle Virtual Pascal, sollte keine unlösbaren Probleme verursachen.

Delphi 4 und FPC sind nicht wirklich 100% kompatibel zu Turbo Pascal. Das liegt an der 32-Bit Struktur beider Compiler und der frühen Versionsnummer von FPC, das noch ein paar Kinderkrankheiten hat. Die Quelltexte der Textopia-Units werden aber von allen drei Compilern problemlos geschluckt. Möglich wird dies durch die Compilerdirektive *{$DEFINE xxx}* am Anfang der Quelltexte. Hier können die Symbole tp, fpc oder delphi definiert werden. An den wenigen kritischen Stellen teilt ein Konstrukt wie

```
{$IFDEF tp}
   ..code..
{$ELSE}
   ..code..
{$ENDIF}
```

dem entsprechenden Compiler dann mit, ob er den Code für DOS, Linux oder Windows übersetzen soll.

2.2 Kompilieren mit Turbo Pascal unter DOS

Lassen Sie den Compiler mit Build aus der Datei tmain.pas eine Unit-Datei erstellen, die anderen drei Quelldateien werden automatisch mitkompiliert. Von den am Dateianfang gesetzten Compileroptionen sollten die I/O-Kontrolle *{$I}* und die strenge Stringkontrolle *{$V}* immer ausgeschaltet, die erweiterte Syntax *{$X}* immer eingeschaltet bleiben.

2.3 Kompilieren mit Delphi unter Windows

Sie können die Units sowohl in der Entwicklungsumgebung (IDE) als auch über die Kommandozeile in einem DOS-Fenster kompilieren. Wenn Sie in der IDE kompilieren, öffnen Sie tmain.pas als Pascal-Projekt, um die benötigten Dateien mitzuübersetzen. Mit der Kommandozeilenversion von Delphi starten Sie die Übersetzung durch Eingabe von

```
dcc32 -cc -b tmain.pas
```

Die Hinweise und Warnungen, die Delphi beim kompilieren ausgibt, sind nur falscher Alarm und können unberücksichtigt bleiben.

Neben den Dateien tmain.pas, tio.pas und tstring.pas übersetzt Delphi auch die Datei dcrt.pas. Diese Unit ersetzt die unter Delphi 4 fehlende Crt-Unit von Turbo Pascal, ohne die Textopia nicht auskommt. DCrt ist jedoch kein vollständiger Ersatz für alle Funktionen der originalen Crt-Unit. Sie simuliert nur die von Textopia verwendeten Routinen und arbeitet in DOS-Fenstern noch recht langsam (zumindest auf meinem 200Mhz Pentium).

Wenn Sie Spiele unter Verwendung der Textopia-Units mit Delphi kompilieren, achten Sie darauf, daß sie als Anwendungen für den Textmodus kompiliert werden. Sie können am Anfang des Quelltextes ein *{$AppType Console}* einfügen, in der IDE unter Projekt/Optionen/Linker den Textmodus einstellen oder den Kommandozeilencompiler mit der Option -cc aufrufen.

2.4 Kompilieren mit Free Pascal unter Linux

Geben Sie im Verzeichnis mit den Quelldateien einfach ppc386 -So -B tmain.pas ein, um die Übersetzung zu starten. Mit dem Schalter -So verhält FPC sich bei der Übersetzung kompatibel zu TP6/7, -B entspricht dem Build-Befehl des TP-Compilers. FPC unterstützt mehr Compileroptionen als TP, so können Sie auch für den Prozessortyp Ihres Computers optimierten Code erzeugen. Wenn Sie nur ppc386 eingeben, erhalten Sie eine Liste aller möglichen Optionen.

Während der Übersetzung gibt FPC einige Hinweise aus, die Sie allesamt ignorieren können. Meldungen über nicht verwendete Parameter erzeugt FPC fälschlicherweise, wenn eine Prozedur einen Parameter entgegennimmt und ihn ohne Änderungen an eine zweite Prozedur weiterreicht.

2.5 Worauf Sie unter Linux achten sollten

Textopia-Programme sind Textanwendungen und funktionieren im Linux-Textmodus oder unter einem Windowmanager in einem Terminalfenster. Letztere kennen meist verschiedene Textmodi, die sich nicht alle mit der Crt-Unit von FPC vertragen. Wenn sich Ihr Terminalfenster im VT100-Modus befindet, wird die von Textopia benutzte Crt-Unit früher oder später einen Segmentation Fault verursachen. Im Console-Modus mit 80x25 oder mehr Zeichen funktioniert Textopia dagegen einwandfrei.

Unter dem KDE-Desktop können Sie das K-Terminal oder das in der SuSE Distribution enthaltene Programm Konsole von Lars Doelle verwenden. Bei K-Terminal wählen Sie unter Einstellungen eine Fenstergröße von mindestens 80x52 Zeichen und schalten unter Einstellungen/Terminal den Console-Modus und die Backspace-Taste ein. Im Programm Konsole stellen Sie den 80x25 Zeichen-Modus ein und schalten die Option BS sendet DEL aus.

Wenn Sie beim Programmieren unter einem Windowmanager in einem Editorfenster (XEmacs oder was Ihnen sonst so gefällt) den Quelltext bearbeiten und in einem Terminalfenster kompilieren und testen, erhalten Sie eine ähnlich einfach zu bedienende Entwicklungsumgebung wie unter Delphi oder Turbo.

3 Textopia und OOP

3.1 Einige wichtige Begriffe

Textopia nutzt die Möglichkeiten der objektorientierten Programmierung (OOP) mit Turbo Pascal. Um mit Textopia Adventures zu programmieren, müssen Sie kein Experte in Object Pascal, C++ oder gar Smalltalk sein. Einige Begriffe der OOP sollten Sie aber kennen. Daher habe ich hier, natürlich ohne Anspruch auf Vollständigkeit, einige formale Definitionen aufgeführt. Eine sehr viel genauere "Einführung in objektorientierte Programmierung mit Turbo Pascal" stammt von Joseph Mittendorfer und sei jedem Interessierten zur Lektüre empfohlen.

- **Objekt** - bildet eine Einheit aus Daten (Variablen) und den sie verarbeitenden Prozeduren und Funktionen, die auch die Methoden des Objekts genannt werden. In Pascal ähnelt ein Objekt einem Record, der auch Prozeduren und Funktionen aufnehmen kann. Ziel der OOP ist es, große Softwareprojekte in mehrere kleine, überschaubare und wiederverwertbare Objekte zu unterteilen.

- **Klasse** - die Definition der Daten und Methoden eines Objekts. In Turbo Pascal werden Klassen Objekttypen genannt und ähnlich wie Records definiert. Mit der Anweisung *TObjekt=OBJECT* erzeugen Sie den neuen Objekttyp TObjekt. Wie im Interface-Teil einer Unit, werden in Objektdefinitionen nur die Methodennamen und ihre Parameter aufgeführt, die vollständige Methode wird dann außerhalb der Objektdefinition geschrieben.

- **Instanz** - Objekt eines Objekttyps. Instanzen und Objekte sind in Pascal Variablen eines Objekttyps. Die Deklaration objekt:TObjekt erzeugt eine Instanz des Objekttyps TObjekt. Wenn Sie einen Zeiger auf einen Objekttyp definieren (pobjekt:=^TObjekt), können Sie Instanzen auch dynamisch mit *NEW()* erzeugen.

- **Vererbung** - Eine neue Objektdefinition kann auf einem bereits bestehenden Objekttyp aufbauen. Bei der Objektdefinition wird dazu der Vorfahre in Klammern hinter das Schlüsselwort *OBJECT* geschrieben (*TNeu=OBJECT(TObjekt)*). Der neue Objekttyp erbt dabei alle Variablen und Methoden des Vorfahren und kann diesen neue hinzufügen. Durch Vererbung kann ein Stammbaum voneinander abgeleiteter Objekttypen (die sog. Objekthierarchie) gebildet werden.

- **Polymorphismus** - von einem Vorfahren geerbte Methoden können in einem neuen Objekttyp überschrieben, d.h. neu definiert werden. Der Name und die Parameterzahl der betreffenden Prozedur oder Funktion bleiben dabei gleich. Verschiedene Aufgaben können so durch Aufruf der gleichen Methode erledigt werden.

- **virtuelle Methoden** - ermöglichen dem Compiler zur Laufzeit festzustellen, welche der überschriebenen Versionen dieser Methode bei ihrem Aufruf abgearbeitet wird. Virtuelle Methoden werden mit dem nachgestellten Schlüsselwort *VIRTUAL* gekennzeichnet.

- **Konstruktoren** - sind Methoden, die die Daten von Instanzen bei ihrer ersten Verwendung initialisieren, d.h. mit bestimmten Werten vorbelegen. In jedem Objekttyp sollte ein Konstruktor definiert werden. In Turbo Pascal erhalten Kontruktoren gewöhnlich den Namen

Init. Bei der Erzeugung dynamischer Instanzen kann Init schon bei der Erzeugung mit *NEW()* aufgerufen werden: *NEW(pobjekt,Init).*

- **Destruktoren** - sind Methoden, die dynamische Daten von nicht mehr benutzten Objekten löschen und somit den Speicher aufräumen. In jedem Objekttyp sollte ein Destruktor definiert werden. In Turbo Pascal erhalten Destruktoren gewöhnlich den Namen Done.

Lassen Sie sich von diesen Definitionen nicht abschrecken. Sie werden schnell herausbekommen, wie Sie Textopia und OOP für eigene Adventures nutzen können.

3.2 Adventures und Objekte

Die Spielwelt fast aller Adventures besteht aus Elementen wie Räumen, simplen Goldmünzen, komplizierten Maschinen und NPCs, die sprechen und sich bewegen können. Neben der Programmierung eines Parsers stellt die Verwaltung dieser Spielwelt das größte Problem bei der Adventureprogrammierung dar. Textopia löst dieses Problem, indem es alle in einem Spiel vorkommenden Dinge als Objekte darstellt. In tmain.pas werden der Objekttyp *TBasic* und seine drei Nachfolger *TRoom*, *TLink* und *TItem* definiert. Diese Objekttypen stellen die Grundbausteine jedes Textopia-Programms dar:

- **TBasic** - verwaltet die Informationen, die alle zur Spielwelt gehörenden Objekte gemeinsam haben, u.a. einen Namen.

- **TRoom** - stellt einen einzelnen Ort in der Spielwelt dar. Dabei kann es sich um einen Raum in einem Gebäude oder um einen Ort unter freiem Himmel handeln.

- **TLink** - verbindet jeweils zwei Räume und kann in Verbindung mit einem Objekt vom Typ *TLock* verschiedene Durchgänge und Türen darstellen.

- **TItem** - besitzt eine Reihe von Attributen, die sich ein- und ausschalten lassen, um die unterschiedlichsten Dinge in einem Adventure zu simulieren.

Objekte der Typen *TRoom*, *TLink* und *TItem* werden im folgenden Text auch Räume, Verbindungen und Gegenstände genannt.

Aber nicht nur die Spielwelt wird durch Objekte dargestellt. Auch der Spieler (genauer: sein Alter-Ego im Spiel) und das Spiel selbst sind Objekte. Spieler und Spiel stellen jeweils ein Objekt der Typen *TPlayer* und *TGame* dar:

- **TPlayer** - verwaltet verschiedene Informationen über den Zustand des Spielers, z.B. seine Position in der Spielwelt und seine Punktzahl.

- **TGame** - enthält den Parser und verschiedene Methoden zur Kontrolle des Spielverlaufs.

Die genannten Objekttypen besitzen virtuelle Methoden, die Sie in abgeleiteten Typen überschreiben können, um ihr Verhalten in einem Spiel an Ihre Vorstellungen anzupassen.

Textopia ist weitgehend objektorientiert, besitzt aber auch viele Prozeduren und Funktionen, die nicht zu einem Objekttyp gehören. Wenn Sie den Quelltext durchstöbern, werden Sie zu-dem feststellen, daß sich einige Sachen noch eine Spur objektorientierter hätten umsetzen lassen. Textopia entwickelte sich aus Experimenten mit einem deutschsprachigen Parser. Daher beansprucht es nicht, ein Lehrbuchbeispiel für OOP zu sein.

4 Das erste Programm

In den folgenden Kapiteln soll schrittweise ein erstes Textopia-Programm entwickelt werden, das ein Mini-Adventure darstellt. Textopia-Programme sind Pascal-Programme und folgen natürlich deren bekanntem Schema:

```
PROGRAM programmname;
USES    { Einbinden von Units }
CONST   { Konstantendeklaration }
TYPE    { Typendeklaration }
VAR     { Variablendeklaration }
...     { Funktionen und Prozeduren }
BEGIN   { Hauptprogramm }
END.
```

4.1 Units, Konstanten, Typen und Variablen

In jedes Textopia-Programm werden die drei Textopia-Units eingebunden:

```
PROGRAM demo1;
USES TString,TIO,TMain;
```

Für jeden Raum, jede Verbindung und für jeden Gegenstand im Programm wird nun eine numerische Konstante benötigt, über die später jedes Objekt eindeutig zu identifizieren ist. Die Konstanten dürfen (eindeutige) Werte zwischen 1 und 65535 annehmen und müssen in keiner Weise geordnet sein. Eine solche Konstante soll im weiteren Text die ID eines Objekts der Spielwelt genannt werden. Für das denkbar einfachste Textopia-Programm, die Simulation eines einzigen leeren Raums (hier ein Wald), können wir also deklarieren:

```
CONST
  forest_id = 10;
```

In der Typendeklaration können neue Objekttypen definiert werden, um einzelne Methoden neu zu definieren. Mit Ausnahme einiger interner Typen können Sie von allen Objekttypen in Textopia Nachfolger ableiten. Für unser Beispiel können wir aber bei TRoom bleiben und die Typendeklaration überspringen.

Alle Objekte werden in Textopia dynamisch angelegt. Im Variablen-Abschnitt müssen wir daher Zeiger auf die benötigten Objekttypen deklarieren, um von ihnen Instanzen erzeugen zu können. Wir können hierbei auf die vordefinierten Zeigertypen *PBasic*, *PRoom*, *PLink*, *PItem*, *PPlayer* und *PGame* zurückgreifen, die einen Zeiger auf die entsprechenden Objekttypen darstellen. Es müssen immer auch zwei Zeiger für das Spieler- und das Spielobjekt (oder für ihre Nachfolger) deklariert werden. Zur Darstellung eines Raums benötigen wir also drei Zeiger:

```
VAR
  room   : PRoom;
  player : PPlayer;
  game   : PGame;
```

Die im Var-Abschnitt deklarierten Zeiger vom Typ *PRoom*, *PLink*, und *PItem* dienen der Erzeugung aller im Spiel vorkommenden Objekte des entsprechenden Typs. Es wird also nicht für jedes Objekt ein eigener Zeiger benötigt. Wenn Sie später im Spiel einen Zeiger auf ein bestimmtes Objekt benötigen, verwenden Sie die Funktion

```
Adr(_id : WORD) : POINTER
```

Mit einer expliziten Typumwandlung der Form *PRoom(Adr(forest_id))* können Sie diesen Pointer in einen Zeiger auf einen Nachfolger von *TBasic* umwandeln. Verwechseln Sie diese Funktion nicht mit der Pascal-Funktion *Addr*.

Hätten wir Nachfolger von Textopia-Objekten deklariert, müßten wir nun ihre Methoden schreiben. So aber können wir gleich mit dem Hauptprogramm loslegen.

4.2 Vorbereitungen mit Prepare

Auf das *BEGIN* des Hauptprogramms muß in Textopia immer zuerst die Anweisung

```
Prepare(_filename: STRING; _new: BOOLEAN)
```

folgen. *Prepare* erfüllt folgende Aufgaben:

Zunächst legt es den Dateinamen für die Dateien fest, die im Spielverlauf angelegt werden. Das sind *.sav und *.ttx für die Spielstandverwaltung und *.txt für eine Textdatei, die automatisch das Spielgeschehen protokolliert. Eine Datei *.dat wird nur in der DOS Version angelegt. Sie nimmt die zu den Objekten gehörenden Texte in codierter Form auf. Diese Datei wird nur angelegt, um unter DOS Speicher zu sparen. Unter Windows und Linux bleiben alle Texte im Speicher. Der Parameter *_new* bestimmt, ob die Datei *.dat bei jedem Programmstart neu angelegt wird, was während der Programmierung sehr zu empfehlen ist, oder eine bereits bestehende Datei verwendet wird. Unter Linux und Windows hat dieser Parameter also keine Bedeutung. Wenn Sie verhindern wollen, daß Spieler die den Objekten zugeordneten Texte (und nur diese) im Programmcode per HexEditor verändern können, setzen Sie *_new* im fertigen Programm auf *False* (ein hundertprozentiger Schutz ist das aber auch nicht).

Ein Hinweis: Die Werte *True* und *False* werden in Textopia häufig verwendet und können mit den bereits deklarierten Konstanten t und f abgekürzt werden.

Als nächstes initialisiert *Prepare* eine interne Liste, die alle von Ihnen definierten Objekte aufnimmt. Dann erzeugt *Prepare* eine Instanz von *PRroom* mit der ID 0 und trägt sie als erstes Element in die Liste ein. Dieser Raum 0 dient als einfache Alternative für die Definition und Beseitigung von von Objekten während des Spielverlaufs, die in Textopia nicht vorgesehen ist. Ißt der Spieler beispielsweise einen Keks, wird dieser nicht mehr benötigt und landet im Raum 0. Umgekehrt können Sie aus diesem Raum auch bislang verborgene Objekte hervorzaubern. Daher können wir diesen Raum auch Top (Zylinder eines Zauberkünstlers) nennen.

4.3 Objekte und ihre Namen: Ein erster Raum

Nun aber weiter mit dem Hauptprogramm. Unsere Spielwelt soll zunächst aus nur einem Raum bestehen. Ein Objekt vom Typ *TRoom* mit der ID room_id erzeugen wir wie folgt:

```
NEW(room,Init(forest_id,'+W%ald#er'));
```

Mit *NEW* wird eine neue Instanz erzeugt und gleichzeitig der für die Initialisierung ihrer Daten zuständige Konstruktor Init ausgeführt. Die ersten beiden Parameter der Konstruktoren von *TRoom*, *TLink* und *TItem* sind die ID und der Name der neuen Instanz. Über die ID identifizieren Sie als Programmierer eine Instanz und über ihren Namen bezieht der Spieler sich auf ein Objekt in der Spielwelt. Die Sonderzeichen im Namen zeigen Textopia an, wie der Name zu deklinieren ist und welcher Artikel ihm vorgestellt werden muß, wenn er in einer Ausgabe erwähnt wird. Textopia kennt für diesen Zweck folgende Zeichen:

- `-` Markiert ein feminines Nomen: -Flasche → die Flasche.
- `+` Markiert ein maskulines Nomen: +Troll → der Troll.
- `$` Markiert einen Eigennamen: $Tom → Tom
- `%` Markiert vor einem Vokal eine Lautveränderung im Plural: %Apfel → Äpfel.
- `#` Markiert den Buchstaben oder die Silbe, die im Plural angehängt wird: Flasche#n → die Flaschen
- `*` Markiert ein vorgestelltes Adjektiv. Vor einem Namen darf höchstens ein Adjektiv stehen. Beachten Sie, daß Minus-, Plus- und Dollarzeichen vor dem Adjektiv stehen müssen: +rot* Drache#n → der rote Drache.

Steht vor dem Namen kein Minus-, Plus- oder Dollarzeichen, so gilt der Name als Neutrum: Fenster → das Fenster.

Sie können einem Objekt auch mehrere synonyme Namen geben, diese müssen Sie mit einem Semikolon trennen: Raumschiff#e;Raumfahrzeug#e;Sternenschiff#e. Zwischen den Namen dürfen keine Leerzeichen stehen. Hat ein Objekt mehrere Namen, so erwähnt Textopia immer nur den ersten, während der Spieler alle Namen benutzen kann, um sich auf das Objekt zu beziehen. Schließlich können Sie den Namen auch ganz weglassen, was bei Verbindungen oft sinnvoll ist.

Der Objektname wird von einem Objekt vom Typ *TMemText* gespeichert. Wenn p ein Zeiger auf einen Nachfolger von *TBasic* ist, können Sie mit der Konstruktion

```
p^.name.GetText
```

auf den Objektnamen zugreifen.

Für die Deklination von Objektnamen ist die in tstring.pas definierte Stringfunktion *Noun*() zuständig. Ihr werden folgende Parameter übergeben:

- **n1:String** - ein Objektname mit seinen Sonderzeichen.

- **casus:TCasus** - einer der vier grammatischen Fälle, für den der Name dekliniert werden soll. Mögliche Werte sind: *nom* (Nominativ), *acc* (Akkusativ), *gen* (Genitiv) und *dat* (Dativ).

- **def:Boolean** - bestimmt, ob der Objektname mit einem bestimmten (*True*) oder unbestimmten (*False*) Artikel zurückgegeben wird.

- **num:Byte** - bestimmt, ob der Objektname im Singular (num=1) oder im Plural (num>1) zurückgegeben wird.

Manchmal ist ein Blick in den Duden hilfreich, um mit *Noun()* das gewünschte Ergebnis zu erzielen.

Wenn ein Objektname nur im Nominativ Singular ohne seine Sonderzeichen benötigt wird, kann auch die ebenfalls in tstring.pas definierte Funktion

```
ShortName(str : STRING; mode : BYTE) : STRING
```

verwendet werden. Sie nimmt den Objektnamen mit Sonderzeichen und einen Byte-Parameter entgegen und gibt den Namen ohne Sonderzeichen zurück. Ist der Byte-Parameter gleich 0, wird der Objektname wie bei der Initialisierung angegeben zurückgegeben, bei 1 und 2 wird er in Klein- bzw. Großbuchstaben umgewandelt.

4.4 Statische Objektbeschreibungen und Sonderzeichen

Wir haben jetzt einen Raum erzeugt und ihm eine ID und einen Namen gegeben. Das reicht natürlich noch nicht. Wir brauchen einen Text, der dem Spieler mitteilt, was in dem Raum los ist, wenn er sich in ihm aufhält. Jedem Objekt eines Nachfolgers von *PBasic* können wir mit der Methode

```
AddText(str : STRING)
```

eine ausführliche Beschreibung zuordnen. Diese Methode erreichen wir über den Zeiger, mit dem wir das Objekt auch erzeugt haben:

```
WITH room^ DO
BEGIN
  AddText('Fast undurchdringliches Unterholz läßt Sie nur mühsam '+
          'weiterkommen. Weiter westlich scheinen die Bäume weniger '+
          'dicht beieinander zu stehen.\n');
END;
```

Der an *AddText()* übergebene Text darf maximal 255 Zeichen lang sein. Sie dürfen aber beliebig viele *AddText()* hintereinander schreiben, womit auch die Länge des Textes nur von Speicher begrenzt wird. Unter DOS werden die mit *AddText()* hinzugefügten Texte in der Datei *.dat gespeichert.

Bei der Textausgabe kümmert sich Textopia selbständig um den Zeilenumbruch. Wird ein zusammengehöriger Text länger als eine Bildschirmseite, wartet Textopia nach der Ausgabe einer Seite automatisch auf einen Tastendruck des Spielers, um den Rest anzuzeigen.

Textopia kennt eine Reihe von Steuerzeichen, die Sie im Text einfügen können, um die Textausgabe zu beeinflussen. Eingeleitet werden diese Anweisungen mit dem Zeichen \ (Backlash):

\n Erzwingt einen Zeilenumbruch
\h Der folgende Text wird mit erhöhter Intensität ausgegeben. Dies entspricht der TP-Anweisung *HighVideo*.
\l Der folgende Text wird mit normaler Intensität ausgegeben. Dies entspricht der TP-Anweisung *NormVideo*.
\r Schaltet die inverse Textdarstellung ein und aus.
\b Gibt das folgende Zeichen in Großschrift aus.

Einige weitere Steuerzeichen sind für die systemunabhängige Darstellung deutscher Umlaute vorgesehen (s. 8.8).

Mit *AddText()* angegebene Objektbeschreibungen sind statisch. Das heißt, sie lassen sich während des Spiels nicht mehr verändern. Wenn aber z.B. während des Spiels ein Teil der Decke eines Raums einstürzt und eine Tür versperrt, muß natürlich auch die entsprechende Raumbeschreibung geändert werden. Um eine Objektbeschreibung an bestimmte Spielsituationen anpassen zu können, muß ein neuer Objekttyp von einem der Nachfolger von *TBasic* abgeleitet werden und die Methode *TBasic.Mytext* neu definiert werden. Wird während des Spiels eine Objektbeschreibung ausgegeben, prüft Textopia immer zuerst, ob dem Objekt ein Text mit *AddText()* hinzugefügt wurde. Ist das nicht der Fall, wird die Methode *MyText*, die *TRoom*, *TLink* und T*I*tem von *TBasic* erben, aufgerufen. *MyText* tut gewöhnlich nichts und ist nur definiert, um in einem Enkel von *TBasic* überschrieben zu werden. In *MyText* können nun bedingte Objektbeschreibungen erfolgen. Für die Textausgabe muß nicht nur in *MyText* immer die Prozedur

```
Print(line : STRING)
```

statt *Writeln* verwendet werden, da sie sich auch um die Bildschirmverwaltung kümmert. In unserem Beispiel bleiben wir aber erstmal bei einer statischen Beschreibung.

4.5 Spieler und Spiel

Nachdem die Objekte der Spielwelt, hier nur ein Wald, erschaffen sind, müssen wir uns um einige Eigenschaften des Spielers und des eigentlichen Spiels kümmern. Zunächst erzeugen wir eine Instanz von *TPlayer* (oder eines Nachfolgers davon). Um sie zu initialisieren, müssen wir den entsprechenden Konstruktor mit fünf Parametern aufrufen:

```
Init(_where: WORD; _adress: TAdress; _gender: TGender; _wmin,_wmax: BYTE)
```

- **_where** - ist die ID eines Raums oder eines Items, das dem Spieler als Behälter dient (wir kommen darauf zurück), und bestimmt, wo das Spiel beginnt.

- **_adress** - kann den Wert du, sie oder ihr annehmen und bestimmt die von Textopia benutzte Anrede für den Spieler. In den von Ihnen vorgegeben Texten müssen Sie sich an die gewählte Anrede halten. Entsprechende Personalpronomen und Verbformen können noch nicht automatisch erzeugt werden.

- **_gender** - bestimmt das Geschlecht von Spielers Alter-Ego und kann während des Spiels mit der *PPlayer*-Methode *ChangeGender* gewechselt werden. Mögliche Werte sind natürlich male und female. Dies ist noch eine experimentelle Option, die von Textopia nur bei einer einzigen Ausgabe berücksichtigt wird. Wenn Sie aber ein Spiel im Stil von Infocoms "Leather Goddesses of Phobos" schreiben wollen, ist _gender sicher nützlich.

- **_wmin** - gibt das Normalgewicht von Spielers Alter-Ego an, wenn er nichts bei sich trägt.

- **_wmax** - bestimmt das Maximalgewicht des Spielers. Da sich auch allen möglichen Gegenständen ein Gewicht zuordnen läßt, können Sie mit _wmax leicht bestimmen, wieviele Dinge der Spieler mit sich herumtragen darf.

Einen Abenteurer können wir also wie folgt in unserem Wald aussetzen:

```
NEW(player,Init(forest_id,sie,male,1,3));
```

Im letzten Schritt initialisieren wir jetzt das eigentliche Spiel. Der Konstruktor von *TGame* verlangt folgende Parameter:

- **_player:PPlayer** - ein Zeiger auf die erzeugte Instanz von *TPlayer*.

- **_statusline:Boolean** - bestimmt, ob zu Spielbeginn eine Statuszeile in der ersten Bildschirmzeile angezeigt wird. Die Statuszeile kann auch während des Spiels ein- und ausgeschaltet werden.

- **_upsize:Boolean** - bestimmt, ob die Spielereingaben in Großbuchstaben erscheinen sollen, um sie optisch von den Spieltexten zu unterscheiden. Textopia unterscheidet aber nicht zwischen Groß- und Kleinschreibung. Intern werden alle Eingaben in Kleinschrift umgewandelt.

- **_verbose:Boolean** - ist _verbose wahr, so gibt Textopia in jedem Raum, den der Spieler betritt, die vollständige Raumbeschreibung aus, auch wenn der Spieler den Raum bereits besucht hat. Andernfalls werden in bereits bekannten Räumen nur deren Namen ausgegeben.

- **_history:Byte** - der Zeileneditor von Textopia erzeugt automatisch eine Liste der letzten n Spielereingaben, die der Spieler mit den Tasten CursorOben/CursorUnten editieren kann. *_history* bestimmt die Anzahl der darin gespeicherten Eingabezeilen.

Unsere Waldszene können wir jetzt mit

```
NEW(game,Init(t,f,t,10));
```

initialisieren. Bevor wir das Programm starten, können wir mit der Methode

```
AddProlog(_str : STRING)
```

noch Informationen wie den Titel, eine Copyrightmeldung und eine Einleitung oder Vorgeschichte hinzufügen:

```
WITH game^ DO
BEGIN
  AddProlog('Das Zwergengrab\nEine Textopia-Demo von Oliver Berse\n\n'+
            'Drei Tage schon durchstreifen Sie abseits der Straßen den '+
```

```
            'alten Wald. Hier irgendwo liegen die mit Schätzen gefüllten '+
            'Gräber der Zwergenkönige, die einst diese Länder '+
            'beherrschten.\n\n');
```

Der mit *AddProlog()* dem Spiel hinzugefügte Text erscheint immer zu Spielbeginn. Diese Methode verhält sich ganz ähnlich wie *AddText().* Mit der Methode *TGame.Run* können wir unsere kleine Demonstration jetzt starten. Dabei übernimmt Textopia die Kontrolle über das Spiel, bis der Spieler es beendet. Nach Beendigung des Spiels sollte noch der Destruktor von *TGame* aufgerufen werden, um belegten Speicherplatz aufzuräumen:

```
  Run;
  Done;
END;
```

Vergessen Sie nicht, das Hauptprogramm mit *END.* abzuschließen.

4.6 Ein erster Dialog

Wenn sich demo1.pas problemlos kompilieren und starten läßt, sehen Sie jetzt die Statuszeile mit dem Namen des aktuellen Raums sowie den Punkten und Zügen des Spielers. In der unteren Bildschirmhälfte erscheint zuerst der mit *AddProlog()* zugewiesene Text, die aktuelle Raumbeschreibung und das Zeichen > als Eingabeaufforderung.

Haben Sie bei der Initialisierung der Objekte irgendeinen Fehler gemacht, gibt Textopia eine Warnung (s. 8.9) aus und fragt Sie, ob Sie abbrechen oder fortfahren wollen (in einigen besonders harten Fällen wird das Programm auch ohne Rückfrage beendet). In diesem Fall sollten Sie immer abbrechen und die Fehlerursache beseitigen. Setzen Sie das Programm nur fort, wenn Sie genau wissen, daß der gemeldete Fehler keine Katastrophe verursachen kann.

Das kleine d vor der Eingabeaufforderung weist darauf hin, daß der Debugmodus aktiv ist. In diesem Modus können Sie einige Debugkommandos eingeben, die Ihnen helfen, das Verhalten Ihrer Objekte zu kontrollieren. Die Debugkommandos werden in 7.6 erläutert. Wenn Ihr Spiel fertig ist und fehlerfrei läuft, sollten Sie den Debugmodus mit der Zeile

```
debug:=f;
```

im Hauptprogramm ausschalten.

Viel läßt sich mit nur einem Raum noch nicht anfangen. Ein erster Dialog könnte so aussehen:

Das Zwergengrab
Eine Textopia-Demo von Oliver Berse

Drei Tage schon durchstreifen Sie abseits der Straßen den alten Wald. Hier
irgendwo liegen die mit Schätzen gefüllten Gräber der Zwergenkönige, die
einst diese Länder beherrschten.

WALD
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter
westlich scheinen die Bäume weniger dicht beieinander zu stehen.

d>w
Hier geht es nicht weiter.

```
d>#i
Sie tragen nichts mit sich
0 Item(s)  Gewicht=1

d>warte
Die Zeit vergeht...

d>#ende
Wollen Sie das Spiel wirklich beenden?  (j/n) j
```

Die Mit dem Zeichen # beginnenden Kommandos sind sog. Meta-Verben, die einige Spieloptionen auslösen, das Spielgeschehen selbst aber nicht beeinflussen. Metaverben werden im Abschnitt 7.5 behandelt. Anzahl und Gewicht der Items im Inventar werden nur im Debugmodus angezeigt.

5 Die Spielwelt

5.1 Gegenstände und Attribute

Bevor wir unserem Programm "Zwergengrab" weitere Räume hinzufügen, geben wir dem Spieler einige nützliche Dinge mit auf den Weg. Mit Ausnahme von Räumen und Türen können mit dem Objekttyp *TItem* fast alle in einem typischen Adventure vorkommenden Objekte dargestellt werden.

Um einen Gegenstand zu erzeugen, brauchen wir zuerst wieder eine Konstante für die ID und einen Zeiger vom Typ *PItem*. Wenn wir dem Spieler ein Schwert und einen Rucksack mitgeben wollen, können wir unser Programm um folgende Zeilen erweitern (ich führe hier nur die neuen Zeilen auf, die demo1.pas an den entsprechenden Stellen hinzugefügt werden):

```
CONST
  sword_id    = 11;
  rucksack_id = 12;

VAR
  item : PItem;
```

Der Konstruktor *TItem.Init()* verlangt zunächst wieder eine ID und einen Namen für das neue Objekt (s. 4.3). Danach müssen folgende vier Parameter angegeben werden:

- **_location: PBasic** - einen Zeiger auf den Ort, an dem sich das Objekt zu Spielbeginn befinden soll. Dieser Ort kann ein Raum, das Inventar des Spielers oder ein anderer Gegenstand sein. Den benötigten Zeiger liefert die Funktion *Adr().* Wenn das Objekt im Inventar des Spielers plaziert werden soll, muß dem Konstruktor der Wert *NIL* übergeben werden. Während des Spiels können Sie die Position eines Objekt vom Typ *PItem* mit der Methode *MoveItemTo()* verändern. Die neue Position wird durch einen Zeiger vom Typ *PBasic* übergeben, der die gleiche Bedeutung wie *_location* hat.

- **_show: BOOLEAN** - bestimmt, ob das Objekt automatisch in der Raumbeschreibung aufgeführt wird. Nach Ausgabe des von Ihnen geschrieben Textes werden bei einer Raumbeschreibung alle im Raum befindlichen Objekte aufgelistet. Wenn Sie das Objekt bereits in Ihrem Text erwähnt haben, können Sie hier *False* übergeben. Diese Voreinstellung läßt sich im Spiel mit der Methode *ListMe()* verändern, der *True* oder *False* übergeben wird. So können Sie Objekte bis zu einem bestimmten Ereignis vor dem Spieler verstecken.

- **_wmin: BYTE** - bestimmt das Eigengewicht des Objekts. Wenn Sie das Objekt im Inventar des Spielers plazieren, achten Sie darauf, daß sein Gewicht nicht das Maximalgewicht des Spielers übersteigt.

- **_wmax: BYTE** - bestimmt das Maximalgewicht des Objekts. Wenn das Objekt andere Objekte aufnehmen kann, läßt sich hiermit bestimmen, wieviele Gegenstände es maximal enthalten darf. Soll das Objekt nicht als Behälter dienen, können Sie hier 0 angeben.

Folgende Zeile erzeugt ein Schwert im Inventar des Spielers:

```
NEW(item,Init(sword_id,'Schwert#er',NIL,t,1,0));
```

Damit wird dem Spiel zwar ein neues Objekt hinzugefügt, aber woher soll das Spiel wissen, was der Spieler damit anfangen kann? Um Textopia mitzuteilen, daß der Spieler den neuen Gegenstand z.B. aufnehmen und weglegen kann, müssen dem Objekt sog. Attribute hinzugefügt werden. Ein Attribut ermöglicht es dem Spieler, mit dem Objekt eine bestimmte Aktionen durchzuführen und ist entweder vorhanden oder nicht vorhanden. Nach seiner Initialisierung besitzt ein Objekt keine Attribute. Mit der Methode

```
TItem.SetAttrib(a : BYTE; _on : BOOLEAN)
```

können Sie abhängig vom Parameter *_on* ein Attribut hinzufügen (*_on=True*) oder entfernen (*_on=False*). Ob ein Objekt über ein bestimmtes Attribut verfügt, können Sie im Spiel mit der Methode

```
TItem.Has(a : BYTE) : BOOLEAN
```

prüfen. Zehn Attribute sind bereits als Konstanten mit der Endung *_at deklariert und können beliebig kombiniert werden. Die einzelnen Attribute haben folgende Bedeutung:

- **drinkable_at** - das Objekt kann getrunken werden.

- **edable_at** - das Objekt kann gegessen werden.

- **enterable_at** - das Objekt kann vom Spieler betreten werden. Betreten meint hierbei sowohl hineingehen (Telefonzelle) als auch draufsetzen (Fahrrad). Wenn Sie dieses Attribut setzen, sollten Sie daher mit der Methode *TItem.SetPraepos(pp:String)* auch die entsprechende Präposition, meistens "in" oder "auf", angeben, damit Textopia den Umstand, daß der Spieler ein Objekt betreten hat, korrekt ausdrücken kann.

- **moveable_at** - der Spieler kann sich mit dem Objekt herumbewegen, wenn er es betreten hat. Ein Objekt mit den Attributen moveable_at und enterable_at kann somit ein Fahrzeug simulieren.

- **readable_at** - das Objekt kann gelesen werden (Bücher, Schilder etc.).

- **shining_at** - das Objekt ist eine Lichtquelle. Eine besondere Aktion kann der Spieler hiermit nicht ausführen. Lichtquellen sind aber in ansonsten dunklen Räumen (s. 8.4) sehr praktisch.

- **switchable_at** - das Objekt kann ein- und ausgeschaltet werden. Nach der Initialisierung ist ein Objekt immer ausgeschaltet. Den aktuellen Schalterzustand können Sie im Spiel mit den Methoden *TItem.IsOn* und *TItem.IsOff* überprüfen. Die Methoden *TItem.SwitchOn* und *TItem.SwitchOff* verändern den Schalterzustand.

- **takeable_at** - das Objekt kann vom Spieler weggelegt, aufgenommen und im Inventar mit herumgetragen werden.

- **talkable_at** - das Objekt kann angeredet werden (NPCs, Mikrophone etc.). Um ein Objekt anzusprechen, muß der Spieler den Objektnamen gefolgt von einem Komma, einem Leerzeichen und einem Text in doppelten Anführungszeichen eingeben. Näheres über NPCs und Dialoge erfahren Sie in 8.1.

- **transparent_at** - das Objekt ist durchsichtig (aber nicht unsichtbar), was für Behälter bedeutet, daß ihr Inhalt auch im geschlossenen Zustand sichtbar bleibt.

Einige Kombinationen der Attribute ergeben sicher sehr surrealistische Objekte. Neben diesen zehn Attributen können Sie auch eigene Attribute deklarieren. Für deren Konstanten können Sie Werte zwischen 10 und 255 wählen. Natürlich müssen Sie dann auch einige Methoden an die neuen Attribute anpassen, lesen Sie also erstmal weiter.

Das grade erzeugte Schwert braucht der Spieler nur mit sich herumzutragen:

```
WITH item^ DO
BEGIN
  SetAttrib(takeable_at,t);
```

Jetzt braucht muß noch der Text geschrieben werden, der erscheint, wenn der Spieler näheres über das Schwert erfahren will. Für die Beschreibung von Gegenständen können wir wieder *AddText()* verwenden:

```
 AddText('Das Schwert hat der Waffenschmied Ihnen kurz vor Ihrer '+
         'Abreise gegeben. Es schimmert und ist sehr scharf. Den '+
         'Schmied zu überreden, Ihnen das Schwert anzuschreiben, weil '+
         'Sie die nötigen Taler erst noch in den Gräbern finden '+
         'müßten, war nicht einfach.\n');
END;
```

Wenn Sie einem Gegenstand weder mit *AddText()* noch mit der Methode *MyText* eine Beschreibung hinzufügen und der Spieler das Objekt untersucht, gibt Textopia die Meldung "An xxx ist nichts Besonderes zu entdecken" aus.

5.2 Behälter

Nach dem Schwert erzeugen wir noch einen Rucksack:

```
NEW(item,Init(rucksack_id,'+Rucks%ack#e',NIL,t,1,5));
```

Den Rucksack kann der Spieler wie das Schwert mit sich herumtragen:

```
WITH item^ DO
BEGIN
  SetAttrib(takeable_at,t);
END;
```

Zusätzlich soll der Rucksack noch als Behälter dienen. Einen Behälter kann der Spieler öffnen, schließen und abschließen. Natürlich kann er in einen Behälter auch andere Gegenstände hineinlegen (und wieder herausnehmen). Um aus einem Gegenstand einen Behälter zu machen, muß ein Objekt vom Typ *Tlock* erzeugt und dem Gegenstand zugeordnet werden. *TLock* verwaltet einige Informationen, die alle Objekte benötigen, die sich öffnen und schließen lassen. Daher wird *TLock* Ihnen auch bei der Erzeugung von Türen wiederbegegnen. Objekte vom Typ *TItem* und *TLink* besitzen einen Zeiger namens lock, der auf das ihnen zugeordnete *TLock*-Objekt zeigt (und der natürlich *NIL* ist, wenn das entsprechende Objekt kein Behälter und keine Tür ist). Über diesen Zeiger können Sie auf die Methoden von *TLock* zugreifen. Der Konstruktor von *TLock* verlangt fünf Parameter:

- **_owner: PBasic** - einen Zeiger auf ein Objekt vom Typ *TItem* oder *TLink*, das als Behälter oder als Tür dienen soll. Einen Zeiger auf das entsprechende Objekt liefert Ihnen während des Spiels die Methode *TLock.GetOwner*.

- **_openable: BOOLEAN** - bestimmt, ob der Spieler das Objekt öffnen kann. Im Spiel können Sie diese Voreinstellung mit der Methode *TLock.SetOpenable(s:Boolean)* ändern. Den aktuellen Zustand dieses Parameters liefert die Methode *TLock.IsOpenable*.

- **_closeable: BOOLEAN** - bestimmt, ob der Spieler das Objekt schließen kann. Diese Voreinstellung können Sie im Spiel mit der Methode *Tlock.SetCloseable(s:Boolean)* ändern. Den aktuellen Zustand dieses Parameters liefert die Methode *TLock.IsCloseable*.

- **_state: TState** - Zustand des Objekts bei Spielbeginn, kann die Werte *open*, *closed* oder *locked* annehmen. Den aktuellen Zustand liefert Ihnen während des Spiels die Methode *TLock.GetState*. Wenn Sie das Schloß einer Verbindung zugeordnet haben, können Sie alternativ auch die Methode *TRoom.IsGate()* benutzen, die Sie informiert, ob die Verbindung in der übergebenen Richtung offen, geschlossen oder verschlossen ist.

- **_key: PItem** - einen Zeiger auf einen Gegenstand, der als Schlüssel dient. Nur mit diesem Schlüssel kann der Spieler das Objekt auf- und abschließen. Wird hier *NIL* angegeben, kann der Spieler das Objekt nur öffnen (wenn es nicht abgeschlossen ist) und schließen, nicht aber abschließen. Der als Schlüssel dienende Gegenstand kann natürlich auch eine Magnetkarte oder Brechstange sein. Im Spiel gelangen Sie über *TLock.GetKey* wieder an den Schlüssel.

Um eine Instanz von *TLock* zu erzeugen, brauchen wir einen neuen Zeiger. Fügen Sie daher im Var-Abschnitt die Zeile

```
lock : PLock;
```

ein. Mit item wurde der Rucksack erzeugt, dieser Zeiger kann daher gleich dem Konstruktor von *TLock* übergeben werden:

```
NEW(lock,Init(item,t,t,closed,NIL));
```

Auf eine besondere Beschreibung des Rucksacks soll einmal verzichtet werden. Wenn Sie während des Spiel auf einen Behälter zugreifen wollen, sollten Sie noch folgende Dinge wissen: Um festzustellen, ob ein Behälter ein bestimmtes Objekt enthält, verwenden Sie die Methode

```
TItem.Contains(x : PItem) : BOOLEAN.
```

Umgekehrt liefert die Methode *TItem.GetContainer* einen Zeiger vom Typ *PItem* auf den Behälter, der das Objekt enthält. Objekte vom Typ *TItem* und *Tplayer* enthalten ein statisches Objekt namens weight. Dessen Methoden *GetCont*, *GetSum* und *GetMax* liefern die Zahl der im Behälter enthaltenen Objekte, das Gesamtgewicht des Behälters und sein zulässiges Maximalgewicht.

Wenn Sie jetzt das Programm demo2.pas starten, können Sie mit unserem Mini-Adventure schon etwas mehr anfangen:

```
>#i
Sie tragen folgende Dinge mit sich:
Ein Schwert.
Einen Rucksack.
2 Item(s)  Gewicht=3

>öffne den rucksack
Sie öffnen den Rucksack.

>lege das schwert in den rucksack
Sie legen ein Schwert in den Rucksack.

>#i
Sie tragen folgende Dinge mit sich:
Einen Rucksack, der Rucksack enthält:
  Ein Schwert.
2 Item(s)  Gewicht=3

>nimm das schwert aus dem rucksack
Sie nehmen ein Schwert.

>#i
Sie tragen folgende Dinge mit sich:
Ein Schwert.
Einen Rucksack.
2 Item(s)  Gewicht=3
```

Wenn Sie meinen, ein Schwert sei zu groß, um in einen Rucksack gelegt zu werden, brauchen Sie nur das Gewicht des Schwerts zu erhöhen, so daß es mehr wiegt als der Rucksack aufnehmen kann. Hierbei werden Sie auch merken, daß Textopia nicht unterscheiden kann, ob ein Objekt von einem Behälter nicht aufgenommen werden kann, weil es zu schwer oder weil es zu groß ist. Für diese Unterscheidung müßte neben dem Gewicht eines Objekts auch noch dessen Größe festgelegt werden (was die Verwaltung der Behälter wesentlich komplizierter machen würde). In den meisten Spielen wird die Gewichtsangabe sicher ausreichen.

5.3 Fortbewegungsmittel

Eine weitere Gruppe nützlicher Gegenstände sind Fortbewegungsmittel wie Reittiere und Fahrzeuge aller Art. Wenn der Spieler sie betreten hat, bewegen sie sich mit ihm durch die Räume. Geben Sie einem *TItem*-Objekt, das dem Spieler zur Fortbewegung dienen soll, die Attribute enterable_at und moveable_at.

Der Spieler im "Zwergengrab" soll mit einem Pferd durch den Wald reisen können. Wir deklarieren die Konstante horse_id und erzeugen vor der Initialisierung des Spielers ein Pferd:

```
horse_id = 13;

NEW(item,Init(horse_id,'Pferd#e',Adr(forest_id),t,1,10);
WITH item^ DO
BEGIN
  SetAttrib(enterable_at,t);
  SetAttrib(moveable_at,t);
  SetPraepos('auf');
```

```
END;
```

Mit *SetPraepos()* bestimmen Sie, ob der Spieler sich in oder auf einem Fahrzeug befindet (s. 5.1). Um den Spieler zu Spielbeginn gleich aufs Pferd zu setzen, muß im Konstruktor von *TPlayer* als Startposition jetzt horse_id statt forest_id angegeben werden. Mit demo3.pas lernen Sie den Umgang mit einem Pferd:

WALD (auf einem Pferd)
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter westlich scheinen die Bäume weniger dicht beieinander zu stehen.

>steig vom pferd
Ok

>schau
WALD
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter westlich scheinen die Bäume weniger dicht beieinander zu stehen.
Ein Pferd.

>steig aufs pferd
Ok

>schau
WALD (auf einem Pferd)
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter westlich scheinen die Bäume weniger dicht beieinander zu stehen.

>us pferd
An dem Pferd ist nichts Besonderes zu entdecken.

Statt "steig aufs pferd" versteht der Parser natürlich auch "steige auf das pferd". Wie Sie erkennen können, wird ein Fortbewegungsmittel nicht in der Raumbeschreibung erwähnt, wenn der Spieler sich darin oder darauf befindet. Beachten Sie auch, daß der Spieler, wenn er sich in einem betretbaren Objekt befindet, nicht an Objekte herankommt, die sich außerhalb des den Spieler enthaltenen Objekts befinden (In einem Auto oder einer Telefonzelle kann man z.B. schlecht eine auf der Straße liegende Münze aufheben).

5.4 Weitere Räume

Einen Wald haben wir bereits in Abschnitt 4.3 erzeugt. Für unser Mini-Adventure brauchen wir noch vier weitere Räume: Vom Wald aus soll der Spieler in westlicher Richtung auf eine Lichtung mit einem Hügelgrab gelangen. In dem Hügel befindet sich ein Tor, über eine dahinterliegende Treppe gelangt der Spieler in die unter dem Hügel liegende Grabkammer. Im Südwesten dieser Kammer soll eine weitere Höhle liegen. Im Osten plazieren wir eine Schatzkammer, die durch eine Tür zunächst versperrt wird. Die Karte sieht dann wie folgt aus:

```
        Hügel -- Wald
         !
   Grabkammer -- Schatzkammer
  /
Höhle
```

Beginnen wir mit der Deklaration der Konstanten für die neuen Räume:

```
hill_id    = 14;
grave_id   = 15;
chamber_id = 16;
cave_id    = 17;
```

Nach der Initialisierung des Waldes können jetzt die neuen Räume hinzugefügt werden:

```
NEW(room,Init(hill_id,'-Lichtung#en'));
NEW(room,Init(grave_id,'-Grabkammer#n'));
WITH room^ DO
BEGIN
  AddText('In der Mitte der engen Grabkammer steht ein Sarkophag. In der '+
          'Ostwand befindet sich eine schwere, mit Eisen beschlagene '+
          'Eichentür. Im Südwesten führt ein schmaler Gang weiter in '+
          'eine hinter der Grabkammer liegende Höhle.\n');
END;
NEW(room,Init(chamber_id,'-Schatzkammer#n'));
WITH room^ DO
BEGIN
  AddText('In dieser Kammer lagerten die Grabbeigaben des Königs.\n');
END;
NEW(room,Init(cave_id,'-Höhle#n'));
WITH room^ DO
BEGIN
  AddText('Sie befinden sich in der kleinen Höhle im Südwesten der '+
          'Grabkammer. Diese Höhle wurde offensichtlich nicht von den '+
          'Zwergen angelegt, als sie das Grab schufen. Die Wand der '+
          'Grabkammer wurde für die Höhle durchbrochen. Ihre Wände ');
  AddText('sind mit scheußlichen Symbolen übersät, die nichts mit den '+
          'Schriftzeichen der Zwerge auf dem Sarkophag zu tun haben.\n');
END;
```

Die Lichtung braucht eine variable Beschreibung (s. 4.4), die erwähnen soll, ob das Tor geöffnet oder geschlossen ist. Diese Beschreibung folgt in 5.6. Wenden wir uns zunächst den Dingen in der Grabkammer zu. Der Sarkophag stellt einen Behälter dar. Er enthält einen Knochen und wird von einer unbeweglichen Steinplatte halb verschlossen:

```
sarko_id = 18;
stone_id = 19;
bone_id  = 20;

NEW(item,Init(sarko_id,'+Sarkophag',Adr(grave_id),f,1,5));
NEW(lock,Init(item,f,f,open,NIL));
NEW(item,Init(stone_id,'-Steinplatte#n',Adr(grave_id),f,1,0));
NEW(item,Init(bone_id,'+Knochen',Adr(sarko_id),t,1,0));
WITH item^ DO
BEGIN
  SetAttrib(takeable_at,t);
  AddText('Dies ist der einzige Knochen, der von dem hier bestatteten '+
          'König geblieben ist. Wenn Sie genau hinschauen, entdecken Sie '+
          'an dem Knochen tiefe Kratzer, die nur von sehr scharfen '+
          'Zähnen stammen können.\n');
END;
```

5.5 Variable Objektbeschreibungen

Der Knochen soll bei der Beschreibung des Sarkophags erwähnt werden. Da der Spieler den Knochen auch herausnehmen kann, sollte der Sarkophag eine variable Raumbeschreibung erhalten. Statt des statischen *AddText()* muß also die Methode *MyText* verwendet werden.

Hierzu deklarieren wir einen Nachfolger von *TItem*, der ja auch ein Nachfolger von *TBasic* ist, und bringen die Beschreibung des Sarkophags in der überschriebenen Methode *MyText* unter:

```
TYPE
  TnItem = OBJECT(TItem)
             PROCEDURE MyText; VIRTUAL;
           END;
  PnItem = ^TnItem;
```

Im Var-Abschnitt muß der Typ des Zeigers item in *PnItem* geändert werden:

```
item : PnItem;
```

Jetzt folgt die Neudefinition von *TnItem.Mytext*, die die variable Beschreibung des Sarkophags enthält. Beachten Sie, daß eine überschriebene Methode von allen Instanzen des entsprechenden Objekttyps verwendet werden kann. Daher muß in *TnItem.MyText* mit der Methode

```
TBasic.GetID : WORD
```

festgestellt werden, für welches Objekt die Beschreibung ausgegeben werden soll:

```
PROCEDURE TnItem.MyText;
BEGIN
  CASE GetID OF
    sarko_id : BEGIN
                 Print('In der Ihnen unbekannten Zwergensprache ist ein '+
                       'Text in die Seiten des Sarkophags gemeißelt. Die '+
                       'schwere Steinplatte, die den Sarkophag einmal '+
                       'verschloß, ist halb zur Seite geschoben. ');
                 IF Contains(Adr(bone_id)) THEN
                    Print('Im Sarkophag liegt ein einzelner Knochen.\n')
                    ELSE Print('Von dem Zwergenkönig ist im Sarkophag '+
                               'nichts mehr zu entdecken.\n');
               END;

  END;
END;
```

5.6 Verbindungen und Türen

Die Tür zur Schatzkammer soll sich mit einem Schlüssel öffnen lassen, den der Spieler in der Höhle finden kann. Als Schlüssel dienende Objekte müssen vor der Initialisierung der entsprechnenden Schlösser initialisiert werden:

```
key_id = 21;

NEW(item,Init(key_id,'+Schlüssel',Adr(cave_id),t,1,0));
item^.SetAttrib(takeable_at,t);
```

Auf der Karte gibt es vier Verbindungen zwischen den Räumen. Jede von ihnen braucht eine eigene ID. Bei umfangreicheren Karten sollten den Verbindungen aussagekräftigere Namen gegeben werden, hier numerieren wir einfach:

```
link1_id = 22;
link2_id = 23;
link3_id = 24;
```

```
link4_id = 25;
```

Im Var-Abschnitt müssen jetzt noch Zeiger vom Typ *PLink* und *PLock* deklariert werden:

```
link : PLink;
lock : PLock;
```

Der Konstruktor von *TLink* verlangt zunächst wieder nach einer ID und einem Namen für die Verbindung. Der Name kann entfallen, wenn die Verbindung keine Tür oder ein ähnliches Hindernis darstellen soll. Geben Sie dann einfach einen Leerstring ('') an. Nach ID und Namen müssen noch folgende vier Parameter angegeben werden:

- **_from:Word** - die ID des ersten Raums.
- **_dirto:TDir** - Himmelsrichtung, in der der Spieler den ersten Raum verlassen kann, um den zweiten zu erreichen. Zusätzlich zu den zehn Himmelsrichtungen (north, northeast usw.) können Sie auch up und down angeben.
- **_to:Word** - die ID des Raums, den der Spieler aus angegebener Himmelsrichtung vom ersten Raum aus erreicht.
- **_show:Boolean** - bestimmt, ob die Verbindung automatisch nach der Raumbeschreibung erwähnt wird.

Die Zeile

```
NEW(link,Init(link1_id,'',forest_id,west,hill_id,f));
```

verbindet den Wald in westlicher Richtung mit dem Hügelgrab auf der Lichtung. Die Verbindung zwischen dem Hügel und der Grabkammer wird mit

```
NEW(link,Init(link2_id,'Tor',hill_id,down,grave_id,f));
```

erzeugt. Damit der Spieler das Tor öffnen und schließen kann, muß dieser Verbindung noch ein Objekt vom Typ *TLink* zugeordnet werden:

```
NEW(lock,Init(link,t,t,closed,NIL));
```

Entsprechend werden die übrigen Verbindungen und Türen erzeugt:

```
NEW(link,Init(link3_id,'-Eichentür#en;-Tür#en',grave_id,east,chamber_id,
    f));
NEW(lock,Init(link,t,t,locked,Adr(key_id)));
NEW(link,Init(link4_id,'',grave_id,southwest,cave_id,f));
```

Es fehlt noch die Beschreibung der Lichtung, die dem Spieler mitteilen soll, ob das Tor im Hügel geöffnet oder geschlossen ist. Hierbei wird ähnlich wie bei der Neudefinition von *TItem.Mytext* vorgegangen, mit dem Unterschied, daß diesmal ein Nachfolger von *TRoom* benötigt wird. Im Type-Abschnitt wird die Deklaration

```
TnRoom = OBJECT(TRoom)
           PROCEDURE MyText; VIRTUAL;
         END;
```

```
PnRoom = ^TnRoom;
```

hinzugefügt. Im Var-Abschnitt wird der Zeiger room geändert:

```
room : PnRoom;
```

Die variable Raumbeschreibung der Lichtung sieht wie folgt aus:

```
PROCEDURE TnRoom.MyText;
BEGIN
  CASE GetID OF
    hill_id : BEGIN
                Print('Ein kleiner Hügel erhebt sich in der Mitte einer '+
                      'Lichtung über das Grab eines Zwergenkönigs. ');
                IF gate[down]^.lock^.GetState=closed THEN
                   Print('Ein steinernes Tor in der Nordseite versperrt '+
                         'Ihnen den Weg ins Innere.\n')
                   ELSE Print('Das steinerne Tor in der Nordseite steht '+
                              'offen. Dahinter führt eine Treppe in '+
                              'das Grab hinab.\n');
              END;
  END;
END;
```

Das Array Gate[] ist ein Datenfeld von *TRoom* und enthält Zeiger auf die Verbindungen in den entsprechenden Himmelsrichtungen. Mit der Abfrage

```
IF gate[south]<>NIL
```

können Sie z.B. feststellen, ob im Süden eine Verbindung zu einem anderen Raum besteht. Ob einer Verbindung ein Schloß zugeordnet ist, teilt Ihnen die Abfrage

```
IF gate[down]^.lock<>NIL
```

mit. Wenn der Verbindung ein Schloß zugeordnet ist, können Sie über lock auf die Methoden von *TLock* zugreifen. Statt dem Konstrukt *gate[down]^.lock^.GetState* läßt sich auch das kürzere *TRoom.IsGate(down)* verwenden.

Wenn Sie jetzt einmal demo4.pas starten, ist folgender Dialog möglich:

WALD (auf einem Pferd)
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter westlich scheinen die Bäume weniger dicht beieinander zu stehen.

>w
LICHTUNG (auf einem Pferd)
Ein kleiner Hügel erhebt sich in der Mitte einer Lichtung über das Grab eines Zwergenkönigs. Ein steinernes Tor in der Nordseite versperrt Ihnen den Weg ins Innere.

>steig vom pferd
Ok

>öffne das tor
Sie öffnen das Tor.

>runter
GRABKAMMER

```
In der Mitte der engen Grabkammer steht ein Sarkophag. In der Ostwand
befindet sich eine schwere, mit Eisen beschlagene Eichentür. Im Südwesten
führt ein schmaler Gang weiter in eine hinter der Grabkammer liegende Höhle.

>us sarkophag
In der Ihnen unbekannten Zwergensprache ist ein Text in die Seiten des
Sarkophags gemeißelt. Die schwere Steinplatte, die den Sarkophag einmal
verschloß, ist halb zur Seite geschoben. Im Sarkophag liegt ein einzelner
Knochen.

>nimm den knochen
Sie nehmen einen Knochen.

>us knochen
Dies ist der einzige Knochen, der von dem hier bestatteten König geblieben
ist. Wenn Sie genau hinschauen, entdecken Sie an dem Knochen tiefe Kratzer,
die nur von sehr scharfen Zähnen stammen können.

>sw
HÖHLE
Sie befinden sich in der kleinen Höhle im Südwesten der Grabkammer. Diese
Höhle wurde offensichtlich nicht von den Zwergen angelegt, als sie das Grab
schufen. Die Wand der Grabkammer wurde für die Höhle durchbrochen und die
Wände der Höhle sind mit scheußlichen Symbolen übersät, die sich stark von
den Schriftzeichen der Zwerge auf dem Sarkophag unterscheiden.
Ein Schlüssel.

>nimm den schlüssel
Sie nehmen einen Schlüssel.

>no
GRABKAMMER
In der Mitte der engen Grabkammer steht ein Sarkophag. In der Ostwand
befindet sich eine schwere, mit Eisen beschlagene Eichentür. Im Südwesten
führt ein schmaler Gang weiter in eine hinter der Grabkammer liegende Höhle.

>öffne die tür mit dem schlüssel
Sie öffnen die Eichentür.

>o
SCHATZKAMMER
In dieser Kammer lagerten die Grabbeigaben des Königs.
```

Wenn Sie wollen, können Sie in der Schatzkammer noch verschiedene Schätze wie Münzen, Kronen und Waffen unterbringen. Denken Sie daran, daß mehrfach vorhandene Dinge (z.B. mehrere Goldmünzen) unterschiedliche IDs aber identische Namen bekommen müssen und maximal zehnfach vorhanden sein dürfen.

Den Weg des Spielers durch Ihre Räume können Sie mit den Methoden

```
TRoom.FromDir : TDir
```

und

```
TRoom.ToDir : TDir
```

verfolgen. Aus der Richtung *FromDir* hat der Spieler den Raum betreten, in die Richtung *ToDir* hat er ihn verlassen. Wenn der Spieler den Raum noch nicht betreten oder verlassen hat, liefern diese Methoden den Wert *nowhere*.

5.7 Der Scope

In 5.3 haben Sie bereits erfahren, daß der Spieler Objekte nicht erreichen kann, die sich außerhalb des Behälters oder Fahrzeugs befinden, in dem der Spieler sich befindet. Auch in anderen Fällen kann ein Objekt für den Spieler unerreichbar sein. Jeder Nachfolger von *TBasic* verfügt über die Methode

```
TBasic.Scope : BYTE
```

An ihrem Rückgabewert können Sie erkennen, inwieweit das entsprechende Objekt für den Spieler erreichbar ist. Die einzelnen Werte von *Scope* sind in tmain.pas als Konstanten mit der Endung *_sd deklariert und haben folgende Bedeutung:

- **notinroom_sd** - Das Objekt befindet sich nicht mit dem Spieler in einem Raum, es ist daher außer Sicht- und Reichweite.

- **visible_sd** - Das Objekt ist nur sichtbar, kann vom Spieler aber nicht berührt oder aufgenommen werden. Dies tritt z.B. ein, wenn das Objekt sich in einem verschlossenen, transparenten Behälter befindet.

- **reachable_sd** - Das Objekt befindet sich mit dem Spieler in einem Raum, aber nicht im Inventar des Spielers oder eines NPC.

- **bynpc_sd** - Das Objekt wird von einem NPC (s. 8.1) gehalten.

- **held_sd** - Das Objekt befindet sich im Inventar des Spielers.

Die Methode *Scope* ist in allen Nachfolgern von *TBasic* unterschiedlich definiert, so kann z.B. der Scope eines Raums niemals gleich held_sd sein.

6 Aktionen und Reaktionen

6.1 Ereignisse

Die bisherige Version unseres Mini-Adventures (demo4.pas) läßt dem Spieler eine etwas zu große Bewegungsfreiheit: Nachdem er vom Pferd gestiegen ist und das Tor geöffnet hat, kann er sich wieder auf das Pferd setzen und in die Grabkammer hineinreiten. Wir wollen aber annehmen, daß das Tor und das Grab zu eng und zu klein sind, um sich auf einem Pferd hineinzubegeben. Der Spieler sollte also daran gehindert werden, das Grab zu betreten, wenn er sich noch auf dem Pferd befindet.

Um auf Aktionen des Spielers (hier: das Betreten der Grabkammer mit einem Pferd) Einfluß zu nehmen, müssen Sie zunächst wissen, wie diese von Textopia ausgeführt werden.

Der Parser versucht, jeder Eingabe des Spielers ein ihm bekanntes Ereignis zuzuordnen. Ein Ereignis tritt z.B. ein, wenn der Spieler ein Objekt untersucht, einen Behälter öffnet oder den Spielstand speichert. Textopia kennt insgesamt 40 Ereignisse, die im Interface-Abschnitt von tmain.pas als Konstanten mit der Endung *_ev deklariert sind. In Abschnitt 7.2 werden Sie erfahren, wie Sie eigene Ereignisse hinzufügen können.

Die einzelnen Ereignisse können von einer normalen Spielereingabe, einem Metaverb (s. 7.5) oder einem Debugkommando (s. 7.6) ausgelöst werden und haben folgende Bedeutung:

close_ev	Ein Schloß schließen
dance_ev	Spieler tanzt
dec_ev	Debugkommando .dec
drink_ev	Spieler trinkt Objekt
eat_ev	Spieler ißt Objekt
enter_ev	Spieler betritt Objekt
examine_ev	Spieler untersucht Objekt
give_ev	Spieler gibt NPC Objekt
go_ev	Spieler verläßt Raum
inv_ev	Metaverb #i
drop_ev	Spieler legt Objekt weg
jump_ev	Spieler springt in die Luft
kill_ev	Spieler tötet NPC
kiss_ev	Spieler küßt NPC
leave_ev	Spieler verläßt zuvor betretenes Objekt
lista_ev	Debugkommando .browse
lists_ev	Debugkommando .list
load_ev	Metaverb #laden
lock_ev	Schloß abschließen
look_ev	Raumbeschreibung
map_ev	Metaverb #karte
open_ev	Schloß öffnen
press_ev	Spieler drückt Objekt
quit_ev	Metaverb #ende
restart_ev	Metaverb #neu
read_ev	Spieler liest Objekt

score_ev	Metaverb #punkte
save_ev	Metaverb #sichern
set1_ev	Metaverb #status
set2_ev	Metaverb #beschreibung
show_ev	Spieler zeigt NPC Objekt
sleep_ev	Spieler schläft
smell_ev	Spieler riecht an Objekt
switchoff_ev	Spieler schaltet Objekt aus
switchon_ev	Spieler schaltet Objekt ein
take_ev	Spieler nimmt Objekt auf
tell_ev	Spieler spricht zu NPC
touch_ev	Spieler berührt Objekt
trace_ev	Debugkommando .trace
wait_ev	Spieler wartet

An einem Ereignis können bis zu zwei Objekte beteiligt sein. Die betroffenen Objekte erwähnt der Spieler meist in der Eingabe: "Lege das Schwert in den Rucksack" oder "Öffne das Tor". Auch Eingaben, in denen der Spieler kein Objekt der Spielwelt ausdrücklich erwähnt, können sich auf ein bestimmtes Objekt beziehen. Das Kommando "schau" bezieht sich z.B. auf den aktuellen Raum und bewirkt die Ausgabe der Beschreibung eines Objekts vom Typ *TRoom*.

Die an einem Ereignis beteiligten Objekte werden vom Parser benachrichtigt und führen die vom Spieler gewünschte Aktion aus. Dabei ruft der Parser die Methode

```
HandleEvent(VAR event : TEvent); VIRTUAL;
```

auf, die alle Nachfolger von *TBasic* und *TGame* besitzen. Der Methode wird ein Record vom Typ *TEvent* übergeben, in der der Parser einige Informationen über die Eingabe speichert. Von den Datenfeldern dieses Records soll zunächst nur das Feld action vom Typ *Byte* interessieren. In action ist der Wert der dem Ereignis zugeordneten Konstante gespeichert. In der Methode *HandleEvent()* kann so festgestellt werden welches Ereignis ausgeführt werden soll.

Die *HandleEvent()*-Methoden der einzelnen Objekttypen kümmern sich nur um die Ereignisse, die mit dem jeweiligen Objekttyp auch einen Sinn ergeben. Die von *TBasic* abgeleiteten Objekttypen leiten ein Ereignis an ihren Vorfahren weiter, wenn sie es nicht selbst abarbeiten. Wenn auch *TBasic* nicht für das Ereignis zuständig ist, wird die Methode *TGame.HandleEvent()* aufgerufen. Folgende Tabelle zeigt, welcher Objekttyp für welches Ereignis zuständig ist:

TItem	*TRoom*	*TLink*	*TLock*	*TBasic*	*TGame*
close_ev	go_ev	close_ev	close_ev	dance_ev	lista_ev
drink_ev	look_ev	lock_ev	lock_ev	dec_ev	sleep_ev
drop_ev		open_ev	open_ev	examine_ev	wait_ev
eat_ev				jump_ev	
enter_ev				lists_ev	
give_ev				touch_ev	
kill_ev				trace_ev	
kiss_ev					
leave_ev					
lock_ev					
open_ev					
read_ev					
show_ev					
switchoff_ev					
switchon_ev					
press_ev					
take_ev					
tell_ev					

TItem und *TLink* leiten die Ereignisse close_ev, lock_ev und open_ev an das ihnen zugeordnete Objekt vom Typ *TLock* weiter. Die Ereignisse inv_ev, load_ev, map_ev, quit_ev, restart_ev, save_ev, score_ev, set1_ev und set2_ev gehören zu den Metaverben und werden in Abschnitt 7.4 behandelt.

6.2 BeforeAction und AfterAction

Da jedes Objekt mehrere Ereignisse abarbeiten kann, erfolgt die Verarbeitung der einzelnen Ereignisse in der Methode *HandleEvent()* immer in einer Case-Anweisung:

```
PROCEDURE Txxx.HandleEvent(VAR event : TEvent);
BEGIN
  WITH event DO
  BEGIN
    CASE action OF
      yyy_ev : { Code... }
      zzz_ev : { Code... }
      ELSE vorgaenger_von_Txxx.HandleEvent(event);
    END;
  END;
END;
```

Die Methode kann auch einen Var-Abschnitt mit lokalen Variablen enthalten. Wenn Sie die Verarbeitung eines einzelnen Ereignisses beeinflussen wollen, müssen Sie nicht die ganze Methode mit allen Ereignissen überschreiben. Der zu einem Ereignis gehörende Codeblock beginnt immer mit dem Aufruf der Methode

```
BeforeAction(VAR event : TEvent) : BOOLEAN; VIRTUAL;
```

BeforeAction() liefert einen Wahrheitswert zurück (normalerweise immer *True*), von dem die Ausführung der weiteren Anweisungen abhängt. Wenn diese Methode *False* zurückgibt, wird das betreffende Ereignis nicht weiter ausgeführt. Wenn Sie ein bestimmtes Ereignis beeinflussen wollen, müssen Sie also nur die Methode *BeforeAction()* des zuständigen Objekttyps überschreiben, in einer Case-Anweisung das Ereignis abfangen und entscheiden, ob die Ausführung in *HandleEvent()* fortgesetzt werden soll. Sie können so auch den kompletten Code für die Ausführung eines Ereignisses durch eigene Anweisungen ersetzen.

In unserem Beispiel müssen wir verhindern, daß der Spieler auf seinem Pferd in das Grab eindringt. Dafür müssen wir das Ereignis go_ev abfangen und prüfen, ob der Spieler auf der Lichtung mit dem Pferd nach unten in den Hügel gehen will. Der Deklaration von *TnRoom* fügen Sie den Methodenkopf von *BeforeAction()* hinzu und schreiben diese Methode neu:

```
FUNCTION TnRoom.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    CASE action OF
      go_ev : IF ToDir=down THEN
              BEGIN
                ok:=player^.GetContainer=NIL;
                IF NOT(ok) THEN
                   Print('Mit dem Pferd können Sie hier nicht hinein.\n');
              END;
    END;
  END;
  BeforeAction:=ok;
END;
```

Ausprobieren können Sie diese Erweiterung mit dem Programm demo5.pas.

Wenn ein Ereignis ausgeführt wurde, gibt *HandleEvent()* eine entsprechende Mitteilung aus, die den Spieler über die erfolgreiche Ausführung seiner Aktion informiert (z.B. "Sie nehmen x" nach der Eingabe "nimm x"). Wenn Sie diese Meldung verändern oder unterdrücken wollen, müssen Sie die zu *BeforeAction()* analoge Methode

```
AfterAction(event : TEvent) : BOOLEAN; VIRTUAL
```

überschreiben, die nach der Ausführung eines Ereignisses und vor Ausgabe einer Meldung aufgerufen wird. Liefert *AfterAction()* den Wert *False* zurück, so wird die Meldung unterdrückt.

Die in tio.pas deklarierte globale Variable replay vom Typ *Boolean* wird bei jeder Textausgabe mit *Print()* auf *True* gesetzt. An ihr erkennt der Parser, ob während der Ausführung eines Ereignisses bereits ein Text ausgegeben wurde. Führt ein Ereignis nicht zur Ausgabe irgendeines Textes, so gibt der Parser anschließend ein schlichtes "Ok" aus. Manche Ereignisse bestehen nur in der Ausgabe einer Meldung, z.B. bewirkt das Kommando "warte" die Ausgabe "Die Zeit vergeht...", ohne daß irgendwelche weiteren Methoden ausgeführt werden. In solchen Fällen kann *AfterAction()* die Ausführung eines Ereignisses also verhindern oder ändern.

Während der Ausführung eines Ereignisses können Sie auch weitere Ereignisse auslösen, indem Sie die Methode *HandleEvent()* eines beliebigen Objekts mit der Variable event und einem veränderten Wert für action aufrufen. Textopia tut dies z.B. bei der Ausführung des Ereignisses go_ev in der Methode *TRoom.HandleEvent().* Wenn der Spieler mit einem Schlüssel durch eine mit diesem Schlüssel verschlossene Tür gehen will, wird zunächst das

Ereignis open_ev ausgelöst. Schauen Sie sich hierzu einmal den Quelltext von *TRoom.HandleEvent()* an.

7 Der Parser

7.1 Die Eingabe

Der Parser (von engl. *to parse*: einen Satz grammatisch analysieren) ist der wichtigste Bestandteil jedes Text-Adventures. Er nimmt die Spielereingabe entgegen, analysiert sie und sorgt dafür, daß die Befehle des Spielers ausgeführt werden. Eine Schleife in der Methode *TGame.Run* wartet in jedem Spielzug auf eine Spielereingabe und übergibt sie dem Parser.

Die Spielereingabe wird von der in tmain.pas definierten Prozedur

```
Scan(VAR line : STRING; prompt,upsize : BOOLEAN; lmem : BYTE);
```

entgegengenommen. *Scan()* bildet einen kompletten Zeileneditor, mit dem Sie auch selber Eingaben des Spielers anfordern können. Ihre Parameter haben folgende Bedeutung:

- **line** - in diesem String wird die Spielereingabe eingetragen.

- **prompt** - bestimmt, ob das Eingabeaufforderungszeichen > ausgegeben wird.

- **upsize** - bestimmt, ob die Eingabe in Großbuchstaben erfolgt. Es gilt die in *TGame.Init()* (s. 4.5) mit dem Parameter _upsize angegebene Voreinstellung.

- **lmem** - Größe des Zeilenspeichers. Es gilt die in *TGame.Init()* (s. 4.5) mit dem Parameter _history angegebene Voreinstellung.

Die Eingabe kann mit den Tasten [←], [Einfg], [Pos1], [Ende] und [Esc] editiert werden. Mit den Tasten CursorOben/CursorUnten kann der Zeilenspeicher durchblättert werden. Groß- und Kleinschreibung werden nicht unterschieden. Intern wird die Eingabe in Kleinbuchstaben umgewandelt. Der Eingabestring kann maximal eine Bildschirmzeile lang sein und darf maximal zehn Wörter enthalten, weitere Wörter werden ignoriert. Diese Zahl können Sie mit der in tstring.pas deklarierten Konstante maxbuf erhöhen. Die Satzzeichen ? . und ! können eingegeben werden, werden vom Parser aber ignoriert. Das Komma und das doppelte Anführungszeichen (") haben eine besondere Funktion beim Umgang mit NPCs (s. 8.1) und sollten nur in diesem Zusammenhang verwendet werden. Umlaute können wahlweise auch ausgeschrieben werden.

Hat der Spieler sich in der letzten Eingabe mit einem Substantiv auf ein Objekt bezogen, kann er sich mit einem Reflexivpronomen erneut auf das selbe Objekt beziehen: z.B. "lege das Schwert weg" und "nimm es" statt "nimm das Schwert".

Für die Bewegung durch die Spielwelt stehen dem Spieler folgende Kommandos zur Verfügung: n, norden, no, nordosten, o, osten, so, südosten, s, süden, sw, südwesten, w, westen, nw, nordwesten. Nach oben und unten gelangt der Spieler mit oben, aufwärts, rauf, hoch, unten, abwärts und runter.

Der Parser versteht die Numerale von zwei bis zehn. Mit ihnen können Mengen gleichartiger Dinge angegeben werden. Wenn in einem Raum z.B. fünf Münzen vorhanden sind, kann der

Spieler folgende Eingaben machen: "nimm eine Münze", "nimm drei Münzen" oder "nimm alle Münzen". Statt eines Numerals kann der Spieler auch die entsprechende Ziffer eingeben.

Für den Parser bislang unverständlich sind Bindewörter (z.B. "und", "oder") und Verneinungen mit "nicht" und "kein".

Nach der Eingabe wird der Eingabestring an die Methode

```
TGame.BeforeParsing(VAR input : STRING); VIRTUAL;
```

übergeben. Diese Methode macht normalerweise nichts mit dem String, kann aber überschrieben werden. Wenn der Spieler in ihrem Spiel z.B. von einem Fluch getroffen wird und seinen Orientierungssinn verliert, könnten Sie in dieser Methode die Himmelsrichtungen im Eingabestring vertauschen.

7.2 Verben und Grammatik

Der Eingabestring wird von der Methode *TGame.Parse()* analysiert, die versucht, das zu der Eingabe passende Ereignis zu finden. Die einzelnen Ereignisse werden von verschiedenen Verben ausgelöst. Der Konstruktor von *TGame* ruft die Methode *TGame.ReadLib* auf, die eine Liste mit Verben, das sog. Wörterbuch, anlegt. Zusätzlich zu jedem Verb werden im Wörterbuch auch eine zugehörige Grammatik sowie das auszulösende Ereignis gespeichert. *Parse()* vergleicht nun den Eingabestring mit jedem Eintrag im Wörterbuch. Wenn in der Eingabe ein Verb aus dem Wörterbuch gefunden wird und die Eingabe auch mit der zum Verb gehörenden Grammatik übereinstimmt, wird das entsprechende Ereignis bei den beteiligten Objekten ausgelöst.

Das Wörterbuch wird in *Readlib* durch wiederholten Aufruf der Methode

```
TGame.AddVerb(_verb,_syntax : STRING; _event : BYTE)
```

angelegt. Ihre Parameter haben folgende Bedeutung:

- **_verb:String** - ein beliebiges Wort, das ein Ereignis auslösen soll. Sie können auch mehrere synonyme Wörter angeben, die durch ein Semikolon getrennt werden müssen. Wenn das Wort ein Metaverb oder ein Debugkommando sein soll, so muß ihm ein Filezeichen (#) bzw. ein Punkt vorangestellt werden.

- **_syntax:String** - die zum Verb gehörende Grammatik, mit der die Eingabe übereinstimmen muß, um ein Ereignis auszulösen. Es können auch mehrere durch ein Semikolon getrennte Grammatiken angegeben werden. Wenn Sie nur einen Leerstring angeben, genügt die Eingabe des Verbs, um ein Ereignis auszulösen.

- **_event:Byte** - die Konstante eines Ereignisses. Geben Sie hier eine der in tmain.pas deklarierten Konstanten mit der Endung *_ev an.

Was hat es nun mit der Grammatik auf sich? Die Grammatik eines Verbs gibt an, welche Wörter dem Verb im Eingabestring folgen müssen. Sie können sich die Grammatik als eine Schablone vorstellen, in die die Eingabe hineinpassen muß. Eine Grammatik besteht aus einzelnen Token, von denen jedes für ein einzelnes Wort oder eine Wortgruppe steht. Die

Token werden in der Grammatik in der Reihenfolge angeordnet, in der die entsprechenden Wörter auch in der Eingabe auftauchen sollen, dabei müssen die einzelnen Token durch ein Pluszeichen voneinander getrennt werden. Textopia kennt sieben verschiedene Token:

- **[wort]** - ein beliebiges Wort, das nicht Bestandteil eines Objektnamens ist, meistens eine Präposition. Das Wort muß nur in der Grammatik in eckigen Klammern gefaßt sein. Wenn der Parser in der Eingabe auf ein Wort stößt, das weder in einem Objektnamen noch in einer Grammatik auftaucht, gibt er die Meldung "Ich verstehe das Wort 'xxx' nicht" aus.

- **held** - ein Substantiv, das sich auf ein Objekt im Inventar des Spielers bezieht. Der Scope (s. 5.7) dieses Objekts muß also dem Wert held_sd entsprechen.

- **noun** - ein Substantiv, das sich auf irgendein beliebiges Objekt in der Spielwelt bezieht.

- **npc** - ein Substantiv, das sich auf einen NPC bezieht, der sich mit dem Spieler in einem Raum befindet.

- **number** - ein Numeral zwischen zwei und zehn oder eine Nummer zwischen 0 und 32767. Beachten Sie, daß für Mengenangaben nur Zahlen bis 10 erlaubt sind. Größere Zahlen können natürlich als Telefonnummern oder Geheimcodes verwendet werden.

- **reachable** - ein Substantiv, das sich auf ein Objekt mit dem Scope reachable_sd bezieht.

- **routine** - ein Substantiv, das sich auf ein Objekt bezieht, für das die zu überschreibende Methode *TGame.MyScope(_id:Word;_action:Byte):Boolean* den Wert *True* liefert. In dieser Methode können Sie bestimmen, ob das Objekt mit der ID _id das Ereignis _action ausführen kann.

Eines der Verben, mit dem der Spieler sich auf das Pferd setzen kann, ist z.B. wie folgt definiert:

```
AddVerb('steige;steig','[auf]+reachable;[aufs]+reachable',enter_ev);
```

Wenn Sie sich *ReadLib* anschauen, werden Sie sehen, daß unterschiedliche Verben gleiche Grammatiken haben und auch das gleiche Ereignis auslösen können. Umgekehrt können gleiche Verben auch zu unterschiedlichen Ereignissen führen, wenn ihre Grammatiken sich unterscheiden.

Einige Verben benötigen zwei Substantive (oder können wahlweise mit zwei Substantiven verwendet werden). Um ein Objekt in einen Behälter zu legen, kann der Spieler z.B. "lege das Schwert in den Rucksack" eingeben. Die Definition des entsprechenden Verbs sieht wie folgt aus:

```
AddVerb('lege;wirf;werfe;schmei\s','held+[in]+reachable',drop_ev);
```

Wenn eine Grammatik zwei Substantive enthält, wird das Ereignis grundsätzlich von dem zum ersten Substantiv gehörenden Objekt ausgeführt. Die Eingabe "lege x in y" führt also immer dazu, daß die Methode *HandleEvent()* des Objekts x das Ereignis drop_ev ausführt und das Objekt in den Behälter y legt. In einigen Fällen können die Substantive x und y jedoch vertauscht werden, ohne daß sich die Bedeutung der Eingabe verändert. Ein Beispiel dafür sind die gleichbedeutenden Eingaben "verschließe die Truhe mit dem Schlüssel" und "verschließe mit dem Schlüssel die Truhe". In diesen Fällen müssen zwei Grammatiken, jeweils eine für eine der beiden möglichen Reihenfolgen, angegeben werden. Wenn in einer Grammatik das zweite Objekt das Ereignis ausführen soll, so muß dem entsprechenden Token das Klammeraffenzeichen (@) vorangestellt werden:

```
AddVerb('verschließe','[mit]+held+@reachable',lock_ev);
AddVerb('verschließe','reachable+[mit]+@held',lock_ev);
```

Bevor Sie in Ihrem Programm *TGame.Run* aufrufen, können Sie dem Wörterbuch mit *AddVerb()* eigene Verben, Grammatiken und Ereignisse hinzufügen. Ein ganz neues Wörterbuch können Sie anlegen, wenn Sie vor *AddVerb()* die von *ReadLib* angelegte Liste mit der Methode

```
TGame.ClearDictionary
```

löschen. Alle Verben eines bestimmten Ereignisses können Sie mit der Methode

```
TGame.DelVerb(_event : BYTE)
```

löschen.

In *Readlib* werden Sie vielleicht die Himmelsrichtungen für die Bewegung (go_ev) vermissen. Diese werden von *Parse()* unabhängig von der Verbenliste verarbeitet, da es mit dem System des Wörterbuchs nicht möglich ist, zu bestimmen, in welche Richtung der Spieler sich bei dem Ereignis go_ev bewegt.

7.3 TEvent und TNoun

Wenn der Parser eine Eingabe erkannt hat, übergibt er der Methode *HandleEvent()* eine Recordvariable vom Typ *TEvent* (s. 6.1). Dieser Record enthält folgende Informationen:

- **action:Byte** - der Wert einer Ereigniskonstante. *HandleEvent()* kann hiermit feststellen, welches Ereignis ausgeführt werden soll.

- **exec:Byte** - wenn das Ereignis von mehreren gleichartigen Objekten ausgeführt wird (z.B. "nimm drei Münzen"), gibt dieses Feld an, von wievielen Objekten es bereits ausgeführt wurde.

- **maxexec:Byte** - bestimmt, von wievielen Objekten das Ereignis maximal ausgeführt wird.

- **return:Boolean** - bestätigt die erfolgreiche Ausführung eines Ereignisses. Diesem Feld wird bei der Ausführung eines Ereignisses in *HandleEvent()* der Wert *True* zugewiesen. Wenn ein Ereignis von *BeforeAction()* unterbrochen wird, genügt eine Textausgabe, um die Variable replay (s. 6.2) auf *True* zu setzen und dem Parser eine erfolgreiche Ausführung zu signalisieren.

- **who:PItem** - ist immer gleich *NIL*, wenn das Ereignis vom Spieler ausgelöst wurde. Ansonsten zeigt who auf das als NPC dienende Objekt (s. 8.1).

- **first:PNoun** - ein Zeiger auf einen Record vom Typ *TNoun*, dieser enthält weitere Informationen über das erste Substantiv der Eingabe.

- **second:PNoun** - ein Zeiger auf einen Record vom Typ *TNoun*, dieser enthält weitere Informationen über das zweite Substantiv der Eingabe.

- **data:Pointer** - einen Zeiger auf weitere Daten, die für die Ausführung des Ereignisses benötigt werden. Dieser Zeiger wird nur bei den Ereignissen go_ev und tell_ev (s. 8.1) benötigt, ansonsten ist er gleich *NIL*.

Die Methode *TGame.SearchNoun()* wird von *Parse()* aufgerufen und sucht in der Eingabe nach einem Objektnamen. In ihr werden die Recordvariablen vom Typ *TNoun* ausgefüllt, auf die first und second zeigen. *TNoun* enthält folgende Informationen über die in der Eingabe vorkommenden Substantive:

- **detect:Boolean** - das Substantiv konnte einem Objekt zugeordnet werden, wenn dieses Feld *True* ist, ansonsten wird kein Ereignis ausgeführt.

- **adj:Boolean** - das Substantiv enthält ein Adjektiv, wenn das Feld *True* ist.

- **number:Integer** - eine vom Spieler angegebene Menge.

- **xroom:PRoom** - ein Zeiger auf ein zum Substantiv gehörendes Objekt vom Typ *TRoom*.

- **xlink:PLink** - ein Zeiger auf ein zum Substantiv gehörendes Objekt vom Typ *TLink*.

- **xitem:PItem** - ein Zeiger auf ein zum Substantiv gehörendes Objekt vom Typ *TItem*.

Von den Zeigern xroom, xlink und xitem ist immer nur einer ungleich *NIL*.

7.4 Fehlermeldungen

Wenn bei der Analyse der Eingabe oder der Ausführung eines Ereignisses ein Problem auftritt, wird die Methode

```
TGame.ParserError(n : BYTE; str : STRING; i : BYTE; VAR error : BOOLEAN);
```

aufgerufen. Mit einer von 37 möglichen Meldungen, informiert diese Methode den Spieler, daß seine Eingabe nicht ausgeführt werden kann. Mögliche Gründe für den Aufruf von *ParserError()* sind z.B. unbekannte Wörter in der Eingabe, der Versuch des Spielers, mit einem nicht erreichbaren Objekt zu hantieren oder eine bereits abgeschlossene Tür erneut zu verschließen. Die einzelnen Parameter haben folgende Bedeutung:

- **n:Byte** - ein Fehlercode zwischen 1 und 37.

- **str:String** - ein Wort aus der Eingabe oder der Name des Objekts, mit dem sich ein Ereignis nicht ausführen läßt. Die Bedeutung dieses Strings hängt vom jeweiligen Fehlercode ab.

- **i:Byte** - Anzahl der vom fehlgeschlagenen Ereignis betroffenen Objekte. Der entsprechende Objektname wird abhängig von dieser Variable im Singular oder Plural ausgegeben.

- **error:Boolean** - ist ein Var-Wert und wird von *ParserError()* auf *True* gesetzt. Der Aufruf von *ParserError()* setzt in *Parse()* und *HandleEvent()* eine lokale Error-Variable auf *True*, was die weitere Analyse und Ausführung verhindert.

Wenn Sie eine der Meldungen von *ParserError()* ändern oder weitere Meldungen hinzufügen möchten, müssen Sie diese Methode überschreiben, die gewünschte Meldung abfangen und für andere Fehlercodes wieder *TGame.ParserError()* aufrufen. Wenn Sie von *TGame* den Objekttyp *TnGame* abgeleitet haben, kann das so aussehen:

```
PROCEDURE TnGame.ParserError(n : BYTE; str : STRING; i : BYTE;
                             VAR error : BOOLEAN);
BEGIN
  CASE n OF
    5 : Print('Ihre Meldung\n');
    ELSE TGame.ParserError(n : BYTE; str : STRING; i : BYTE;
                           VAR error : BOOLEAN);
  END;
END;
```

7.5 Metaverben

Metaverben beeinflussen einige von der Spielhandlung unabhängige Optionen und verbrauchen keine Spielzeit. Sie beziehen sich nicht auf Objekte in der Spielwelt. Daher werden sie nicht von einer *HandleEvent*-Methode ausgeführt, sondern führen zum Aufruf verschiedener Methoden von *TGame*. Bevor der Parser ein Metaverb ausführt, wird die Methode

```
TGame.MetaVerb(mv : BYTE) : BOOLEAN; VIRTUAL;
```

mit dem Wert des entsprechenden Ereignisses aufgerufen, deren Rückgabewert über die weitere Bearbeitung des Ereignisses entscheidet. Wenn Sie *MetaVerb()* überschreiben, können Sie also die Ausführung von Metaverben beeinflussen. Um eine Verwechslung mit anderen Verben zu verhindern, muß den Metaverben das Filezeichen (#) vorangestellt werden. Folgende Metaverben stehen zur Verfügung:

- **#beschreibung** - wechselt zwischen ausführlichen und kurzen Beschreibungen bereits bekannter Räumen, ändert also die bei der Initialisierung des Spiels getroffene Voreinstellung. Kann mit #b abgekürzt werden.

- **#ende** - beendet das Spiel nach einer Sicherheitsabfrage.

- **#inventar** - zeigt das Inventar des Spielers an. Im Debugmodus werden noch die Anzahl der Objekte im Inventar und das Gesamtgewicht des Spielers angezeigt. Kann mit #inv und #i abgekürzt werden.

- **#karte** - zeigt eine Karte der bereits bekannten Räume an. Im Debugmodus werden alle Räume angezeigt, egal ob sie bereits besucht wurden. Der aktuelle Raum erscheint dabei immer in der Mitte der Karte, deren Größe von der Zahl der Bildschirmzeilen und Spalten begrenzt wird.

- **#laden** - lädt einen zuvor gespeicherten Spielstand. Wurden bereits mehrere Spielstände gespeichert, wird ein Menü mit den gespeicherten Spielständen angezeigt, von denen einer ausgewählt werden kann.

- **#neu** - stellt den Ausgangszustand des Spiels wieder her.

- **#punkte** - zeigt Punkte- und Zugzahl des Spielers an. Die hierbei aufgerufene Methode *TPlayer.RankStr* kann überschrieben werden, um zusätzlich zu der Punktzahl einen sich hieraus ergebenden Status des Spielers anzuzeigen.

- **#status** - schaltet die Statuszeile ein und aus.

- **#sichern** - speichert einen Spielstand. Nach Eingabe dieses Kommandos wird der Spieler nach einem Namen für den Spielstand gefragt.

7.6 Debugkommandos

Vier Debugkommandos sollen Ihnen helfen, Überblick über die Objekte der Spielwelt zu behalten. Debugkommandos beeinflussen wie die Metaverben nicht das Spielgeschehen, beziehen sich aber auf einzelne Objekte und werden daher von einer *HandleEvent*-Methode ausgeführt. Debugkommandos müssen mit einem Punkt beginnen und werden vom Parser nur im Debugmodus (s. 4.6) verstanden. Folgende Debugkommandos stehen zur Verfügung:

- **.browse** - zeigt die Eigenschaften (Namen, Attribute, Position etc.) einzelner Objekte an. Der Index entspricht der Position des Objekts in der internen Objektliste. Die Eigenschaften eines Objekts werden von der Methode *View* ausgegeben, die jeder Nachfolger von *TBasic* besitzt. Mit den Tasten CursorOben/CursorUnten können Sie alle Objekte anzeigen lassen, mit [Esc] beenden Sie die Ausgabe.

- **.dec** - gibt alle Deklinationen des angegebenen Objektnamens aus. Sie können so prüfen, ob Sie die Sonderzeichen für die Deklination (s. 4.4) richtig angegeben haben.

- **.list** - zeigt die gleichen Informationen wie browse, jedoch nur für das angegebene Objekt.

- **.trace** - schaltet die Anzeige des Namens und des Scopes eines Objekts an oder aus. Diese Anzeige erfolgt nach jeder Eingabe, so daß die Veränderung des Scopes im Spiel beobachtet werden kann. Wenn Sie die Methode *Inspect*, die alle Nachfahren von *TBasic* besitzen, überschreiben, können Sie auch die Veränderung anderer Eigenschaften eines Objekts verfolgen.

8 Verschiedenes

8.1 Nichtspielercharaktere

Kehren wir noch einmal zu unserem Mini-Adventure zurück. Damit es dem Spieler in dem Zwergengrab nicht zu langweilig wird, soll ihm ein Troll Gesellschaft leisten. Der Troll soll ein Nichtspielercharakter (kurz NPC) sein. Der Spieler kann mit einem NPC reden und ihm Anweisungen geben, die der NPC ausführt. NPCs werden von Objekten des Typs *TItem* dargestellt. Um unserem Adventure einen Troll hinzuzufügen, muß demo5.pas also zunächst um die Zeilen

```
troll_id = 26;
```

```
NEW(item,Init(troll_id,'+Troll#e',Adr(cave_id),t,1,5));
```

erweitert werden. Jedes Objekt vom Typ *TItem* ist entweder unbelebt, lebendig oder tot. Als NPCs dienende Objekte sind lebendig oder tot, alle anderen Objekte vom Typ *TItem* sind immer unbelebt. Diesen Zuständen entsprechen die Ausprägungen des in tmain.pas deklarierten Variablentyps *TMatter*: *inanimate*, *dead* und *alive*. Jedes Objekt wird automatisch mit dem Zustand *inanimate* initialisiert. Mit der Methode

```
TItem.SetMatter(state : TMatter)
```

läßt sich der Zustand eines Objekts ändern, mit

```
TItem.GetMatter : TMatter
```

läßt er sich abfragen. Ein "lebendes" Objekt kann nicht automatisch angesprochen werden. Damit der Spieler den Troll auch anreden kann, muß noch das Attribut talkable_at gesetzt werden. Folgende Zeilen machen aus dem eben erzeugten Objekt einen NPC:

```
WITH item^ DO
BEGIN
  SetMatter(alive);
  SetAttrib(talkable_at);
  AddText('Berichte über Trolle mit ihren langen, scharfen Fangzähnen '+
          'und Klauen haben Sie bisher nur in Tavernen gehört und nie '+
          'ernst genommen. Jetzt sollten Sie Ihre Meinung ändern.\n');
END;
```

Um einen NPC anzusprechen, muß der Spieler dessen Namen gefolgt von einem Komma, einem Leerzeichen und einem Text in doppelten Anführungszeichen eingeben. Starten Sie jetzt einmal demo6.pas und gehen in die Höhle:

HÖHLE
Sie befinden sich in der kleinen Höhle im Südwesten der Grabkammer. Diese Höhle wurde offensichtlich nicht von den Zwergen angelegt, als sie das Grab schufen. Die Wand der Grabkammer wurde für die Höhle durchbrochen und die Wände der Höhle sind mit scheußlichen Symbolen übersät, die sich stark von den Schriftzeichen der Zwerge auf dem Sarkophag unterscheiden.
Ein Schlüssel.
Ein Troll.

>troll, "wer sind sie?"

Der Troll hört Ihnen nicht zu.

Die Meldung "x hört Ihnen nicht zu" ist die Standardantwort auf jede Anrede. Um eine sinnvollere Reaktion des NPC hervorzurufen, müssen Sie die Methode *TItem.BeforeAction()* überschreiben und das Ereignis tell_ev abfangen. Den Objekttyp *TnItem* haben wir bereits von *TItem* abgeleitet, so daß wir seiner Deklaration nur noch die Zeile

```
FUNCTION BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
```

hinzufügen müssen, um folgende Reaktion des Trolls auf eine Anrede bestimmen zu können:

```
FUNCTION TnItem.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    CASE action OF
      tell_ev : BEGIN
                  ok:=f;
                  IF POS('zwerg',STRING(data^))>0 THEN
                     Print('Der Troll sagt: "Die Zwergenknochen sind mir '+
                           'ausgegangen, du kommst genau richtig..."\n')
                     ELSE Print('Der Troll leckt nur seine Zähne und '+
                                'schaut Sie mit einem sehr '+
                                'beunruhigenden Blick an.\n');
                END;
    END;
  END;
  BeforeAction:=ok;
END;
```

Um den String mit der wörtlichen Rede des Spielers zu erreichen, müssen Sie die Pointer-Variable data des Records event in einen String umwandeln. In unserem Fall hängt die Antwort des Trolls davon ab, ob in der Anrede des Spielers der Teilstring "zwerg" auftaucht. Diese Typumwandlung funktioniert nur bei dem Ereignis tell_ev.

Beachten Sie, daß wenn in Ihrem Spiel mehrere NPCs auftauchen, Sie erst mit der Methode *GetID* feststellen müssen, welcher NPC angesprochen wurde.

Bevor Sie den Spieler mit seinem Schwert den Troll meucheln lassen, schauen Sie sich noch an, wie NPCs Befehle ausführen können. Um einem NPC ein Kommando zu geben, muß der Spieler nur den Namen des NPC gefolgt von einem Komma, einem Leerzeichen und der gewünschten Anweisung eingeben:

HÖHLE
Sie befinden sich in der kleinen Höhle im Südwesten der Grabkammer. Diese
Höhle wurde offensichtlich nicht von den Zwergen angelegt, als sie das Grab
schufen. Die Wand der Grabkammer wurde für die Höhle durchbrochen und die
Wände der Höhle sind mit scheußlichen Symbolen übersät, die sich stark von
den Schriftzeichen der Zwerge auf dem Sarkophag unterscheiden.
Ein Troll.
Ein Schlüssel.

>troll, nimm den schlüssel
Der Troll nimmt einen Schlüssel.

>troll, gib mir den schlüssel
Der Troll reicht Ihnen einen Schlüssel.

>gib den schlüssel dem troll
Sie geben dem Troll einen Schlüssel.

>troll, lege den schlüssel weg
Der Troll legt einen Schlüssel weg.

Der Variable who vom Typ *PItem* im Record event können Sie entnehmen, ob ein Ereignis vom Spieler oder von einem NPC ausgelöst wurde. Wurde ein Ereignis vom Spieler ausgelöst, ist dieser Zeiger gleich *NIL*, ansonsten zeigt er auf den handelnden NPC. Wenn der Troll für die Anweisungen des Spielers taub sein soll, müssen Sie die obige Methode wie folgt erweitern:

```
FUNCTION TnItem.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    IF who<>NIL THEN
    BEGIN
      ok:=f;
      Print('Der Troll denkt nicht daran, Ihre Befehle auszuführen.\n');
    END;
    CASE action OF
      ...
    END;
  END;
  BeforeAction:=ok;
END;
```

Der Versuch, den Troll zu erschlagen, führt zu der wenig befriedigenden Meldung "Sie sind ein friedliebender Spieler" (bzw. "Spielerin", dies ist die einzige Stelle, an der Textopia das Spielergeschlecht berücksichtigt). Das ist natürlich nur eine Standardantwort, die Sie in *BeforeAction()* ändern können. Um es dem Spieler nicht zu einfach zu machen, tötet nur jeder zweite Schwertstreich den Troll:

```
FUNCTION TnItem.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    IF who<>NIL THEN
    BEGIN
      ...
    END;
    CASE action OF
      tell_ev : BEGIN
                  ...
                END;
      kill_ev : BEGIN
                  ok:=f;
                  IF RANDOM(2)=0 THEN
                  BEGIN
                    SetMatter(dead);
                    Print('Mit einem schrecklichen Schrei stirbt der '+
                          'Troll, als Ihr Schwert ihn trifft.\n');
```

```
                    player^.IncScores(5);
                  END ELSE Print('Sie haben noch nie ein Schwert in der '+
                                 'Hand gehabt, stimmts?\n');
                END;
    END;
  END;
  BeforeAction:=ok;
END;
```

Der Spieler kann jetzt den Troll töten, einen Schlüssel finden und die Schatzkammer ausräumen (wenn Sie da Schätze reinlegen). Für die Bewältigung bestimmter Aufgaben können Sie dem Spieler Punkte geben. Benutzen Sie hierfür die Methode

```
TPlayer.IncScores(ds : WORD)
```

Mit der Methode

```
TPlayer.GetScores : WORD
```

können Sie die aktuelle Punktzahl abfragen. Das Ereignis kill_ev können Sie
mit dem Programm demo7.pas ausprobieren.

8.2 Leben, Tod und Sieg des Spielers

In dem Mini-Adventure können Sie dem Troll auch die Chance geben, den Spieler zu töten. Wenn es Ihnen gefällt, können Sie das Alter-Ego des Spielers natürlich auch auf andere Weise umkommen lassen. Der Spieler verfügt über einen Status, der die Werte der Konstanten alive_ps, dead_ps und victory_ps annehmen kann. Bei Spielbeginn bekommt der Spielerstatus automatisch den Wert alive_ps zugeordnet. Der Dialog zwischen Spieler und Parser endet, wenn der Spielerstatus den Wert dead_ps oder victory_ps annimmt. Den Spielerstatus können Sie mit der Methode

```
TPlayer.SetState(s : BYTE)
```

ändern. Töten Sie den Spieler mit *SetState(dead_ps),* so wird die Methode

```
TPlayer.AfterLife
```

aufgerufen, die normalerweise die erreichte Punktzahl ausgibt. Hat der Spieler hingegen das finale Rätsel gelöst oder eine bestimmte Punktzahl erreicht, können Sie das Spiel mit *SetState(victory_ps)* beenden. Hiernach wird die Methode

TPlayer.Victory

aufgerufen, die normalerweise ebenfalls die erreichte Punktzahl ausgibt. Beide Methoden können Sie überschreiben, um einen Nachruf oder Glückwünsche auszugeben. Nach Beendigung dieser Methoden kann der Spieler das Spiel erneut beginnen, einen gespeicherten Spielstand laden oder das Programm ganz beenden.

8.3 Hintergrundprozesse, Zeitschalter und Spielzeit

Ein Hintergrundprozeß, auch Dämon genannt, ist eine Methode mit beliebigem Programmcode, die jedem Objekt vom Typ *TRoom*, *TItem* und *TLink* zugeordnet werden kann und nach jeder Spielrunde einmal ausgeführt wird. Ein Hintergrundprozeß wird mit der Methode

```
StartDaemon
```

gestartet. Die Methode

```
StopDaemon
```

schaltet ihn wieder aus. Ist ein solcher Dämon einmal aktiv, wird nach jeder Spielrunde die Methode

```
RunDaemon; VIRTUAL
```

aufgerufen, die Sie überschreiben müssen, damit sie etwas tut. In dieser Methode können Sie z.B. NPCs von Raum zu Raum bewegen (s. 5.1) oder überprüfen, ob eine bestimmte Spielsituation eingetreten ist, auf die irgendein Objekt reagieren soll.

Im Gegensatz zu Hintergrundprozessen führen Zeitschalter nicht nach jeder Spielrunde zur Ausführung einer Routine, sondern erst nach Ablauf einer bestimmten Anzahl von Runden. Gestartet wird ein Zeitschalter (oder auch Timer) mit der Methode

```
StartTimer(rounds : BYTE)
```

Ihr müssen Sie die Zahl der Runden übergeben, nach deren Ablauf die Methode

```
TimeOut; VIRTUAL
```

aufgerufen wird. *TimeOut* müssen Sie überschreiben, um zu bestimmen, was nach der angegeben Rundenzahl geschehen soll. Einen Timer können Sie mit *StopTimer* auch vorzeitig abschalten (wenn es dem Spieler z.B. gelingt, eine Zeitbombe zu entschärfen). Zeitschalter und Dämon eines Objekts dürfen auch gleichzeitig aktiv sein, beachten Sie dann, daß *RunDaemon* immer vor *TimeOut* ausgeführt wird.

Textopia verfolgt auch die Spielzeit (die nichts mit der Systemzeit Ihres Computers zu tun hat), die Ihre Dämonen und Zeitschalter auch berücksichtigen können. Die Spielzeit wird mit einer 24 Stunden Uhr ab Spielbeginn gemessen und kann mit der Methode

```
TGame.GetTime(VAR hour,min : BYTE)
```

ermittelt werden. Die Spielzeit wird zu Spielbeginn auf 0:00 Uhr gesetzt. Die Uhrzeit können Sie zu Spielbeginn oder irgendwann im Spiel mit der Methode

```
TGame.SetTime(_time,_rate : WORD)
```

ändern. Der Parameter _time gibt die Uhrzeit als Zahl der seit Mitternacht vergangenen Minuten an und berechnet sich nach der Formel 60*Stunden+Minuten. Der Parameter _rate gibt an, wieviele Minuten während einer Runde vergehen.

Mit einem Dämon können wir das Mini-Adventure noch mal erweitern. Bislang steht der Troll nur regungslos in der Höhle herum. Jetzt soll er die Initiative ergreifen, sobald der Spieler die Höhle betritt. Der Spieler wird in der Höhle vom Troll angegriffen und getötet, wenn er flüchten will. Erschlägt der Spieler den Troll, so findet er den Schlüssel, der dem toten Troll aus dem Wams fällt. Zunächst muß statt der statischen Beschreibung der Höhle eine variable Beschreibung in *TnRoom.MyText* erfolgen. Hier wird festgestellt, ob der Troll noch lebt. Ist er tot, wird die normale Beschreibung der Höhle ausgegeben. Lebt der Troll noch, so wird dem Spieler der Angriff mitgeteilt und der Dämon des Trolls gestartet, um den Troll nun jede Runde den Spieler angreifen zu lassen. In *TnRoom.BeforeAction()* wird der Spieler an der Flucht gehindert und getötet. Wenn der Spieler den Troll erschlägt, was nur mit dem Schwert möglich ist, wird der Dämon des Trolls abgeschaltet und der Schlüssel in die Höhle gelegt. Der Deklaration von *TnItem* wird die Zeile

```
PROCEDURE RunDaemon; VIRTUAL;
```

hinzugefügt. Definiert wird die Methode wie folgt:

```
PROCEDURE TnItem.RunDaemon;
BEGIN
  CASE RANDOM(3) OF
    0 : Print('Die furchtbaren Klauen des Trolls verfehlen Sie um '+
              'Haaresbreite.\n');
    1 : Print('Der Troll schlägt Ihnen beinahe das Schwert aus der '+
              'Hand.\n');
    2 : Print('Sie können grade noch dem Schlag des Trolls ausweichen.\n');
  END;
END;
```

Echten Schaden fügt der Troll dem Spieler also nur zu, wenn dieser zu flüchten versucht. Auf eine Prüfung, ob der Spieler tatsächlich ein Schwert hält, soll bei der zweiten Ausgabe einmal verzichtet werden. Beachten Sie, daß Sie in *RunDaemon* mit *GetID* feststellen müssen, welcher Dämon grade ausgeführt wird, wenn mehrere Hintergrundprozesse aktiv sind. Eingeschaltet wird der Dämon des Trolls in *TnRoom.MyText*, wenn der Spieler die Höhle zum ersten Mal betritt:

```
PROCEDURE TnRoom.MyText;
BEGIN
  CASE GetID OF
    hill_id : BEGIN
                ...
              END;
    cave_id : IF PItem(Adr(troll_id))^.GetMatter=alive THEN
              BEGIN
                Print('Als Sie die Höhle betreten, springt Ihnen mit '+
                      'markerschütterndem Gebrüll ein Troll entgegen!\n');
                PItem(Adr(troll_id))^.StartDaemon;
              END ELSE BEGIN
                         Print('Sie befinden sich in der kleinen Höhle '+
                               'im Südwesten der Grabkammer. Diese Höhle '+
                               'wurde offensichtlich nicht von den '+
                               'Zwergen angelegt, als sie das Grab '+
                               'schufen. Die Wand der Grabkammer wurde '+
                               'für die Höhle durchbrochen und die ');
                         Print('Wände der Höhle sind mit '+
                               'scheußlichen Symbolen übersät, die sich '+
                               'stark von den Schriftzeichen der Zwerge '+
                               'auf dem Sarkophag unterscheiden.\n');
                       END;
  END;
```

```
END;
```

Bei der Initialisierung des Troll wurde der Parameter show auf *False* gesetzt, damit der Troll nach dem ersten Angriff nicht noch mal erwähnt wird. Erst wenn er stirbt, wird seine Leiche mit *ListMe(True)* wieder in die automatische Raumbeschreibung aufgenommen. Wenn der Dämon nun aktiv ist, prüfen wir, ob der Spieler die Höhle verlassen will und der Troll noch lebt:

```
FUNCTION TnRoom.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    CASE action OF
      go_ev : IF ToDir=down THEN
              BEGIN
                ...
              END ELSE IF (ToDir=northeast) AND
                          (PItem(Adr(troll_id))^.GetMatter=alive) THEN
                          BEGIN
                            Print('Sie spüren kurz eine Klaue im Nacken, '+
                                  'dann wird es um Sie dunkel...\n');
                            player^.SetState(dead_ps);
                          END;
    END;
  END;
  BeforeAction:=ok;
END;
```

Schließlich soll der Spieler den Troll mit dem Schwert töten, dessen Dämon damit ausschalten und einen Schlüssel finden:

```
FUNCTION TnItem.BeforeAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    IF who<>NIL THEN
    BEGIN
      ...
    END;
    CASE action OF
      tell_ev : BEGIN
                  ...
                END;
      kill_ev : BEGIN
                  ok:=f;
                  IF PItem(Adr(sword_id))^.GetLocation=NIL THEN
                  BEGIN
                    IF RANDOM(2)=0 THEN
                    BEGIN
                      SetMatter(dead);
                      ListMe(t);
                      StopDaemon;
                      Print('Mit einem schrecklichen Schrei stirbt der '+
                            'Troll, als Ihr Schwert ihn trifft. Als er '+
                            'zu Boden stürzt, fällt ein Schlüssel aus '+
                            'seinem Wams.\n');
                            PItem(Adr(key_id))^.MoveItemTo(Adr(cave_id));
```

```
                                   player^.IncScores(5);
                           END ELSE Print('Sie haben noch nie ein Schwert in '+
                                          'der Hand gehabt, stimmts?\n');
                         END ELSE Print('Mit bloßen Händen??\n');
                      END;
        END;
      END;
      BeforeAction:=ok;
END;
```

8.4 Licht und Dunkel

Räume werden von Textopia automatisch beleuchtet, so daß der Spieler in ihnen immer sehen kann. Mit der Methode

```
TRoom.SetLight(_light : BOOLEAN)
```

können Sie die automatische Beleuchtung aus- und wieder einschalten. Mit der Methode

```
TRoom.HasLight : BOOLEAN
```

können Sie feststellen, ob der Spieler in einem Raum sehen kann. Wenn der Spieler ein Item mit dem Attribut shining_at bei sich trägt, kann er in jedem Raum sehen. Befindet der Spieler sich in einem dunklen Raum, so wird in der Statuszeile statt dem Raumnamen das Wort "Finsternis" ausgegeben und die Raumbeschreibung erfolgt über die Methode *TRoom.MyDarkness*, die Sie überschreiben müssen, um dem Spieler mitzuteilen, was bzw. was er nicht sehen kann. Bis auf das Kommando "schau" bleiben alle Aktionen des Spielers von der Dunkelheit unbeeinflußt, solange Sie das fehlende Licht nicht in irgendeiner überschriebenen Methode berücksichtigen.

In der letzten Version unseres Mini-Adventures (demo9.pas) statten wir den Spieler jetzt mit einer Fackel aus und schalten das Licht im Grab ab, wenn das Tor von innen geschlossen wird. Zunächst wird die Fackel erzeugt, dazu wird im Abschnitt Const die Zeile

```
torch_id = 27;
```

hinzugefügt. Da zum Anzünden und Löschen der Fackel zwei neue Verben und Ereignisse gebraucht werden, können hier auch gleich die neuen Ereigniskonstanten deklariert werden:

```
fireon_ev  = 140;
fireoff_ev = 141;
```

Jetzt initialisieren wir eine nicht brennende Fackel und legen Sie in den Rucksack:

```
NEW(item,Init(torch_id,'-Fackel#n',Adr(rucksack_id),t,1,0));
item^.SetAttrib(takeable_at,t);
```

Nach der Initialisierung des Spiels müssen die neuen Verben "anzünden" und "löschen" dem Wörterbuch bekanntgemacht werden:

```
AddVerb('zünde;zünd','held+[an]',fireon_ev);
AddVerb('lösch;lösche;mache;mach','held;held+[aus]',fireoff_ev);
```

Die neuen Verben werden von *TItem.HandleEvent()* nicht berücksichtigt und können daher nicht in *TnItem.BeforeAction()* abgehandelt werden. Daher überschreiben wir *HandleEvent()* für die neuen Ereignisse und geben via *Else* alle anderen Ereignisse an den Vorgänger weiter. Nachdem die Zeile

```
PROCEDURE TnItem.HandleEvent(VAR event : TEvent); VIRTUAL;
```

der Definition von *TnItem* hinzugefügt wurde, kann jetzt der Code für die
Fackel geschrieben werden:

```
PROCEDURE TnItem.HandleEvent(VAR event : TEvent);
BEGIN
  WITH event DO
  BEGIN
    CASE action OF
      fireon_ev  : IF GetID=torch_id THEN
                   BEGIN
                     IF NOT(Has(shining_at)) THEN
                     BEGIN
                       IF player^.GetLocation^.GetID
                          IN [grave_id,cave_id,chamber_id] THEN
                       BEGIN
                         PnRoom(Adr(grave_id))^.SetLight(t);
                         PnRoom(Adr(cave_id))^.SetLight(t);
                         PnRoom(Adr(chamber_id))^.SetLight(t);
                       END;
                       SetAttrib(shining_at,t);
                       return:=t;
                     END ELSE Print('Die Fackel brennt bereits.\n');
                   END ELSE Print('Das geht nicht.\n');
      fireoff_ev : IF GetID=torch_id THEN
                   BEGIN
                     IF Has(shining_at) THEN
                     BEGIN
                       IF (player^.GetLocation^.GetID
                           IN [grave_id,cave_id,chamber_id]) AND
                          (PnRoom(Adr(grave_id))^.IsGate(up)=closed) THEN
                       BEGIN
                         PnRoom(Adr(grave_id))^.SetLight(f);
                         PnRoom(Adr(cave_id))^.SetLight(f);
                         PnRoom(Adr(chamber_id))^.SetLight(f);
                       END;
                       SetAttrib(shining_at,f);
                       return:=t;
                     END ELSE Print('Die Fackel ist bereits aus.\n');
                   END ELSE Print('Das geht nicht.\n');
      ELSE TItem.HandleEvent(event);
    END;
  END;
END;
```

Bevor die Fackel ein- und ausgeschaltet wird, wird ihr aktueller Status und die Position des Spielers überprüft, dieser muß sich irgendwo im Grab befinden. Wenn die Fackel gelöscht wird, wird die Beleuchtung nur abgeschaltet, wenn das Tor geschlossen ist und kein Sonnenlicht ins Grab dringt. Die auf *True* gesetzte Variable return teilt dem Parser die erfolgreiche Ausführung der neuen Ereignisse mit.

Die Ereignisse open_ev und close_ev werden von *TLock* behandelt. Hiervon wird ein Nachfolger benötigt, um die Beleuchtung im Grab ein- und auszuschalten, wenn der Spieler das Tor öffnet und schließt. Überschrieben werden müssen die Methoden *BeforeAction()* und *AfterAction()*:

```
TnLock = OBJECT(TLock)
             FUNCTION  BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
             FUNCTION  AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
           END;
PnLock = ^TnLock;
```

Entsprechend muß der generische Zeiger lock nun vom Typ *PnLock* sein:

```
lock : PnLock;
```

Das Tor wird von *TnLock* wie folgt behandelt:

```
FUNCTION TnLock.BeforeAction(event : TEvent) : BOOLEAN;
BEGIN
  WITH event DO
  BEGIN
    CASE action OF
      close_ev : IF (GetOwner=Adr(link2_id)) AND
                    (player^.GetLocation=Adr(grave_id)) AND
                    (NOT(PItem(Adr(torch_id))^.Has(shining_at))) THEN
                 BEGIN
                   PnRoom(Adr(grave_id))^.SetLight(f);
                   PnRoom(Adr(cave_id))^.SetLight(f);
                   PnRoom(Adr(chamber_id))^.SetLight(f);
                   Print('Es wird dunkel im Grab...\n');
                 END;
      open_ev  : IF (GetOwner=Adr(link2_id)) AND
                    (player^.GetLocation=Adr(grave_id)) THEN
                 BEGIN
                   PnRoom(Adr(grave_id))^.SetLight(t);
                   PnRoom(Adr(cave_id))^.SetLight(t);
                   PnRoom(Adr(chamber_id))^.SetLight(t);
                   Print('Sonnenlich strömt wieder ins Grab.\n');
                 END;
    END;
  END;
  BeforeAction:=t;
END;
```

Da Schlösser keine ID besitzen, muß hier mit *GetOwner* geprüft werden, ob es sich um das der Torverbindung zugeordnete Schloß handelt. In *BeforeAction()* wird bereits ein Text ausgegeben, wenn der Spieler den Zustand des Tors verändert, daher sollte *AfterAction()* weitere Meldungen des Parsers ("Sie öffnen/schließen das Tor") verhindern:

```
FUNCTION TnLock.AfterAction(event : TEvent) : BOOLEAN;
VAR
  ok : BOOLEAN;
BEGIN
  ok:=t;
  WITH event DO
  BEGIN
    CASE action OF
      close_ev : IF (GetOwner=Adr(link2_id)) AND
                    (player^.GetLocation=Adr(grave_id)) THEN ok:=f;
      open_ev  : IF (GetOwner=Adr(link2_id)) AND
                    (player^.GetLocation=Adr(grave_id)) THEN ok:=f;
    END;
  END;
  AfterAction:=ok;
```

```
END;
```

Schließlich muß noch *TnRoom.MyDarkness* überschrieben werden. Fügen Sie den Methodenkopf in die Definition von *TnRoom* ein und schreiben Sie etwas wie

```
PROCEDURE TnRoom.MyDarkness;
BEGIN
  Print('Sie können hier kaum etwas sehen...\n');
END;
```

Wenn Sie jetzt demo9.pas starten, sollte der folgende Dialog möglich sein:

```
WALD (auf einem Pferd)
Fast undurchdringliches Unterholz läßt Sie nur mühsam weiterkommen. Weiter
westlich scheinen die Bäume weniger dicht beieinander zu stehen.
>w

LICHTUNG (auf einem Pferd)
Ein kleiner Hügel erhebt sich in der Mitte einer Lichtung über das Grab eines
Zwergenkönigs. Ein steinernes Tor in der Nordseite versperrt Ihnen den Weg
ins Innere.

>steig vom pferd
Ok

>öffne das tor
Sie öffnen das Tor.

>gehe nach unten
GRABKAMMER
In der Mitte der engen Grabkammer steht ein Sarkophag. In der Ostwand
befindet sich eine schwere, mit Eisen beschlagene Eichentür. Im Südwesten
führt ein schmaler Gang weiter in eine hinter der Grabkammer liegende Höhle.

>öffne den rucksack
Sie öffnen den Rucksack.

>nimm die fackel aus dem rucksack
Sie nehmen eine Fackel.

>zünde die fackel an
Ok

>schließe das tor
Ok

>schau
GRABKAMMER
In der Mitte der engen Grabkammer steht ein Sarkophag. In der Ostwand
befindet sich eine schwere, mit Eisen beschlagene Eichentür. Im Südwesten
führt ein schmaler Gang weiter in eine hinter der Grabkammer liegende Höhle.

>lösche die fackel
Ok

>schau
Finsternis
Sie können hier kaum etwas sehen...

>zünde die fackel an
Ok
```

```
>sw
HÖHLE
Als Sie die Höhle betreten, springt Ihnen mit markerschütterndem Gebrüll ein
Troll entgegen!

>töte den troll mit dem schwert
Mit einem schrecklichen Schrei stirbt der Troll, als Ihr Schwert ihn trifft.
Als er zu Boden stürzt, fällt ein Schlüssel aus seinem Wams.

>nimm den schlüssel
Sie nehmen einen Schlüssel.

>no
GRABKAMMER
In der Mitte der engen Grabkammer steht ein Sarkophag. In der Ostwand
befindet sich eine schwere, mit Eisen beschlagene Eichentür. Im Südwesten
führt ein schmaler Gang weiter in eine hinter der Grabkammer liegende Höhle.

>öffne die tür
(mit dem Schlüssel)
Sie öffnen die Eichentür.

>o
SCHATZKAMMER
In dieser Kammer lagerten die Grabbeigaben des Königs.

>#ende
```

Übrigens: Taschenlampen lassen sich leichter programmieren als Fackeln, weil sie ein- und ausgeschaltet werden und daher keine neuen Verben benötigen. Ich fand aber, daß Fackeln sich besser zu Demonstrationszwecken eignen.

Das Mini-Adventure ist natürlich noch nicht perfekt. Fügen Sie ihm weitere Räume, NPCs und vor allem Aufgaben für den Spieler hinzu, die das Spiel interessanter machen.

8.5 Die Spielstandverwaltung

Mit den Metaverben #sichern und #laden kann der Spieler einen Spielstand speichern bzw. laden (s. 7.5). Wenn Sie in eine neue Objektdefinition auch neue Variablen aufnehmen, sollten Sie wissen, wie deren Werte dabei gesichert und wiederhergestellt werden.

Die Verben #sichern und #laden führen zum Aufruf der Methoden *TGame.UserSave* bzw. *TGame.UserLoad*, in der vom Spieler der Name des betreffenden Spielstands angefordert wird. Die Namen und Anzahl der gespeicherten Spielstände werden in der Datei *.sav gespeichert, was das Auslesen aus dem Plattenverzeichnis erspart. Die einzelnen Spielstände werden in untypisierten Dateien mit der Blocklänge 1 und der Endung *.ttx gespeichert, wobei das x für die Nummer eines Spielstands in der Datei *.sav steht. Wenn ein Spielstand gespeichert werden soll, legt die Methode *TGame.SaveGame()* eine Datei *.ttx an und ruft die Methode

```
Store(h : FILE); VIRTUAL
```

auf, die alle Nachfolger von *TBasic* besitzen. Mit dieser Methode speichert nun jedes Objekt die Variablen, deren Werte sich im Spielverlauf ändern können. Dabei werden alle Variablen als Byte-Wert codiert gespeichert. Ein untypisierter Dateityp wurde nur wegen eines inzwischen behobenen Bugs in FPC benutzt, eigentlich tut's auch ein *File of Byte*. Die Umwandlung der verschiedenen Variablentypen in Byte-Werte geschieht mit den Prozeduren

```
WritePtr(VAR h : File; p : PBasic);
WriteWord(VAR h : File; w : WORD);
WriteBool(VAR h : File; v : Boolean);
WriteDir(VAR h : File; d1 : TDir);
```

Bytevariablen werden natürlich direkt mit *BlockWrite()* gespeichert. Der Pointer auf ein Objekt wird einfach als dessen ID gespeichert. Für andere Datentypen müssen Sie eigene Umwandlungen programmieren. Umgekehrt kann jedes Objekt mit der Methode

```
Load(h : FILE); VIRTUAL;
```

einen gespeicherten Zustand wiederherstellen. Hierfür werden die Funktionen und Prozeduren

```
ReadPtr(VAR h : File) : POINTER;
ReadWord(VAR h : File; VAR w : WORD);
ReadBool(VAR h : File; VAR v : Boolean);
ReadDir(VAR h : File; VAR d1 : TDir);
```

verwendet. Wenn Sie in einer Objektdefinition nun eine neue Variable definieren, die sich im Spielverlauf ändern kann, so müssen Sie die Methoden *Store()* und *Load()* überschreiben, die neue Variable speichern bzw. laden und anschließend den Vorgänger des Objekttyps aufrufen, der sich um alle weiteren Variablen kümmert. Achten Sie darauf, daß die Reihenfolge der Variablen in *Store()* und *Load()* gleich bleibt, sonst führt das Kommando #laden unweigerlich zu einem Absturz.

8.6 Statuszeile, Farben und Funktionstasten

Wenn der Spieler die Statuszeile nicht mit dem Kommando #status ausgeschaltet hat, wird sie immer dann neu geschrieben, wenn sich der aktuelle Raum, die Punktzahl oder die Zahl der Spielzüge ändert. Letzteres passiert nach jeder Runde, in der der Spieler kein Metaverb eingegeben hat. Den Text in der Statuszeile liefern die Methoden

```
TPlayer.StatusBarLeftStr : STRING; VIRTUAL
```

und

```
TPlayer.StatusBarRightStr : STRING; VIRTUAL
```

Links wird normalerweise der Name des aktuellen Raums ausgegeben, während rechts die Punktzahl und die Zahl der bisherigen Spielrunden von einem Schrägstrich getrennt ausgegeben werden. Die Informationen hierüber sind in *TPlayer* gespeichert, weshalb diese Methoden nicht zu *TGame* gehören. Sie können diese Methoden überschreiben, um beliebige andere Informationen in der Statuszeile auszugeben.

Die Bildschirmfarben können Sie mit der in tio.pas definierten Prozedur

```
SetColor(fc,bc : BYTE)
```

einstellen. Dabei gibt fc die Text- und bc die Hintergrundfarbe an. Als Textfarben sind alle 16 in tio.pas deklarierten Farbkonstanten verfügbar, während für die Hintergrundfarbe nur die Farben 0 (Black) bis 7 (Lightgray) verfügbar sind. Bei der Textausgabe können Sie mit den Sonderzeichen (s. 4.4) \h und \l die Farbintensität ändern, das Zeichen \r vertauscht die Text- und Hintergrundfarben. Leider führen nicht alle Farbkombinationen unter Windows, DOS und Linux zur selben Darstellung. Wenn Sie Ihr Spiel auf ein anderes System übertragen wollen, müssen Sie hier experimentieren. Mit den voreingestellten Farben gibt es keine Probleme.

Um dem Spieler Tipparbeit abzunehmen, können Sie die Funktionstasten F1 bis F10 mit häufig benötigten Verben vorbelegen. Hierzu benutzen Sie die Prozedur

```
SetFKey(n : BYTE; str : STRING)
```

Der Parameter n gibt dabei die gewünschte Funktionstaste an.

8.7 Kommandozeilenoptionen

Der Spieler kann das fertige Programm mit einigen Kommandozeilenoptionen aufrufen, um Voreinstellungen zu verändern. Jede Option beginnt dabei mit einem Minuszeichen gefolgt von einem Buchstaben und einer Zahl (<n>) oder einem Schalter (<x>). Schalter werden mit dem Pluszeichen ein- und mit dem Minuszeichen ausgeschaltet. Die Optionen sind:

-F<n>	Bestimmt die Textfarbe
-B<n>	Bestimmt die Hintergrundfarbe
-R<n>	Bestimmt die Zahl der Bildschirmzeilen. Diese Zahl muß am Monitor bereits eingestellt sein. Textopia ändert nicht die Fenstergröße oder Bildschirmauflösung. Voreingestellt sind 25 Zeilen.
-C<n>	Bestimmt die Zahl der Zeichen pro Bildschirmzeile. Voreingestellt sind 80 Zeichen.
-S<x>	Bestimmt, ob das Spiel mit oder ohne Statuszeile gestartet wird.
-V<x>	Bestimmt, ob in bereits bekannten Räumen die ausführliche Raumbeschreibung oder nur der Raumname ausgegeben wird.
-U<x>	Bestimmt, ob die Eingabe in Großbuchstaben umgewandelt wird.

Die Option -? gibt einen kurzen Hilfetext aus. Die Berücksichtigung der Kommandozeilenoptionen können Sie mit der Zeile

```
useroption:=False;
```

verhindern.

8.8 Umlaute

DOS, Windows und Linux gebrauchen unterschiedliche Zeichensätze. Daher entsteht bei der Übertragung von Programmen zwischen den Systemen das Problem, daß die deutschen Umlaute nicht als solche ausgegeben werden.

Um dieses Problem zu umgehen, sind in tstring.pas die Char-Konstanten lae_kc, loe_kc, lue_kc und ss_kc für die kleinen und uae_kc, uoe_kc, uue_kc für die großen Umlaute definiert. Die Werte der Konstanten sind dabei vom Compiler abhängig, so daß für eine Konstante auf jedem System das richtige Zeichen ausgegeben wird.

Wenn Sie Ihr Programm nur für ein System schreiben, brauchen Sie sich um die Konstanten nicht zu kümmern und können in Ihren Texten die Umlaute direkt schreiben. Andernfalls haben Sie zwei Möglichkeiten, die Texte in Ihrem Spiel an unterschiedliche Zeichensätze anzupassen. Sie können die Umlaute wie in den Textopia-Quelltexten gegen ihre entsprechenden Konstanten austauschen. Dadurch werden die Texte allerdings recht unleserlich. In allen Texten, die mit der Prozedur *Print()* ausgegeben werden (auch in *AddText* und *AddProlog*), können Sie aber auch statt eines Umlauts einen Backlash gefolgt von einem Vokal eingeben. *Print()* tauscht diese Zeichenfolge dann automatisch gegen die Konstante des entsprechenden Umlauts aus (aus \a wird also ä).

Die Umwandlung von Umlauten kann auch das Programm umlaut.pas für Sie ausführen. Der Aufruf

```
umlaut -a dateiname
```

wandelt alle Umlaute in Sonderzeichen um. Umgekehrt macht

```
umlaut -e dateiname
```

aus Sonderzeichen wieder Umlaute.

8.9 Warnungen

Wenn Sie bei der Initialisierung der Objekte einen Fehler machen, gibt Textopia in spitzen Klammern eine Warnung aus und fragt Sie, ob Sie das Programm fortsetzen möchten. Folgende Warnungen sind möglich:

Keine Verbindung in xxx
Sie haben in *TRoom.IsGate* eine Richtung angegeben, in der keine Verbindung existiert.

Objekt xxx kann kein Schloß zugewiesen werden
Nur Objekten vom Typ *PLink* und *PItem* können Schlösser zugewiesen werden.

xxx und yyy sind bereits verbunden
Sie haben versucht, zwei Räume doppelt zu verbinden.

xxx kann nicht ein-/ausgeschaltet werden
Die Methoden *TItem.SwitchOn* und *Titem.SwitchOff* funktionieren nur, wenn das Objekt das Attribut switchable_at hat.

Objekt xxx kann nicht in yyy abgelegt werden
Mit *TItem.MoveItemTo()* können Objekte nur in Räume oder Behälter bewegt werden.

Objekt xxx kann nicht bei Objekt yyy positioniert werden
Im Konstruktor von *TItem* muß ein Raum oder ein Behälter als Position angegeben werden.

Index xxx existiert nicht

Es wurde auf einen ungültigen Index von *TObjList* zugegriffen. Wenn Sie diese Meldung erhalten, ist das meine Schuld.

Ungültiger Spielerstatus
Der Methode *TPlayer.SetState()* können nur die Konstanten dead_ps, alive_ps und victory_ps übergeben werden.

Spieler kann nicht bei Objekt xxx positioniert werden
TPlayer.MovePlayerTo() kann den Spieler nur in Räume oder Objekte mit dem Attribut enterable_at versetzen.

Spielerobjekt nicht initialisiert
Eine Instanz von *TPlayer* muß vor *TGame* initialisiert werden.

Unbekanntes Token xxx
Sie haben sich in einer mit *TGame.AddVerb()* angegebenen Grammatik verschrieben.

xxx wurde nicht bearbeitet
Ein Ereignis wurde nicht verarbeitet. Es erfolgte keine Rückmeldung durch return oder replay.

9 Die Units

9.1 Funktionen und Prozeduren von tstring.pas

Es folgt eine Liste nützlicher Stringroutinen, die Sie in Ihren Programmen verwenden können. Einige andere Routinen in tstring.pas dienen obskuren internen Zwecken und werden hier nicht aufgeführt.

Datentypen

```
CONST
  {$IFNDEF fpc}       { Umlaute unter Dos/Windows }
  lae_kc   = #132;
  loe_kc   = #148;
  lue_kc   = #129;
  ss_kc    = #225;
  uae_kc   = #142;
  uoe_kc   = #153;
  uue_kc   = #154;
  {$ELSE}             { Umlaute unter Linux }
  lae_kc   = #228;
  loe_kc   = #246;
  lue_kc   = #252;
  ss_kc    = #223;
  uae_kc   = #196;
  uoe_kc   = #214;
  uue_kc   = #220;
  {$ENDIF}
  maxbuf   = 10; { Max. Zwischenspeicher fuer Woerter }
  boolstr  : ARRAY[BOOLEAN] OF STRING = ('Nein','Ja  ');
  {
    Abkuerzungen fuer True/False
  }
  t  = TRUE;
  f  = FALSE;

TYPE
  TAdress = (du,sie,ihr);       { Anrede Spieler }
  TGender = (male,female);      { Geschlecht Spieler/in }
  TCasus  = (nom,acc,gen,dat);  { Grammatische Faelle }
  TBuffer = ARRAY[1..maxbuf] OF STRING;
```

Prozeduren und Funktionen

```
PROCEDURE DelStr(VAR str : STRING; sub : STRING);
```
Löscht alle Vorkommen von sub in str.

```
FUNCTION  LowCase(a : CHAR) : CHAR;
```
Wandelt ein Zeichen in Kleinschrift. Berücksichtigt auch Umlaute.

```
FUNCTION Lower(s1 : STRING) : STRING;
```
Wandelt einen String in Kleinschrift. Berücksichtigt auch Umlaute.

```
FUNCTION  Noun(n1 : STRING; casus : TCasus; def : BOOLEAN;
               num : BYTE) : STRING;
Dekliniert den Objeknamen n1 (s. 4.3).
```

```
FUNCTION Numeral(n : BYTE) : STRING;
```
Liefert ein Numeral für eine Zahl zwischen 2 und 10.

`FUNCTION NumToStr(n : WORD) : STRING;`
Wandelt n in einen String.

`FUNCTION PerPron(adress : TAdress; casus : TCasus) : STRING;`
Liefert das zu Anrede und Kasus passende Personalpronomen (Du, Sie, Er etc.).

`FUNCTION RandStr(line : STRING) : STRING;`
Wählt zufällig einen der durch Semikolons getrennten Teilstrings von line aus und liefert ihn zurück.

`PROCEDURE SwpStr(VAR str : STRING; oldstr,newstr : STRING);`
Tauscht in str alle Vorkommen von oldstr gegen newstr.

`FUNCTION UpChar(a : CHAR) : CHAR;`
Wandelt ein Zeichen in Großschrift. Berücksichtigt auch Umlaute.

`FUNCTION Upper(s1 : STRING) : STRING;`
Wandelt einen String in Großschrift. Berücksichtigt auch Umlaute.

9.2 Funktionen und Prozeduren von tio.pas

Die Unit tio.pas stellt verschiedenen Ein-Ausgaberoutinen zur Verfügung. Auch hier werden nur Prozeduren und Funktionen aufgeführt, die nicht ausschließlich für interne Zwecke gedacht sind.

Datentypen

```
TYPE
  PChain    = ^TChain;   { Element der String-Liste }
  TChain    = RECORD
                c    : Char;
                next : PChain;
              END;
  {$IFDEF tp}
  PDiskText = ^TDiskText; { Stringverwaltung mit Textdatei unter TP/Dos }
  TDiskText = OBJECT
                CONSTRUCTOR Init;
                FUNCTION    HasText : BOOLEAN;
                PROCEDURE   SetText(str : STRING; clear : BOOLEAN);
                PROCEDURE   PrintText;
                FUNCTION    GetText : STRING;
                DESTRUCTOR  Done;
                PRIVATE
                start       : LONGINT; { Dateiposition }
              END;
  {$ENDIF}
  PMemText  = ^TMemText;     { Stringverwaltung mit Liste }
  TMemText  = OBJECT
                CONSTRUCTOR Init;
                FUNCTION    HasText : BOOLEAN;
                PROCEDURE   SetText(str : STRING; clear : BOOLEAN);
                PROCEDURE   PrintText;
                FUNCTION    GetText : STRING;
                DESTRUCTOR  Done;
                PRIVATE
                pfirst      : PChain; { Startadresse des Textes Speicher }
                PROCEDURE   DelStr;
              END;
  PMStr     = ^TMStr;
  TMStr     = RECORD
                line : STRING[132];
                next : PMStr;
              END;
```

```
  PMap       = ^TMap;  { Nimmt Karte fuers Automapping auf }
  TMap       = OBJECT
                 start       : PMStr;
                 CONSTRUCTOR Init;
                 PROCEDURE   InsStr(x1,y1 : SHORTINT; str : STRING);
                 PROCEDURE   Show;
                 DESTRUCTOR  Done;
               END;

CONST
  back_kc  = #08;      { Tastaturcodes }
  del_kc   = #83;
  down_kc  = #80;
  end_kc   = #79;
  enter_kc = #13;
  esc_kc   = #27;
  f1_kc    = #59;
  f2_kc    = #60;
  f3_kc    = #61;
  f4_kc    = #62;
  f5_kc    = #63;
  f6_kc    = #64;
  f7_kc    = #65;
  f8_kc    = #66;
  f9_kc    = #67;
  f10_kc   = #68;
  home_kc  = #71;
  ins_kc   = #82;
  left_kc  = #75;
  pdown_kc = #81;
  pup_kc   = #73;
  right_kc = #77;
  up_kc    = #72;
  { Farben, da im Spiel keine CRT gebraucht wird }
  black        = 0;
  blue         = 1;
  green        = 2;
  cyan         = 3;
  red          = 4;
  magenta      = 5;
  brown        = 6;
  lightgray    = 7;
  darkgray     = 8;
  lightblue    = 9;
  lightgree    = 10;
  lightcyan    = 11;
  lightred     = 12;
  lightmagenta = 13;
  yellow       = 14;
  white        = 15;

VAR
  maxcol,              { Bildschirmspalten }
  maxrow,              { Bildschirmzeilen }
  row     : BYTE;      { aktuelle Zeile }
  replay,              { Bestaetigung fuer Textausgabe }
  debug   : BOOLEAN;   { Debug-Modus ein/aus }
```

Funktionen und Prozeduren

`PROCEDURE Print(line : STRING);`

Gibt line ohne Zeilenvorschub aus. Um einen Zeilenvorschub zu erzeugen, muß am Ende von line ein \n eingefügt werden. Die Textausgabe in Ihrem Spiel sollte immer über *Print()* erfolgen.

```
FUNCTION Question(str,chars : STRING) : CHAR;
```
Gibt str aus und wartet auf einen Tastendruck, der ein in chars vorkommendes Zeichen liefern muß. Kann für eine Menüauswahl verwendet werden.

```
PROCEDURE SetColor(fc,bc : BYTE);
```
Setzt Text- und Hintergrundfarben, wobei fc die Text- und bc die Farbe des Hintergrunds angibt.

```
PROCEDURE SetFKey(n : BYTE; str : STRING);
```
Belegt die Funktionstaste n mit str.

```
PROCEDURE SetWinTitle(title : STRING);
```
Setzt unter Windows den Fenstertitel. Unter DOS und Linux wird diese Anweisung ignoriert.

9.3 Datentypen, Funktionen und Prozeduren von tmain.pas

Neben den Objekten *TBasic*, *TItem*, *TRoom*, *TLink*, *TLock*, *TPlayer* und *TGame* definiert tmain.pas auch die von den Objekten verwendeten Datentypen, die Funktion *Adr*() sowie die Prozeduren und Funktionen für die Spielstandverwaltung.

Datentypen

```
TYPE
  PBasic  = ^TBasic;  { Zeiger auf Basisobjekt }
  PLink   = ^TLink;   { Zeiger auf Verbindung }
  PGame   = ^TGame;   { Zeiger auf Spielobjekt }
  PLib    = ^TVerb;   { Zeiger auf Eintrag im Wörterbuch }
  PItem   = ^TItem;   { Zeiger auf Gegenstand }
  PLock   = ^TLock;   { Zeiger auf Schloss fuer Tueren und Behaelter }
  PNoun   = ^TNoun;   { Zeiger auf Informationen ueber Eingabe }
  PPlayer = ^TPlayer; { Zeiger auf Spielerobjekt }
  PRoom   = ^TRoom;   { Zeiger auf Raum }
  TDir    = (north,northeast,  { Himmelsrichtungen }
             east,southeast,
             south,southwest,
             west,northwest,
             up,down,
             nowhere);
  T_Class = (room,link,item);        { Bausteine der Spielwelt }
  TMatter = (inanimate,dead,alive);  { moegliche Zustaende der Items }
  {
    TNoun enthaelt Informationen ueber die in
    der Spielereingabe ausgewaehlten Objekte
  }
  TNoun   = RECORD
              detect,
              adj      : BOOLEAN;
              number   : INTEGER;
              xroom  : PRoom;
              xlink  : PLink;
              xitem  : PItem;
            END;


  {
    Record fuer Nachricht an Objekte
  }
  TEvent  = RECORD
              action,              { Event-Konstante des Ereignisses }
              exec,                { Wie oft ausgefuehrt? }
              maxexec : BYTE;      { Wie oft maximal ausfuehren? }
              return  : BOOLEAN;   { erfolgreich ausgefuehrt? }
              who     : PItem;     { Akteur: Spieler oder NPC }
              first,               { Empfaengerobjekt }
```

```
            second   : PNoun;    { Weiteres beteiligtes Objekt }
            data     : POINTER;  { Zeiger auf zusaetzliche Daten }
          END;
{
  TWeight verwaltet Gewichte
}
TWeight   = OBJECT
              CONSTRUCTOR Init(_owner : POINTER; _min,_max : BYTE);
              FUNCTION    GetCont : BYTE;     { Zahl enthaltener Objekte}
              FUNCTION    GetSum  : BYTE;     { Gewicht }
              FUNCTION    GetMax  : BYTE;     { Maximalgewicht }
              DESTRUCTOR  Done;
              PRIVATE
              owner       : POINTER; { Item oder Spieler }
              counter,               { Anzahl enthaltener Items }
              wsum,                  { Eigengewicht + Gewicht von Items }
              wmax        : BYTE;    { Maximalgewicht }
              PROCEDURE   Pick(x : POINTER; VAR error : BOOLEAN);
              PROCEDURE   Drop(x : POINTER);
            END;
{
  Spielerobjekt
}
TPlayer = OBJECT
            CONSTRUCTOR Init(_where : WORD; _adress : TAdress;
                             _gender : TGender; _wmin,_wmax : BYTE);
            PROCEDURE   AfterLife; VIRTUAL;
            FUNCTION    GetAdress : TAdress;
            FUNCTION    GetContainer : PItem;
            FUNCTION    GetContent : BYTE;
            FUNCTION    GetGender : TGender;
            FUNCTION    GetLocation : PRoom;
            FUNCTION    GetMaxWeight : BYTE;
            FUNCTION    GetScores : WORD;
            FUNCTION    GetMoves : WORD;
            FUNCTION    GetState : BYTE;
            FUNCTION    GetWeight : BYTE;
            FUNCTION    HasPlayer(i : WORD) : BOOLEAN;
            PROCEDURE   IncScores(ds : WORD);
            PROCEDURE   Load(VAR h : File); VIRTUAL;
            PROCEDURE   MovePlayerTo(where : PBasic);
            PROCEDURE   SetState(s : BYTE);
            FUNCTION    StatusBarLeftStr : STRING; VIRTUAL;
            FUNCTION    StatusBarRightStr : STRING; VIRTUAL;
            PROCEDURE   Store(VAR h : File); VIRTUAL;
            FUNCTION    RankStr : STRING; VIRTUAL;
            PROCEDURE   Victory; VIRTUAL;
            DESTRUCTOR  Done;
          PRIVATE
            inside      : PItem;   { Zeiger auf Behaelter oder Fahrzeug }
            position    : PRoom;    { Zeiger auf aktuellen Raum }
            weight      : TWeight;
            adress      : TAdress;  { Anrede fuer Spieler }
            gender      : TGender;  { Spielergeschlecht }
            moves,                  { Runden }
            scores      : WORD;     { Punkte }
            state       : BYTE;     { Status des Spielers }
            PROCEDURE   Enter(_item : PItem);
            PROCEDURE   IncMoves;
            PROCEDURE   Inventory;
            PROCEDURE   Leave;
          END;
{
  Basisobjekt fuer Raeume, Durchgaenge und Items
}
TBasic  = OBJECT
            name        : TMemText;   { Objektname als Listenobjekt }
```

```
              lock         : PLock;      { Schlosser fuer Links und Items }
              CONSTRUCTOR Init(_id : WORD; _name : STRING;
                                   _objclass : T_Class);
              PROCEDURE   AddText(str : STRING);
              PROCEDURE   DelText;
              FUNCTION    GetClass : T_Class;
              FUNCTION    GetCopies : BYTE;
              FUNCTION    GetID : WORD;
              PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
              FUNCTION    HasDaemon : BOOLEAN;
              FUNCTION    HasTimer : BOOLEAN;
              PROCEDURE   Inspect; VIRTUAL;
              PROCEDURE   Load(VAR h : File); VIRTUAL;
              PROCEDURE   MyText; VIRTUAL;
              PROCEDURE   RunDaemon; VIRTUAL;
              FUNCTION    Scope : BYTE; VIRTUAL;
              PROCEDURE   StartDaemon;
              PROCEDURE   StopDaemon;
              PROCEDURE   StartTimer(rounds : BYTE);
              PROCEDURE   StopTimer;
              PROCEDURE   Store(VAR h : File); VIRTUAL;
              PROCEDURE   TimeOut; VIRTUAL;
              PROCEDURE   View; VIRTUAL;
              DESTRUCTOR  Done; VIRTUAL;
            PRIVATE
              _class      : T_Class;  { room, item oder link }
              id          : WORD;     { ID zur eindeutigen Identifikation }
              copies,                 { Anzahl erreichbarer Objekte mit
                                        gleichem Namen }
              counter     : BYTE;     { Zaehler fuer Timer }
              flag,                   { wichtig }
              daemon,                 { Daemon eingeschaltet? }
              timer,                  { Timer eingeschaltet? }
              tracing     : BOOLEAN;  { Objektzustand verfolgen? }
              {$IFDEF tp}
              objtext     : PDiskText; { TP: Text in Datei speichern }
              {$ELSE}
              objtext     : PMemText; { Delphi/FPC: Text in Liste }
              {$ENDIF}
              PROCEDURE   ObscureEvents;
              PROCEDURE   RunTimer;
              PROCEDURE   Reset;
              PROCEDURE   TextOut;
            END;
  {
    TLock: Schloesser fuer Tueren und Behaelter
  }
  TState  = (open,closed,locked);  { Zustaende fuer Schloesser }
  TLock   = OBJECT
              CONSTRUCTOR Init(_owner : PBasic; _openable,
                                   _closeable : BOOLEAN; _state : TState;
                                   _key : PItem);
              FUNCTION    AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    GetKey : PItem;
              FUNCTION    GetOwner : PBasic;
              FUNCTION    GetState : TState;
              PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
              FUNCTION    HasKey(event : Tevent) : BOOLEAN;
              FUNCTION    IsOpenable : BOOLEAN;
              FUNCTION    IsCloseable : BOOLEAN;
              PROCEDURE   Load(VAR h : File); VIRTUAL;
              PROCEDURE   SetCloseable(s : BOOLEAN);
              PROCEDURE   SetOpenable(s : BOOLEAN);
              PROCEDURE   SetState(s : TState);
              PROCEDURE   Store(VAR h : File); VIRTUAL;
              PROCEDURE   View; VIRTUAL;
```

```
              DESTRUCTOR  Done; VIRTUAL;
              PRIVATE
              key         : PItem;    { Instanz von TItem oder NIL }
              owner       : PBasic;   { Instanz von TItem oder TLink }
              openable,               { kann das Schloss geoeffnet und }
              closeable   : BOOLEAN;  { geschlossen werden? }
              state       : TState; { offen, geschlossen, verschlossen }
            END;
  {
    TLink stellt eine Verbindung zwischen jeweils zwei Raeumen her
  }
  TLink   = OBJECT(TBasic)
              r1,r2       : PRoom;
              CONSTRUCTOR Init(_id : WORD; _name : STRING; _from : WORD;
                                              _dirto : TDir; _to : WORD;
_show : BOOLEAN);
              FUNCTION    BeforeAction(VAR event : TEvent) : BOOLEAN;
VIRTUAL;
              FUNCTION    AfterAction(VAR event : TEvent) : BOOLEAN;
VIRTUAL;
              PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
              FUNCTION    HasAutoDescription : BOOLEAN;
              PROCEDURE   Load(VAR h : File); VIRTUAL;
              FUNCTION    Scope : BYTE; VIRTUAL;
              PROCEDURE   Store(VAR h : File); VIRTUAL;
              PROCEDURE   View; VIRTUAL;
              DESTRUCTOR  Done; VIRTUAL;
            PRIVATE
              show        : BOOLEAN; { Objekt erwaehnen? }
            END;
  {
    TRoom, der Objekttyp fuer alle Raeume
  }
  TRoom   = OBJECT(TBasic)
              gate        : ARRAY[north..down] OF PLink;
              CONSTRUCTOR Init(_id : WORD; _name : STRING);
              FUNCTION    AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    FromDir : TDir;
              PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
              FUNCTION    HasLight : BOOLEAN;
              FUNCTION    IsGate(d : TDir) : TState;
              PROCEDURE   Load(VAR h : File); VIRTUAL;
              PROCEDURE   MyDarkness; VIRTUAL;
              PROCEDURE   SetLight(_light : BOOLEAN);
              PROCEDURE   Store(VAR h : File); VIRTUAL;
              FUNCTION    Scope : BYTE; VIRTUAL;
              FUNCTION    ToDir : TDir;
              PROCEDURE   View; VIRTUAL;
              DESTRUCTOR  Done; VIRTUAL;
            PRIVATE
              light,                  { Kann Spieler hier etwas sehen? }
              explored    : BOOLEAN;  { Deja-vu? }
              wto,wfrom   : TDir;     { Spieler will wohin / kommt woher? }
              PROCEDURE   CountLinks;
              PROCEDURE   RoomDescription;
            END;
  {
    TAttrib dient der Verwaltung von Item-Attributen in einer Liste
  }
  PAttrib   = ^TAttrib;
  TAttrib   = RECORD
                attribute : BYTE;
                prev,next : PAttrib;
            END;
  {
    TItem stellt alle Gegenstaende im Spiel dar
```

```
  }
  TItem   = OBJECT(TBasic)
              CONSTRUCTOR Init(_id : WORD; _name : STRING;
                               _location : PBasic; _show : BOOLEAN;
                               _wmin,_wmax : BYTE);
              FUNCTION    AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
              FUNCTION    Contains(x : PItem) : BOOLEAN;
              FUNCTION    DecAmount : BOOLEAN;
              FUNCTION    GetAmount : BYTE;
              FUNCTION    GetContainer : PItem;
              FUNCTION    GetContent : BYTE;
              FUNCTION    GetLocation : PRoom;
              FUNCTION    GetMatter : TMatter;
              FUNCTION    GetMaxWeight : BYTE;
              FUNCTION    GetWeight : BYTE;
              PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
              FUNCTION    Has(p : BYTE) : BOOLEAN;
              FUNCTION    HasAutoDescription : BOOLEAN;
              FUNCTION    IsOn : BOOLEAN;
              FUNCTION    IsOff : BOOLEAN;
              PROCEDURE   ListMe(s : BOOLEAN);
              PROCEDURE   Load(VAR h : File); VIRTUAL;
              PROCEDURE   MoveCmd(where : POINTER; VAR event : TEvent);
              PROCEDURE   MoveItemTo(where : PBasic);
              FUNCTION    Scope : BYTE; VIRTUAL;
              PROCEDURE   SetAttrib(a : BYTE; _on : BOOLEAN);
              PROCEDURE   SetMatter(state : TMatter);
              PROCEDURE   SetPraepos(pp : STRING);
              PROCEDURE   SetAmount(n : BYTE);
              PROCEDURE   Store(VAR h : File); VIRTUAL;
              PROCEDURE   SwitchOn;
              PROCEDURE   SwitchOff;
              PROCEDURE   View; VIRTUAL;
              DESTRUCTOR  Done; VIRTUAL;
            PRIVATE
              location     : PRoom;
              inside       : PItem;
              weight       : TWeight;
              matter       : TMatter;   { Objektzustand? }
              pstart       : PAttrib;   { Startadresse Eigenschaften }
              show,                     { In Raumbeschreibung erwaehnen? }
              on           : BOOLEAN;   { Schalterzustand }
              amount       : BYTE;        { Menge,Tankfuellung etc. }
              praepos      : STRING[4]; { Praeposition }
              PROCEDURE   ClearAttributes;
              PROCEDURE   Follow;
              PROCEDURE   Register;
            END;
  {
    TVerb bildet das Woerterbuch
  }
  TVerb  = RECORD
             verb,syntax : TMemText;  { Verb und zugehoerige Tokenfolge }
             message     : BYTE;      { Vom Verb ausgeloestes Ereignis }
             prev,next   : PLib;      { vorheriges/naechstes Verb }
           END;
  {
    Hauptobjekt: Steuert Dialog mit Spieler, interpretiert dessen
    Eingabe und versendet Nachrichten an betroffene Objekte
  }
  TGame   = OBJECT
              CONSTRUCTOR Init(_statusline,_upsize,_verbose : BOOLEAN;
                               _history : BYTE);
              FUNCTION    ActorWhere : PRoom;
              FUNCTION    ActorInside(container : PItem) : BOOLEAN;
              PROCEDURE   AddProlog(_str : STRING);
```

```
    PROCEDURE   AddVerb(_verb,_syntax : STRING; _event : BYTE);
    PROCEDURE   Admonition(adress : TAdress); VIRTUAL;
    FUNCTION    AfterAction(event : TEvent) : BOOLEAN;
    FUNCTION    BeforeAction(event : TEvent) : BOOLEAN;
    PROCEDURE   BeforeParsing(VAR input : STRING); VIRTUAL;
    PROCEDURE   ClearDictionary;
    PROCEDURE   DelVerb(_event : BYTE);
    FUNCTION    GetActor : PItem;
    PROCEDURE   GetTime(VAR hour,min : BYTE);
    PROCEDURE   HandleEvent(VAR event : TEvent); VIRTUAL;
    FUNCTION    MetaVerb(mv : BYTE) : BOOLEAN; VIRTUAL;
    FUNCTION    MyScope(_id : WORD; _action : BYTE) : BOOLEAN;
                        VIRTUAL;
    PROCEDURE   ParserError(n : BYTE; str : STRING; i : BYTE;
                            VAR error : BOOLEAN); VIRTUAL;
    PROCEDURE   Run;
    PROCEDURE   SetTime(_time,_rate : WORD);
    PROCEDURE   WriteProlog;
    DESTRUCTOR  Done;
  PRIVATE
    {$IFDEF tp}
    prologue    : PDiskText;  { Vorspann }
    {$ELSE}
    prologue    : PMemText;
    {$ENDIF}
    statusline,             { Statuszeile anzeigen? }
    verbose,                { Ausfuehrliche Beschreibung? }
    upsize,                 { Eingabe in Grossschrift? }
    meta        : BOOLEAN;  { Ist erkanntes Verb ein Metaverb? }
    history,                { Anzahl Zeilen im Historypuffer }
    quit,                   { Spielende erreicht? }
    maxwords    : BYTE;     { Zahl eingegebener Woerter }
    time,                   { Zeit in Minuten seit 0:00 Uhr }
    rate        : WORD;     { Minuten pro Runde }
    buffer      : TBuffer;  { Eingabepuffer }
    actor       : PItem;    { Akteuer }
    pverb       : PLib;     { Startadresse Woerterbuch }
    refpro      : PNoun;    { Substantiv fuer Reflexivpronomen }
    nounstr     : ARRAY[1..maxbuf] OF TNoun;  { wichtig }
    PROCEDURE   Browser;
    PROCEDURE   CheckCommandLine;
    FUNCTION    CmpScope(index,ttype,_action : BYTE) : BOOLEAN;
    PROCEDURE   DetectNumeral(w0 : BYTE; VAR number : INTEGER;
                              VAR ok : BOOLEAN);
    PROCEDURE   DrawMap;
    PROCEDURE   DrawStatusline(location : BOOLEAN);
    PROCEDURE   InitEvent(VAR event : TEvent; _what,
                          _maxexec : BYTE; _who : PItem; _first,
                          _second : PNoun; _data : POINTER);
    FUNCTION    IsArtOrPron(str : STRING) : BOOLEAN;
    PROCEDURE   LoadGame(n : BYTE);
    PROCEDURE   Mapping(p : PRoom; x0,y0 : SHORTINT;
                        VAR map : PMap);
    PROCEDURE   Parse(line : STRING);
    PROCEDURE   QuitGame;
    PROCEDURE   ReadLib;
    PROCEDURE   SaveGame(n : BYTE);
    PROCEDURE   SearchNoun(VAR w0 : BYTE; t0 : BYTE;
                           VAR tfound : BYTE; ttype,
                           _action : BYTE;
                           VAR userstr : STRING;
                           VAR pfailed : PBasic;
                           VAR error : BOOLEAN);
    FUNCTION    IsVerbose : BOOLEAN;
    PROCEDURE   UserLoad;
    PROCEDURE   UserSave;
  END;
```

```
CONST
  sdstr : ARRAY[0..4] OF STRING[12]   { Namen derScope-Definitionen }
        = ('notinroom_sd',
           'visible_sd',
           'reachable_sd',
           'bynpc_sd',
           'held_sd');
  {
    PlayerState: Spiel wird beendet, wenn Status des Spielers <> alive_ps
  }
  dead_ps    = 0;
  alive_ps   = 1;
  victory_ps = 2;
  {
    Scope-Definitionen
  }
  notinroom_sd = 0;   { Objekt ist ausser Sicht und Reichweite }
  visible_sd   = 1;   { Objekt ist sichtbar }
  reachable_sd = 2;   { Objekt ist sichtbar und erreichbar }
  bynpc_sd     = 3;   { Objekt wird von NPC gehalten }
  held_sd      = 4;   { Objekt wird vom Spieler gehalten }
  {
    Vordefinierte Ereignisse
  }
  close_ev      = 100;
  dance_ev      = 101;
  dec_ev        = 102;
  drink_ev      = 103;
  eat_ev        = 104;
  enter_ev      = 105;
  examine_ev    = 106;
  give_ev       = 107;
  go_ev         = 108;
  inv_ev        = 109;
  drop_ev       = 110;
  jump_ev       = 111;
  kill_ev       = 112;
  kiss_ev       = 113;
  leave_ev      = 114;
  lista_ev      = 115;
  lists_ev      = 116;
  load_ev       = 117;
  lock_ev       = 118;
  look_ev       = 119;
  map_ev        = 120;
  open_ev       = 121;
  press_ev      = 122;
  quit_ev       = 123;
  restart_ev    = 124;
  read_ev       = 125;
  score_ev      = 126;
  save_ev       = 127;
  set1_ev       = 128;
  set2_ev       = 129;
  show_ev       = 130;
  sleep_ev      = 131;
  smell_ev      = 132;
  switchoff_ev  = 133;
  switchon_ev   = 134;
  take_ev       = 135;
  tell_ev       = 136;
  touch_ev      = 137;
  trace_ev      = 138;
  wait_ev       = 139;
```

```
  {
    vordefinierte Item-Eigenschaften
  }
  drinkable_at   = 0;
  edable_at      = 1;
  enterable_at   = 2;
  moveable_at    = 3;
  readable_at    = 4;
  shining_at     = 5;
  switchable_at  = 6;
  takeable_at    = 7;
  talkable_at    = 8;
  transparent_at = 9;

VAR
  top        : PRoom;      { Zeiger auf Raum 0 }
  useroption : BOOLEAN;    { Kommandozeilenoptionen ein/aus }
```

Funktionen und Prozeduren

`FUNCTION  Adr(_id : WORD) : POINTER;`
Liefert einen Zeiger auf das Objekt mit der ID _id. Kann mit einer expliziten Typumwandlung in einen Zeiger vom Typ *PBasic*, *PRoom*, *PItem* oder *Plink* umgewandelt werden: *PRoom(Adr(forest_id)).*

`PROCEDURE Prepare(_filename : STRING; _new : BOOLEAN);`
Bestimmt Dateinamen und initialisiert den als Vesteck für Objekte dienenden Raum 0 (top).

`PROCEDURE WritePtr(VAR h : File; p : PBasic);`
Speichert einen Zeiger auf einen Nachfolger von *TBasic* in der Spielstanddatei h.

`FUNCTION  ReadPtr(VAR h : File) : POINTER;`
Liest einen Zeiger auf einen Nachfolger von *TBasic* aus der Spielstanddatei h.

`PROCEDURE WriteWord(VAR h : File; w : WORD);`
Speichert einen Word-Wert in der Spielstanddatei h.

`PROCEDURE ReadWord(VAR h : File; VAR w : WORD);`
Liest einen Word-Wert aus der Spielstanddatei h.

`PROCEDURE WriteBool(VAR h : File; v : Boolean);`
Speichert einen Boolean-Wert in der Spielstanddatei h.

`PROCEDURE ReadBool(VAR h : File; VAR v : Boolean);`
Liest einen Boolean-Wert aus der Spielstanddatei h.

`PROCEDURE WriteDir(VAR h : File; d1 : TDir);`
Speichert einen TDir-Wert in der Spielstanddatei h.

`PROCEDURE ReadDir(VAR h : File; VAR d1 : TDir);`
Liest einen TDir-Wert aus der Spielstanddatei h.

9.4 Methoden von TBasic

TBasic stellt das Basisobjekte für alle Objekte der Spielwelt dar. Von ihm werden die Objekttypen *TItem*, *TRoom* und *TLink* abgeleitet. Die privaten Methoden von *TBasic* und der anderen in tmain.pas definerten Objekttypen werden hier nicht aufgeführt, sind im Quelltext aber kommentiert.

`CONSTRUCTOR Init(_id : WORD; _name : STRING; _objclass : T_Class);`
Initialisiert ein Objekt. Wird bei der Initialisierung aller Nachfolger von *TBasic* aufgerufen. Der Parameter _objclass kann die Werte *item*, *room* und *link* annehmen.

`PROCEDURE AddText(str : STRING);`
Ordnet dem Objekt eine Beschreibung zu. Mehrere Methodenaufrufe können einen Text mit mehr als 255 Zeichen zuordnen.

`PROCEDURE DelText;`
Entfernt die mit *AddText()* hinzugefügte Beschreibung wieder.

`FUNCTION GetClass : T_Class;`
Liefert den Typ eines Objekts: *item*, *room* oder *link*. So läßt sich feststellen, in welchen Zeigertyp sich ein Zeiger vom Typ *PBasic* umwandeln läßt.

`FUNCTION GetCopies : BYTE;`
Liefert die Anzahl der Objekte, die sich im gleichen Raum befinden und den gleichen Namen haben.

`FUNCTION GetID : WORD;`
Liefert die ID des Objekts.

`PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;`
Ereignisroutine (s. 6.1). Verarbeitet die Ereignisse dance_ev, dec_ev, examine_ev, jump_ev, lists_ev, touch_ev und trace_ev. Diese Ereignisse werden von den Ereignisroutinen der Nachfolger von *TBasic* weitergereicht und können auch dort behandelt werden.

`FUNCTION HasDaemon : BOOLEAN;`
Ist *True*, wenn der Dämon des Objekts aktiv ist.

`FUNCTION HasTimer : BOOLEAN;`
Ist *True*, wenn der Zeitschalter des Objekts aktiv ist.

`PROCEDURE Inspect; VIRTUAL;`
Gibt nach jeder Runde Informationen aus, wenn das Objekt mit .trace (s. 7.6) verfolgt wird.

`PROCEDURE Load(VAR h : File); VIRTUAL;`
Lädt gespeicherten Zustand.

`PROCEDURE MyText; VIRTUAL;`
Gibt variable Beschreibung aus, wenn dem Objekt mit *AddText()* kein Text zugeordnet wurde.

`PROCEDURE RunDaemon; VIRTUAL;`
Führt einen aktiven Hintergrundprozeß (s. 8.3) aus.

`FUNCTION Scope : BYTE; VIRTUAL;`
Liefert Information über die Erreichbarkeit des Objekts (s. 5.7).

`PROCEDURE StartDaemon;`
Aktiviert den Hintergrundprozeß des Objekts.

`PROCEDURE StopDaemon;`
Beendet den Hintergrundprozeß des Objekts.

`PROCEDURE StartTimer(time : BYTE);`
Aktiviert den Zeitschalter des Objekts (s. 8.3).

`PROCEDURE StopTimer;`
Beendet den Zeitschalter vorzeitig.

`PROCEDURE Store(VAR h : File); VIRTUAL;`
Speichert den aktuellen Zustand.

`PROCEDURE TimeOut; VIRTUAL;`
Wird von abgelaufenem Zeitschalter ausgeführt.

```
PROCEDURE View; VIRTUAL;
```
Gibt Informationen für den von .browse und .list aktivierten Objekt-Betrachter aus. Wird von den Nachfolgern von *TBasic* überschrieben.

```
DESTRUCTOR Done; VIRTUAL;
```
Löscht dynamische Variablen.

9.5 Methoden von TItem

TItem ist ein Nachfolger von *TBasic* und kann fast alle möglichen und unmöglichen Dinge in der Spielwelt darstellen.

```
CONSTRUCTOR Init(_id : WORD; _name : STRING; _location : PBasic;
                 _show : BOOLEAN; _wmin,_wmax : BYTE);
```
Initialisiert einen Gegenstand (s. 5.1).

```
FUNCTION AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von *TItem.HandleEvent()* nach Ausführung eines Ereignisses aufgerufen und verhindert eine Textausgabe, wenn *False* zurückgegeben wird.

```
FUNCTION BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von *TItem.HandleEvent()* vor Ausführung eines Ereignisses aufgerufen und verhindert eine weitere Bearbeitung, wenn *False* zurückgegeben wird.

```
FUNCTION Contains(x : PItem) : BOOLEAN;
```
Stellt fest, ob das Objekt enthalten ist.

```
FUNCTION DecAmount : BOOLEAN;
```
Reduziert die Menge des Objekts um eine Einheit. Wird bislang nur von den Ereignissen eat_ev und drink_ev verwendet (s. *SetAmount()).*

```
FUNCTION GetAmount : BYTE;
```
Liefert die Menge des Objekts.

```
FUNCTION GetContainer : PItem;
```
Liefert den Behälter des Objekts.

```
FUNCTION GetContent : BYTE;
```
Liefert Anzahl enthaltener Objekte.

```
FUNCTION GetLocation : PRoom;
```
Liefert den Raum, in dem sich das Objekt (und evtl. sein Behälter) befindet.

```
FUNCTION GetMatter : TMatter;
```
Liefert den Zustand des Objekts: inanimate, dead oder alive.

```
FUNCTION GetMaxWeight : BYTE;
```
Liefert das Maximalgewicht.

```
FUNCTION GetWeight : BYTE;
```
Liefert das Gewicht.

```
PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;
```
Ereignisroutine für die Ereignisse close_ev, drink_ev, drop_ev, eat_ev, enter_ev, give_ev, kill_ev, kiss_ev, leave_ev, lock_ev, open_ev, read_ev, show_ev, switchoff_ev, switchon_ev, press_ev, take_ev und tell_ev.

```
FUNCTION Has(p : BYTE) : BOOLEAN;
```
Stellt fest, ob das Objekt das Attribut p hat. p ist der Wert einer Attributskonstante mit der Endung *_at.

`FUNCTION HasAutoDescription : BOOLEAN;`
Stellt fest, ob das Objekt nach der Raumbeschreibungen automatisch erwähnt wird.

`FUNCTION IsOn : BOOLEAN;`
Stellt fest, ob das Objekt eingeschaltet ist, wenn es das Attribut
switchable_at hat.

`FUNCTION IsOff : BOOLEAN;`
Stellt fest, ob das Objekt ausgeschaltet ist, wenn es das Attribut switchable_at hat.

`PROCEDURE ListMe(s : BOOLEAN);`
Wenn *True* übergeben wird, wird das Objekt nach der Raumbeschreibung automatisch erwähnt. Ansonsten kann es in der Raumbeschreibung erwähnt oder vor dem Spieler versteckt werden.

`PROCEDURE Load(VAR h : File); VIRTUAL;`
s. *TBasic.Load()*.

`PROCEDURE MoveCmd(where : POINTER; VAR event : TEvent);`
Wird von *TItem.HandleEvent()* für die Bearbeitung der Ereignisse give_ev, drop_ev und take_ev aufgerufen und bewegt das Objekt an die durch where angegebene Position. Dabei werden *BeforeAction()* und *AfterAction()* aufgerufen. Wenn Sie die Position eines Gegenstandes ändern wollen, sollten Sie die Methode *MoveItemTo()* verwenden.

`PROCEDURE MoveItemTo(where : PBasic);`
Bewegt das Objekt an die durch where angegebene Position, dabei muß where ein Zeiger auf einen Raum oder einen Behälter sein. *BeforeAction()* und *AfterAction()* werden nicht aufgerufen. Die Positionsveränderung kann vom Spieler unbemerkt erfolgen.

`FUNCTION Scope : BYTE; VIRTUAL;`
s. *TBasic.Scope*.

`PROCEDURE SetAttrib(a : BYTE; _on : BOOLEAN);`
Setzt oder entfernt das Attribut a (s. 5.1).

`PROCEDURE SetMatter(state : TMatter);`
Setzt den Zustand des Objekts auf in*animate*, *dead* oder *alive*.

`PROCEDURE SetPraepos(pp : STRING);`
Bestimmt die Präposition (meist "auf" oder "in"), wenn der Spieler das Objekt betreten kann (s. 5.1).

`PROCEDURE SetAmount(n : BYTE);`
Bestimmt eine Menge, die bislang nur von den Ereignissen eat_ev und drink_ev verringert wird. Vulgo: Diese Menge gibt an, wie oft der Spieler von einem Brot essen oder aus einer Flasche trinken kann. Die Voreinstellung ist eins.

`PROCEDURE Store(VAR h : File); VIRTUAL;`
Speichert aktuellen Objektzustand.

`PROCEDURE SwitchOn;`
Schaltet das Objekt ein. Das Attribut switchable_ev muß vorhanden sein.

`PROCEDURE SwitchOff;`
Schaltet das Objekt aus. Das Attribut switchable_ev muß vorhanden sein.

`PROCEDURE View; VIRTUAL;`
Gibt Informationen für den mit .browse und .list aktivierten Objekt-Betrachter aus.

`PROCEDURE Done; VIRTUAL;`
s. *TBasic.Done*.

9.6 Methoden von TRoom

TRoom ist ein Nachfolger von *TBasic* und stellt einzelne Räume dar.

`CONSTRUCTOR Init(_id : WORD; _name : STRING);`
Initialisiert einen Raum (s. 4.3).

`FUNCTION FromDir : TDir;`
Stellt fest, aus welcher Richtung der Spieler den Raum betreten hat. Wenn die Methode den Wert nowhere liefert, war der Spieler hier noch nie oder hat das Spiel hier begonnen.

`FUNCTION AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;`
Wird von *TRoom.HandleEvent()* nach Ausführung eines Ereignisses aufgerufen und verhindert eine Textausgabe, wenn *False* zurückgegeben wird.

`FUNCTION BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;`
Wird von *TRoom.HandleEvent()* vor Ausführung eines Ereignisses aufgerufen und verhindert eine weitere Bearbeitung, wenn *False* zurückgegeben wird.

`PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;`
Ereignisroutine für die Ereignisse go_ev und look_ev.

`FUNCTION HasLight : BOOLEAN;`
Stellt fest, ob der Spieler hier sehen kann (s. 8.4).

`FUNCTION IsGate(d : TDir) : TState;`
Stellt fest, ob eine Verbindung/Tür in der angegebenen Richtung geöffnet (*open*), geschlossen (*closed*) oder verschlossen (*locked*) ist.

`PROCEDURE Load(VAR h : File); VIRTUAL;`
s. *TBasic.Load().*

`PROCEDURE MyDarkness; VIRTUAL;`
Gibt einen Text an Stelle der Raumbeschreibung aus, wenn der Raum nicht beleuchtet ist.

`PROCEDURE SetLight(_light : BOOLEAN);`
Schaltet die Beleuchtung ein und aus (s. 8.4).

`PROCEDURE Store(VAR h : File); VIRTUAL;`
s. *TBasic.Store().*

`FUNCTION Scope : BYTE; VIRTUAL;`
Der Scope eines Raums ist visible_sd, wenn der Spieler sich in ihm aufhält, oder notinroom_sd, wenn er sich in einem anderen Raum aufhält.

`FUNCTION ToDir : TDir;`
Stellt fest, in welche Richtung der Spieler den Raum verlassen will oder verlassen hat. Wenn diese Methode *nowhere* zurückliefert war der Spieler hier noch nie oder hat das Spiel hier begonnen und den Raum noch nicht verlassen.

`PROCEDURE View; VIRTUAL;`
s. *TBasic.View*.

`DESTRUCTOR Done; VIRTUAL;`
s. *TBasic.Done*.

9.7 Methoden von TLink

TLink ist ein Nachfolger von *TBasic* und verbindet zwei Räume.

```
CONSTRUCTOR Init(_id : WORD; _name : STRING; _from : WORD;
                 _dirto : TDir; _to : WORD; _show : BOOLEAN);
```
Initialisiert eine Verbindung (s. 5.6).

```
FUNCTION BeforeAction(VAR event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von *TLink.HandleEvent()* vor Ausführung eines Ereignisses aufgerufen und verhindert eine weitere Bearbeitung, wenn *False* zurückgegeben wird.

```
FUNCTION AfterAction(VAR event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von *TLink.HandleEvent()* nach Ausführung eines Ereignisses aufgerufen und verhindert eine Textausgabe, wenn *False* zurückgegeben wird.

```
PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;
```
Gibt die Ereignisse close_ev, lock_ev und open_ev an das zugeordnete Objekt vom Typ *TLock* weiter.

```
FUNCTION HasAutoDescription : BOOLEAN;
```
Stellt fest, ob die Verbindung (als Ausgang) nach der Raumbeschreibung erwähnt wird.

```
PROCEDURE Load(VAR h : File); VIRTUAL;
```
s. *TBasic.Load().*

```
FUNCTION Scope : BYTE; VIRTUAL;
```
s. *TBasic.Scope().*

```
PROCEDURE Store(VAR h : File); VIRTUAL;
```
s. *TBasic.Store().*

```
PROCEDURE View; VIRTUAL;
```
s. *TBasic.View.*

```
DESTRUCTOR Done; VIRTUAL;
```
s. *TBasic.Done().*

9.8 Methoden von TLock

TLock dient Behältern und Verbindungen als Schloß.

```
CONSTRUCTOR Init(_owner : PBasic; _openable,_closeable : BOOLEAN;
                 _state : TState; _key : PItem);
```
Initialisiert ein Schloß und ordnet es einem Gegenstand oder einer Verbindung zu (s. 5.2).

```
FUNCTION AfterAction(event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von *TLock.HandleEvent()* nach Ausführung eines Ereignisses aufgerufen und verhindert eine Textausgabe, wenn *False* zurückgegeben wird.

```
FUNCTION BeforeAction(event : TEvent) : BOOLEAN; VIRTUAL;
```
Wird von TLock.*HandleEvent*() vor Ausführung eines Ereignisses aufgerufen und verhindert eine weitere Bearbeitung, wenn *False* zurückgegeben wird.

```
FUNCTION GetKey : PItem;
```
Liefert einen Zeiger auf das Objekt, mit dem sich das Schloß auf- und abschließen läßt.

```
FUNCTION GetOwner : PBasic;
```
Liefert einen Zeiger auf ein Objekt vom Typ *PItem oder PLink*, dem das Schloß zugeordnet ist.

```
FUNCTION GetState : TState;
```
Liefert den Zustand des Schlosses: *open, closed* oder *locked*. Alternativ kann auch die Methode *TRoom.IsGate()* verwendet werden.

```
PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;
```
Ereignisroutine für die Ereignisse close_ev, lock_ev und open_ev.

`FUNCTION HasKey(event : Tevent) : BOOLEAN;`
Stellt fest, ob der Akteur einen Schlüssel für das Schloß trägt.

`FUNCTION IsOpenable : BOOLEAN;`
Stellt fest, ob sich das Schloß öffnen läßt.

`FUNCTION IsCloseable : BOOLEAN;`
Stellt fest, ob sich das Schloß schließen und abschließen läßt.

`PROCEDURE Load(VAR h : File); VIRTUAL;`
Speichert aktuellen Objektzustand.

`PROCEDURE SetCloseable(s : BOOLEAN);`
Bestimmt, ob sich das Objekt schließen und/oder abschließen läßt.

`PROCEDURE SetOpenable(s : BOOLEAN);`
Bestimmt, ob sich das Objekt öffnen läßt.

`PROCEDURE SetState(s : TState);`
Bestimmt den Zustand des Schlosses: *open, closed* oder *locked*.

`PROCEDURE Store(VAR h : File); VIRTUAL;`
Lädt gespeicherten Zustand.

`PROCEDURE View; VIRTUAL;`
Gibt Informationen im Objekt-Betrachter aus.

`DESTRUCTOR Done; VIRTUAL;`
s. *TBasic.Done*.

9.9 Methoden von TPlayer

TPlayer verwaltet die den Spieler betreffenden Informationen.

```
CONSTRUCTOR Init(_where : WORD; _adress : TAdress;
                 _gender : TGender; _wmin,_wmax : BYTE);
```
Initialisiert das Spielerobjekt (s. 4.5).

`PROCEDURE AfterLife; VIRTUAL;`
Wird nach dem Tod des Spielers aufgerufen, um eine beliebige Meldung auszugeben.

`FUNCTION GetAdress : TAdress;`
Liefert die im Konstruktor angegebene Anrede für den Spieler: du, sie ihr.

`FUNCTION GetContainer : PItem;`
Liefert einen Zeiger auf das Objekt, in dem der Spieler sich befindet.

`FUNCTION GetContent : BYTE;`
Liefert die Anzahl der Objekte im Inventar des Spielers.

`FUNCTION GetGender : TGender;`
Liefert das im Konstruktor angegeben Geschlecht des Spielers: *male* oder *female*.

`FUNCTION GetLocation : PRoom;`
Liefert den aktuellen Raum.

`FUNCTION GetMaxWeight : BYTE;`
Liefert das Maximalgewicht, das der Spieler tragen kann.

`FUNCTION GetScores : WORD;`
Liefert die Punktzahl des Spielers.

`FUNCTION GetMoves : WORD;`
Liefert die Zahl der bisherigen Spielzüge.

`FUNCTION GetState : BYTE;`
Liefert den Zustand des Spielers: alive_ps, dead_ps oder victory_ps.

`FUNCTION GetWeight : BYTE;`
Liefert das Gewicht des Spielers.

`FUNCTION HasPlayer(i : WORD) : BOOLEAN;`
Stellt fest, ob sich das Objekt mit der ID i im Inventar befindet.

`PROCEDURE IncScores(ds : WORD);`
Erhöht die Punktzahl des Spielers um ds Punkte.

`PROCEDURE Load(VAR h : File); VIRTUAL;`
Lädt den Zustand des Spielers.

`PROCEDURE MovePlayerTo(where : PBasic);`
Bewegt (oder teleportiert) den Spieler zu der Position where, dieser Parameter muß ein Raum oder ein Objekt mit dem Attribut enterable_at sein.

`PROCEDURE SetState(s : BYTE);`
Verändert den Spielerstatus. *SetState(dead_ps)* tötet den Spieler, während ihn *SetState(victory_ps)* das Spiel gewinnen läßt.

`FUNCTION StatusBarLeftStr : STRING; VIRTUAL;`
Bestimmt den linksbündigen Text in der Statuszeile.

`FUNCTION StatusBarRightStr : STRING; VIRTUAL;`
Bestimmt den rechtsbündigen Text in der Statuszeile.

`PROCEDURE Store(VAR h : File); VIRTUAL;`
Speichert den Zustand des Spielers.

`FUNCTION RankStr : STRING; VIRTUAL;`
Gibt die Punktzahl und wahlweise weitere Informationen aus.

`PROCEDURE Victory; VIRTUAL;`
Wird nach dem Sieg des Spielers aufgerufen, um eine beliebige Meldung auszugeben.

`DESTRUCTOR Done;`
Löscht dynamische Variablen.

9.10 Methoden von TGame

TGame beinhaltet den Parser und steuert den Spielverlauf.

`CONSTRUCTOR Init(_statusline,_upsize,_verbose : BOOLEAN; _history : BYTE);`
Initialisiert das Spiel (s. 4.5).

`FUNCTION ActorWhere : PRoom;`
Liefert den Aufenthaltsort des Akteurs.

`FUNCTION ActorInside(container : PItem) : BOOLEAN;`
Stellt fest, ob sich der Akteur in container befindet.

```
PROCEDURE AddProlog(_str : STRING);
```
Fügt dem Spiel einen Vorspann hinzu. Mit mehreren *AddProlog()* kann auch ein Text von mehr als 255 Zeichen zugewiesen werden.

```
PROCEDURE AddVerb(_verb,_syntax : STRING; _event : BYTE);
```
Fügt dem Wörterbuch ein neues Verb hinzu (s. 7.2).

```
PROCEDURE Admonition(adress : TAdress); VIRTUAL;
```
Fordert den Spieler auf, sich wieder auf das Spiel zu konzentrieren, wenn er eines der Ereignisse dance_ev, jump_ev und kiss_ev auslöst. Um auf diese Ereignisse anders zu reagieren, müssen Sie diese Methode oder einen Nachfolger von *TBasic.HandleEvent()* überschreiben.

```
FUNCTION AfterAction(event : TEvent) : BOOLEAN;
```
Wird von *TGame.HandleEvent()* nach Ausführung eines Ereignisses aufgerufen und verhindert eine Textausgabe, wenn *False* zurückgegeben wird.

```
FUNCTION BeforeAction(event : TEvent) : BOOLEAN;
```
Wird von *TGame.HandleEvent()* vor Ausführung eines Ereignisses aufgerufen und verhindert eine weitere Bearbeitung, wenn *False* zurückgegeben wird.

```
PROCEDURE BeforeParsing(VAR input : STRING); VIRTUAL;
```
Ermöglicht die Veränderung des Eingabestrings, bevor er an den Parser übergeben wird.

```
PROCEDURE ClearDictionary;
```
Löscht das gesamte Wörterbuch. Die Richtungsanweisungen (norden, süden, etc.) bleiben erhalten, da sie unabhängig vom Wörterbuch bearbeitet werden.

```
PROCEDURE DelVerb(_event : BYTE);
```
Entfernt alle Verben mit dem Ereignis _event aus dem Wörterbuch.

```
FUNCTION  GetActor : PItem;
```
Liefert einen Zeiger auf den Akteur. Wenn *NIL* zurückgegeben wird, ist der Spieler der Akteur.

```
PROCEDURE GetTime(VAR hour,min : BYTE);
```
Liefert die Spielzeit.

```
PROCEDURE HandleEvent(VAR event : TEvent); VIRTUAL;
```
Wird von *TBasic.HandleEvent()* für die Ereignisse lista_ev, sleep_ev und wait_ev aufgerufen.

```
FUNCTION MetaVerb(mv : BYTE) : BOOLEAN; VIRTUAL;
```
Wird vor der Ausführung eines Metaverbs aufgerufen und kann dessen weitere Bearbeitung verhindern (s. 7.5).

```
FUNCTION MyScope(_id : WORD; _action : BYTE) : BOOLEAN; VIRTUAL;
```
Bestimmt während der Analyse der Eingabe, ob das vom Spieler genannte Objekt mit der ID _id das Ereignis _action ausführen kann. Solange Sie diese Methode nicht überschreiben, liefert sie immer *True*. Diese Methode wird aufgerufen, wenn Sie in einer Grammatik das Token routine angeben.

```
PROCEDURE ParserError(n : BYTE; str : STRING; i : BYTE;
                      VAR error : BOOLEAN); VIRTUAL;
```
Gibt eine Meldung des Parsers aus, wenn bei der Analyse der Eingabe oder der Ausführung eines Ereignisses ein Problem auftritt.

```
PROCEDURE Run;
```
Startet das Spiel.

```
PROCEDURE SetTime(_time,_rate : WORD);
```
Bestimmt die Spielzeit in Minuten seit Mitternacht. Der Parameter _rate gibt an, wieviele Minuten während einer Runde verstreichen.

```
DESTRUCTOR Done;
```
Ruft am Programmende die Methode *Done* aller Nachfolger von *TBasic* auf.

10 Stichwortverzeichnis

I

K

L

M

N

O

T

U

U

W

Z