Der Abentheurliche Informissimus Teutsch

Frank Borger – mehl@frob.de

Version 0.91 β – 1. Mai 2003

„Der wichtige Umstand, bei dem uns, wie man behauptet, so viel daran gelegen ist, ihn voraus zu hören, ist nämlich der, daß Wutz eine ganze Bibliothek - wie hätte der Mann sich eine kaufen können? - sich eigenhändig schrieb. ... Er war kein verdammter Nachdrucker, der das Original hinlegt und oft das meiste daraus abdruckt: sondern er nahm gar keines zur Hand."

(Jean Paul Friedrich Richter, „Schulmeisterlein Wutz")

Einführung und Kurzreferenz für die Programmierung deutschsprachiger interaktiver Belletristik im **Z-Code** mit dem **Inform-**Compiler von GRAHAM NELSON und der deutschen Library von TONI ARNOLD (Stand: Version 18 vom Oktober 2002). Enthält außerdem Installationsanleitungen für Unix/Linux und Windows, Beispiele aus „Eden", keinen Anspruch auf Vollständigkeit und jede Menge Fehler.

Dies hier ist eine Betaversion: Wer also einen Fehler findet, darf ihn keinesfalls behalten, sondern muss so ehrlich sein, ihn mir zurückzugeben. Bitte E-Mail an `mailto:mehl@frob.de` – und im Subject irgendwo den String „Informis" unterbringen.

Inhaltsverzeichnis

1 Adam und Eva: Was ist was und was brauche ich?

1.1 Begriffe

IB oder IF Interaktive Belletristik. Halb Computerspiel, halb experimentelles Literaturgenre. (Auf Neudeutsch auch genannt „Textadventures" oder „Interactive Fiction")

Inform Objektorientierte Programmiersprache und Compiler. Entworfen zur Erstellung Interaktiver Belletristik von Graham Nelson. Ausgabeformat ist eine Spieldatei im Z-Code.

Library Interaktive Belletristik muss (einfache) in normaler Sprache abgefasste Befehle verstehen können. Intern verwendet **Inform** ein abstrahiertes, logisches Sprachmodell („Informese"). Die Library ist – vereinfacht gesagt – der Übersetzer zwischen Informese und Deutsch. Gleichzeitig funktioniert sie auch als eine Art „Schiedsrichter" und blockt (viele, nicht alle) regelwidrige Aktionen ab. Ein wesentlicher Teil der Library ist der

Parser Programmteil, der den vom Spieler eingetippten – deutschen – Satz analysiert und (hoffentlich) herausfindet, welche Aktion der Spieler ausführen will und welche Objekte dazu verwendet werden sollen. Frühe Adventures hatten *Zwei-Wort-Parser* („Nimm Schwert. Töte Troll. Nimm Schatz."); Der Parser der Inform-Library erlaubt grammatisch vollständige Sätze, Adjektive, Pronomen, ein oder mehrere direkte und ein indirektes Objekt („Nimm das leuchtende Schwert von der Wand und erschlage den Troll. Nimm alles außer der Spinne aus der Schatztruhe und schließe sie."[1])

Z-Code Befehlssprache eines virtuellen Computers (der Z-Maschine). Die legendäre Firma **Infocom** entwarf die Z-Maschine[2], um ihre Textadventures für viele verschiedene Computertypen anbieten zu können: Für jeden Typ braucht es nur einen Interpreter für Z-Code, die Spieldatei ist für alle gleich. Heute gibt es Z-Code-Interpreter für fast alles zwischen Apple und Zapple. Für den Palm gibt es z.B. Frobnitz `http://member.newsguy.com/~hangard/frobnitz/`.

[1] Das kann allerdings zu dem Versuch führen, die *Spinne* zu schließen, wenn die Autorin nicht aufgepasst hat.

[2] Wie die Legende zu sagen weiß, auf einem Küchentisch in Pittsburgh.

Glulx ist eine von Andrew Plotkin (http://www.eblong.com/zarf/glulx/) entworfene[3] modernere virtuelle Maschine mit erweiterten Möglichkeiten für Fenster, Grafik und Sound. Der Inform-Compiler kann Spieldateien im Glulx-Format erzeugen; auch die deutsche Library ist für Glulx vorbereitet. Leider gibt es außer WinGlulxe noch keinen Interpreter, der den vollen Latin-1-Zeichensatz verarbeitet und die Grafik- und Soundmöglichkeiten tatsächlich unterstützt.[4] Und auch noch keinen, der auf dem Palm läuft. Ich beschreibe die gegenüber Z-Code erweiterten Möglichkeiten von Glulx hier nicht gesondert[5], ich weise nur auf die wenigen Inkompatibilitäten hin.

TAG/TAM ist ein anderes System (Compiler und Interpreter) zum Erstellen und Spielen deutschsprachiger interaktiver Belletristik. Es ist nicht wie Inform auf Englisch entstanden und später für deutschsprachige Spiele übersetzt und angepasst, sondern von Anfang an von Martin Oehm in und für Deutsch entworfen worden. Leider ist es nur für eine Plattform (MS-DOS) erhältlich (http://www.martin-oehm.de/tag/tag.html). Es läuft allerdings auch auf Unix-Systemen unter dem DOS-Emulator XDOSEMU http://www.dosemu.org.

TADS ist wieder ein anderes System für Interaktive Belletristik. Es unterstützt mehrere Plattformen (leider nicht den Palm); seine Library ist aber bislang nicht ins Deutsche übersetzt worden.

Autorin nenne ich im nachfolgenden Text eine Person, die es nach (oder trotz) der Lektüre dieses Textes unternimmt, Interaktive Belletristik mit Inform zu verfassen.

Spieler nenne ich die Person, die das von der Autorin erstellte Werk liest. Oder löst. Oder eben damit spielt.

Spielfigur ist das Programmobjekt, das durch die Befehle des Spielers auf den von der Autorin vorgegebenen Bahnen bewegt wird.

1.2 Software

Wir brauchen den Compiler – in der deutschen Fassung – und die deutsche Library. Beides gibt bei textfire.de bereits zusammengepackt als 'Jump Start Kit' für Windows http://www.textfire.de/werkst/jumpkit.zip; verschiedene Pakete für Linux zusammen mit Installationstips sind unter http://www.textfire.de/werkst/linuxinf.htm zu finden.

[3] Ob auf dem Küchentisch oder sonstwo, ist nicht bekannt.

[4] Unter Linux funktioniert mit deutschen Sonderzeichen nur Glulxe mit cheapGlk – ohne Grafik und Sound. Glulxe mit XGlk „bockt" bei der Eingabe vun Umlauten und ß. Wer andere Interpreter getestet hat (Mac? Zaurus?) möge mir bitte Bescheid sagen.

[5] Vielleicht eines Tages in Band 2. Hoffentlich kommt mir jemand zuvor. Auf Englisch gibt es z.B. Texte von Zarf (http://www.eblong.com/zarf/glulx/inform-guide.txt), Marnie Parker (http://members.aol.com/doepage/glkdunces.htm) und Adam Cadre (http://adamcadre.ac/gull/).

Aktuelle Versionen der Library finden sich auch unter `http://www.textfire.de/archiv/index.htm`.[6]

Und natürlich brauchen wir einen Z-Code Interpreter zum „Spielen" der erstellten Datei. Die meistverwendeten sind **ZIP** und **FROTZ**, beide erhältlich für eine Vielzahl von Systemen. Die beste Quelle ist das if-Archiv, z.B. für FROTZ:

`http://www.ifarchive.org/indexes/if-archiveXinfocomXinterpretersXfrotz.html`

Eine IDE für Inform gibt es (noch) nicht; wer es bequem haben möchte, der verwendet den Editor EMACS, für den es einen Inform-modus `http://www.merguez.demon.co.uk/inform-mode/` gibt[7]. (Der Z-Code Interpreter für den EMACS, `malyon.el` kann leider wieder keine deutschen Sonderzeichen.)

1.3 Installation und Probelauf

Was hier folgt, sind nur Vorschläge, an die sich niemand halten muss. Wie auch immer der Compiler installiert wird, es muss nur sichergestellt sein, dass er das Verzeichnis mit den Bibliotheksdateien findet und dass die zu verwendende Sprache auf `german` gesetzt ist.[8] Ich beschreibe hier, wie man das mit einer systemweiten Voreinstellungsdatei macht, die Informs eigene *Command Language* ICL benutzt.[9]

In Unix-Systemen installiert man den Compiler in ein auf dem Pfad liegendes Verzeichnis, typischerweise nach `/usr/local/bin`. Dort installiert man auch den Frotz, wenn ihn nicht die verwendete Distribution (bei SuSE ist er z.B. dabei) schon anderweitig installiert hat. Die Bibliotheksdateien (einschließlich der hier mitgelieferten `GInfo.h` und `GInfoG.h`) legt man entweder in ein Verzeichnis unterhalb von `/usr/local/share/` oder im eigenen home-Bereich. Letzteres empfiehlt sich vor allem dann, wenn man selber noch ein wenig daran herumfrickeln will. Wer es bequem haben will, definiert sich noch ein alias oder ein Skript, mit dem inform immer als `inform '(~/default.icl)'` aufgerufen wird. Dabei muss man – jedenfalls mit der bash – die Klammern selbst

[6] Nur für alte Versionen (bis einschließlich 8) müssen bei einer Installation unter Linux die Dateinamen verschiedener Librarydateien angepasst werden; die Versionen ab 9 enthalten durchgängig kleingeschriebene Dateinamen:

```
mv parser.h Parser.h
mv tgerman.h TGerman.h
mv germang.h GermanG.h
mv german.h German.h
mv verblib.h VerbLib.h
```

[7] Man kann damit (und dem Programmsystem noweb) sogar Interaktive Literatur „literate" programmieren. Siehe die Beispieldatei `inform-template.nw`.

[8] Für Mac und andere Systeme als Unix und Windows kann ich zur Installation über diesen Satz hinaus wenig sagen. Wer hier eine Installationsanleitung hat, möge sie mir bitte schicken.

[9] Ich selbst bevorzuge den Aufruf über ein Makefile für das jeweilige Projekt. Wer sich für das Makefile für „Eden" interessiert, kann gern per Mail nachfragen.

nochmal in Hochkommata setzen. Die Datei kann man praktikablerweise auch `~/.inform` nennen, von selber aufgerufen wird sie deshalb aber nicht,

In Windows-Systemen wird man `inform.exe` und `winfrotz.exe`[10] wohl irgendwo unter `c:\Programme\` installieren; anschließend den Editor für die registrierten Dateiendungen im Dateiexplorer anwerfen und sicherstellen, dass die Endungen `.z5` und `.z8` für WinFrotz und die Endungen `.inf` und `.icl` für Inform (und natürlich noch für den auserwählten Editor) registriert sind; aber für `.inf` in der Weise, dass zwischen `inform` und dem Platzhalter für den Dateinamen eine Voreinstellungsdatei in Klammern steht, z.B: `(c:\Programme\inform\default.icl)` und für `.icl` in der Weise, dass die Befehlszeile die so aufgerufene Datei in Klammern enthält (Inform erkennt eine Kommandodatei nicht an der Endung, sondern daran, dass sie in Klammern angegeben wird).

Jetzt müssen wir noch `default.icl` schreiben (hier eine mögliche Unix-Version; angepasst werden muss auch für Windows nur der *include path*, d.h. das Verzeichnis, in dem die Bibliotheksdateien gesucht werden sollen)

```
+include_path=/usr/local/share/informlib-de/
+language_name=german
-sixE2d2v5
```

Wenn das erstellte Spiel dann so weit ist, dass es veröffentlicht werden kann, muss an die letzte Zeile noch ein `~D~S` angehängt werden, um das Debuggingvokabular aus dem Spiel zu nehmen.

1.3.1 Hallo, Compiler!

Jetzt wird es Zeit, unsere Installation auszuprobieren; von der Unix-Befehlszeile aus etwa mit

```
inform '(~/default.icl)' hallo.inf
```

Wenn bei der Windows-Installation alles passt, müsste es ausreichen, die Datei `hallo.inf` im Datei-Explorer doppelzuklicken.

```
! hallo.inf

! dies durch den Compiler gedreht, müsste "hallo.z5" erzeugen ...
! ... und z.B. "frotz hallo.z5" gibt uns dann den Begrüßungstext aus.
```

[10] Vorsicht: Alte Versionen von WinFrotz können mit deutschen Sonderzeichen nicht korrekt umgehen. Benutzen Sie Frotz 2002 für Windows.

```
[ Main;
     print "Grüß Gott, ich bin der Compiler und habe soeben laufen
            gelernt.^";
];
```

An diesem Beispielprogramm[11] sehen wir schon einige wichtige Dinge:

Kommentare beginnen mit einem Ausrufungszeichen `!` und gehen bis zum Zeilenende.

Anweisungen müssen mit einem Semikolon `;` abgeschlossen werden.

Routinen bzw. Unterprogramme stehen in eckigen Klammern `[ ]`. Ihr Name ist ihre erste Anweisung. (Das mit der Routine `Main` funktioniert wie in C, d.h. sie repräsentiert das Hauptprogramm und wird nach dem Programmstart automatisch aufgerufen. Da `Main` in der Library vordefiniert ist, braucht man sie normalerweise nicht explizit anzugeben.)

Strings (Zeichenketten) werden in Anführungszeichen `"` eingeschlossen. Sie können über das Zeilenende hinaus fortgesetzt werden; denn erst das Caret (zu deutsch „Dächle") `^` steht für eine Zeilenschaltung im String.

`print` ist ein Befehl, mit dem Strings ausgedruckt werden können. (Da `print` das Häufigste ist, was man mit einem String in Inform machen kann, könnte man `print` an dieser Stelle auch weglassen, der String würde dennoch ausgegeben. Allerdings hätte der Befehl dann einen anderen Rückgabewert. Rückgabewerte interessieren uns aber hier – noch – nicht.)

Das war aber noch kein bisschen interaktiv. Das Programm versteht ja noch kein Deutsch, dazu brauchen wir die Library.

1.3.2 Library und Grundgerüst

Die Dateien der Library müssen in einer festen Reihenfolge eingebunden werden; dies führt dazu, dass jedes Spiel gewissermaßen ein Grundgerüst haben muss:

```
! Template-Datei für deutsches Inform: template.inf

Constant Story "[Name bzw. Titel des Spiels]";
Constant Headline "^[Untertitel, Autorenangabe etc.]^";
Constant R_NEU;

! Vor dem Benutzerprogramm muss der erste Teil der Library eingebunden werden:
```

[11] Wer nördlich der Mainlinie wohnt, kann statt „Grüß Gott" gerne „Hallo" schreiben, dann weiß er aber nicht, ob der deutsche Zeichensatz richtig funktioniert.

```
Include "parser";
Include "verblib";

! Hier beginnt das Spiel
Object Startraum "[Der Raum hat einen Namen]"
   with description
      "[Und hier steht die Beschreibung des Raums.]",
   has light;

! Hier ist Platz für das restliche Programm :-)

! Die Library ruft die Hauptschleife "Main" selbsttätig auf,
! wir müssen nur ein paar Initialisierungen machen:

[ Initialise;
  location = Startraum;
  "^^^[Hier folgt ein Einleitungstext, der am Spielanfang ausgegeben
   wird.]^";
];

! Unterprogramme folgen jetzt:

! Erst nach dem Benutzerprogramm kommt der zweite Teil der Library:
Include "germang";

! Und zum Schluss Ergänzungen zur Grammatik:
```

Wenn das ohne Fehlermeldung durchläuft[12] und die entstehende `.z5`-Datei sich mit einem Interpreter öffnen und "spielen" lässt, ist auch die Library richtig installiert und die Arbeit kann beginnen.

[12] Prüfe bei Fehlern zuerst, ob nicht durch die Formatierung das Ende irgendeiner Kommentarzeile in die nächste Zeile gerutscht ist!

Wird unter Linux eine ältere Version der Bibliotheksdateien verwendet (Versionsnummer<9), gibt es Probleme mit der Groß/Kleinschreibung der Dateinamen. Lösung: bei textfire.de die aktuellsten Bibliotheksdateien herunterladen oder die Dateinamen umbenennen.

2 Schritt für Schritt ins Paradies: Eine Einführung in die Programmierung

Es gibt verschiedene Ansichten darüber, wie man ein Inform-Programm entwerfen soll. Die einen beginnen wie Tolkien mit einer Landkarte, verteilen auf ihr die Objekte und die Rätsel und haben allenfalls ein grobes Konzept der Story. Andere schreiben zuerst ein Skript mit allen Möglichkeiten und Alternativen des Spielablaufs und beginnen erst dann mit dem Entwurf von Objekten.

Ich werde einfach mit ein paar Objekten beginnen. So habe ich es mit „Eden" auch gemacht. Allerdings ist das Spiel weder groß, noch ist die Story allzu elaboriert.

2.1 „Und Gott der Herr pflanzte einen Garten in Eden gegen Morgen . . . "

2.1.1 Ein Raum, ein Baum (Objekte, Namen, Beschreibungen)

Wir gehen aus vom bereits bekannten Grundgerüst (siehe 1.3.2 auf Seite 12). Der Titel (und der Untertitel) werden jeweils als Textkonstante definiert:

```
Constant Story "Eden";
Constant Headline "^ein interaktiver Anfang  ((c) frob 2002)^";
```

Statt des Startraums beginnen wir mit etwas Ansprechenderem:

```
Object Mitte "Inmitten des Gartens"
 with  description
       "Eine Lichtung inmitten eines großen Gartens.
        Ringsum sind Beete, Obstbäume, Gebüsch;
        alles ist gepflegt und feierlich. Aus einer Quelle sprudelt
        frisches Wasser und fließt in vier verschiedene Richtungen
        ab. Neben der Quelle steht ein einzelner, majestätischer
        Baum -- der Baum der Erkenntnis.",
 has   light;
```

Das ist die Minimalausstattung eines Raums – ein Name, eine Beschreibung und Licht, damit man die Beschreibung auch sieht.

Wie alles, mit dem der Spieler in Berührung kommt, ist auch der Raum, in dem er sich bewegt, ein Objekt, deklariert mit dem Schlüsselwort `Object`, gefolgt von einem Bezeichner (dem Namen des Objekts, den das Programm verwendet, hier `Mitte`) und einem String (einer Zeichenkette), in dem der Name des Objekts steht, wie ihn der Spieler zu sehen bekommt. Nach diesem „Kopf" kommt der „Körper" des Objekts; er besteht aus *properties* (Eigenschaften) nach dem Schlüsselwort `with` und *attributes* (Attributen) nach dem Schlüsselwort `has`.

Attribute sind nichts anderes als boolesche (logische) Werte. Sie bezeichnen Eigenschaften, die das Objekt entweder hat oder nicht hat: `light` zeigt an, dass das Objekt beleuchtet ist (bei einem Raum; bei einem anderen Gegenstand zeigt es an, dass dieser Licht abgibt).

Properties sind zur Aufnahme größerer Datenmengen bestimmt, sie enthalten meist einen (auszugebenden) String oder eine Routine (ein Unterprogramm). `description` enthält die Beschreibung eines Objekts (bei Räumen angezeigt nach „sieh dich um", bei anderen Objekten angezeigt auf den Befehl „Untersuche den Gegenstand").

Die einzelnen *properties* und die Liste der Attribute werden voneinander durch Kommata getrennt; ein Semikolon schließt die Definition eines Objektes ab. Wie im nächsten Beispiel zu sehen ist, steht *zwischen mehreren Attributen kein Komma*, nur Leerraum.. Auch das einleitende Schlüsselwort jeder *property* und ihr Inhalt sind nur durch Leerraum getrennt.

Der Raum ist noch etwas leer, also lassen wir einen Baum wachsen:

```
Object -> Apfelbaum
 with   name 'Baum' 'Erkenntnis' 'Apfelbaum' 'majestaetisch',
        short_name "Baum",
        post "der Erkenntnis",
        dekl 1,
        description "Es ist eine Art Apfelbaum, aber kein
            normaler Apfelbaum. Er steht in Blüte und trägt
            zugleich Frucht. Du weißt, dass es der
            ~Baum der Erkenntnis~ ist - aber du weißt nicht,
            was dieser Name bedeuten soll.",
 has    scenery male;
```

Mit dem Pfeil `->` wird angezeigt, dass der Apfelbaum nicht selbständig in der Welt steht, sondern zum vorher definierten Objekt (`Mitte`) gehört, d.h. in diesem Raum zu

finden ist. Auf diese Art und Weise werden Objekte anderen Objekten zugeordnet, d.h. ihr „Ort" in der Welt des Spiels festgelegt. Um dies schneller beschreiben zu können, werden Begriffe aus dem Wortfeld des Stammbaums verwendet: `Apfelbaum` ist ein `child` (Kind) von `Mitte`, `Mitte` ist ein `parent` (Elternteil) von `Apfelbaum`. Ein Objekt ohne `parent` ist entweder ein Raum oder gerade nicht im Spiel.

Die Zuordnungen beweglicher Objekte (und der Spielfigur) können im Lauf des Spiels wechseln. Die Spielfigur und andere Objekte bewegen sich durch die Spielwelt. Funktionen wie `child(Objekt)` und `parent(Objekt)` stehen der Autorin zur Verfügung, um den aktuellen Stand ermitteln zu können (mehr darüber unten in 3.1.2 auf Seite 68).

Im Unterschied zum Raum ist der Baum für den Spieler ein körperlicher Gegenstand: Er kann ihn z.B. anfassen, beriechen, versuchen auf ihn zu klettern oder ihn auszureißen. Dazu muss er ihn benennen können, d.h. einen Namen für ihn haben, den das Spiel versteht. Dazu ist die *property* `name` da: in ihr stehen – in Hochkommata eingeschlossen – die Namen, mit denen der Spieler das Objekt bezeichnen kann. Dazu zählen auch Adjektive (der Spieler kann sie verwenden, der Parser beachtet den Unterschied zwischen Adjektiv und Substantiv nicht, was regelmäßig nicht stört). Nur die ersten 9 Zeichen eines Namens sind signifikant, dabei zählen Umlaute und ß als zwei Zeichen, denn sie müssen an dieser Stelle ausgeschrieben werden[1]. Statt `'majestaetisch'` hätte also `'majestaet'` genügt.

Der Spieler weiß, dass es „der Baum, des Baums" etc. heißt, das Programm weiß es nicht. Es braucht daher Regeln, mit denen es den Namen des Objekts grammatisch richtig ausgeben kann. Es muss wissen, welche Teile des Namens es deklinieren muss („Baum") und welche nicht („der Erkenntnis"). Es muss das Deklinationsschema für „Baum" und das grammatische Geschlecht kennen. Das ist alles in der Objektdefinition enthalten:

`short_name` ist der „eigentliche" Name des Objekts. Er wird dekliniert.

`dekl` gibt das zu verwendende Deklinationsschema an (es gibt 9 davon, siehe 3.2.7 auf Seite 96).

`post` sind nachgestellte Namensbestandteile, die nicht dekliniert werden.

`male, female` **oder** `neuter` sind Attribute, die das grammatische Geschlecht festlegen.

Außerdem gibt es noch

`adj` **(property)** Eine Liste von Adjektiven, die dem `short_name` vorangestellt und dekliniert werden (z.B. `"einzeln" "majestätisch"`)

`pluralname` **(Attribut)** Das Objekt ist mehrere Gegenstände (steht also im Plural), z.B. „zwei Abflussrohre".

[1] Es handelt sich bei diesen Namen nicht um simple Zeichenketten (in doppelten Anführungszeichen), sondern um einen besonderen Datentyp: *Wörterbucheinträge* (in Hochkommata). Sie werden in einer besonderen Tabelle abgelegt, um das Vergleichen zu beschleunigen. Wir brauchen sie hauptsächlich in `name` und später beim Definieren neuer Verben (siehe 2.13). Das Ausschreiben der Umlaute und des ß ist nur in solchen Wörterbucheinträgen erforderlich.

Merke: Einzig `name` bestimmt, wie das Spiel auf die Eingabe des Spielers reagiert; alle anderen eben beschriebenen Attribute und Eigenschaften steuern nur, wie der Name des Objekts vom Spiel ausgegeben wird. Betritt der Spieler einen Raum oder sieht er sich dort um, erhält er nicht nur eine Raumbeschreibung, sondern auch eine Auflistung, welche Objekte sich dort befinden. Das ist nicht immer erwünscht; z.B. ist der Apfelbaum ja schon in der Raumbeschreibung erwähnt. Das Attribut `scenery` bewirkt, dass der Apfelbaum als Kulisse behandelt und nicht gesondert erwähnt wird. Außerdem bewirkt es, dass der Apfelbaum an dieser Stelle „fest" eingebaut wird; der Spieler kann ihn nicht an sich nehmen oder wegschieben. Letzteres allein kann auch mit dem Attribut `static` erzielt werden.

2.2 „. . . und setzte den Menschen drein . . . "

Am Anfang muss die Spielfigur in einem bestimmten Raum (hier: in der `Mitte`) eingesetzt werden. Dies erledigt die Routine `Initialise()`. Im Grundgerüst war sie schon vorhanden; wir müssen sie nur noch anpassen:

```
[Initialise;

    location = Mitte;

    "^^^Was vorher gewesen ist? Du weißt es nicht.
     Du kamst hier, in diesem Garten, zu Bewusstsein.
     Es ist schön hier. Dir fehlt es an nichts.
     Du bist glücklich. ^^^";
];
```

Routinen werden in eckige Klammern eingeschlossen und wie andere Definitionen mit einem Semikolon beendet. Der erste Bezeichner (hier: `Initialise`) ist der Name der Routine. `location` ist eine vordefinierte Variable, die den Ort kennzeichnet, in dem sich der Spieler gerade aufhält; ihr wird der Wert Mitte zugewiesen und so der Anfangsort festgelegt.

Anschließend wird üblicherweise eine Startmeldung ausgegeben; wir benutzen hier eine für Inform typische Kurzschreibweise und lassen das `print` vor dem auszugebenden String weg. [2]

Die Startmeldung bekommt der Spieler am Anfang zu sehen; noch vor dem Titel des Spiels.

[2] Nota bene: Das hat den durchaus heftigen Nebeneffekt, dass die Routine an dieser Stelle zu Ende ist. Da aber ohnehin nichts anderes mehr kommt, ist der Nebeneffekt egal. Im Detail dazu im nächsten Abschnitt.

2.3 Und was hängt an dem Baum? (`before` und `after`)

Erwartungsgemäß hängt ein Apfel am Baum; aber es ist ein ganz spezieller Apfel: Wer davon isst, der gewinnt die Erkenntnis. Es ist aber gar nicht so einfach, ihn zu essen: zuerst muss man ihn haben. Und es ist verboten, ihn zu nehmen! Wer es aber zweimal unmittelbar hintereinander versucht, hat Erfolg[3]:

```
Object -> Apfel
 with  name 'Apfel' 'verboten' 'Frucht' 'gross' 'schillernd' 'rund',
       short_name "Apfel",
       adj "verboten",
       dekl 2,
       initial "Am Baum hängt ein großer Apfel.",
       description "Das also ist die verbotene Frucht:
                    ein auffallend großer, runder Apfel.
                    Er schillert in der Sonne, seine
                    Farbe ist schwer auszumachen.",
       before [;
                take: if (self has general)
                          print "Du fasst dir ein Herz und ... ";
                      else
                      {
                          give self general;
                          "Du lässt die verbotene Frucht
                           lieber am Baum hängen.";
                      }
              ],
       after  [;
                take: "pflückst den Apfel mit zitternden Fingern.";
                eat:  give Adam Erkenntnis;
                     "Du isst den Apfel vom Baum der Erkenntnis --
                      Gedanken strömen auf dich ein wie ein Fluss,
                      der über seine Ufer tritt.
                      Du bist einen Augenblick lang benommen.^^
                      Du hast etwas Verbotenes getan.
                      Du bist böse. Und du bist nackt."
              ],
       react_before [;
                take: if (noun~=self)
                         give self ~general;
                default: give self ~general;
              ],
 has   male edible;
```

[3] Der „originale" Apfel aus „Eden" verhält sich anders.

`Initial` ist eine Kurzbeschreibung, die anstelle des normalen „Du siehst hier einen Apfel" ausgegeben wird, solange der Spieler den Apfel noch nicht genommen hat (d.h. solange er das in diesem Fall automatisch vergebene Attribut `moved` nicht besitzt).[4]

`edible` ist ein Attribut, das den Apfel essbar macht (man muss ihn aber zuerst haben!)

`general` ist ein „allgemeines" Attribut, das der Autorin zur Benutzung offen steht – zu Beginn ist es immer ungesetzt; oft wird es gesetzt, sobald ein mit dem Objekt verbundenes Rätsel gelöst ist. Hier verwende ich es, um zu markieren, ob der Spieler bereits einmal versucht hat, den Apfel zu nehmen; der zweite Versuch wird dann erfolgreich sein.

`self` ist ein Platzhalter, der in Code innerhalb einer Objektdefinition das jeweilige Objekt selbst meint; statt `self` könnte im obigen Beispiel überall auch `Apfel` stehen.

`has` ist an dieser Stelle ein logischer Operator, der testet, ob ein Objekt ein bestimmtes Attribut hat oder nicht.

`if` und `else` ermöglichen es, abhängig von einer Bedingung `(self has general)` entweder einen (hier: `print "Du fasst dir ein Herz und ... ";`) oder einen anderen `{`den in geschweiften Klammern nach `else` folgenden`}` bestimmten Teil des Programms auszuführen. Wie das genau funktioniert, ist beschrieben in 3.1.3 auf Seite 69. Es kann nicht schaden, gleich auch den folgenden Abschnitt über `switch` zu lesen, denn die Anweisungen in `before` usw. folgen derselben Syntax wie dort beschrieben.

`print` ist der Befehl, die nachfolgende Zeichenkette (oder mehrere, mit Komma getrennte) zu drucken und nichts weiter zu tun. Will man danach eine neue Zeile haben, so kann man die auszudruckende Zeichenkette mit `^` enden lassen oder den Befehl `new_line` anhängen. Dies darf keinesfalls verwechselt werden mit `print_ret`.

`print_ret` druckt ebenfalls den oder die nachfolgenden String(s) aus und läßt eine Zeilenschaltung folgen. Damit aber nicht genug, **es enthält *außerdem* noch die abschließende Befehlsfolge `return true.`**

`return` bewirkt einen *Rücksprung* aus dem `[;` in eckige Klammern eingeschlossenen`];` Programmblock. `return` kann von einem Rückgabewert gefolgt sein; die häufigst benötigten Rückgabewerte sind die logischen Werte `true` und `false.` Für ein `return` mit diesen Rückgabewerten gibt es die Kurzschreibweisen `rtrue` bzw. `rfalse`.

[4] Tatsächlich hängt der Apfel nur aus Sicht des Spielers – der diesen Initial-Text zu lesen bekommt – *am* Baum. Aus Sicht der Spielerfigur und der Autorin hängt der Apfel freischwebend daneben (er ist nicht Unterobjekt des Baums, sondern Unterobjekt desselben Raums, in dem auch der Baum steht). Das hat den Vorteil, dass der Spieler nicht erst den Baum durchsuchen oder aufmachen muss, um den Apfel zu finden; aber es hat auch den Nachteil, dass das Nachmodellieren des Befehls „Nimm den Apfel vom Baum" nachher noch Arbeit machen wird (2.13.3 auf Seite 54).

`"..."` als selbständiger Befehl (also nicht nach einer Zuweisung oder nach einem `print`) steht tatsächlich für die häufig benötigte Kombination `print "..."; newline; return true;`[5]

`Take` und

`Eat` stehen für bestimmte *Aktionen*; hier die Aktionen, die ausgelöst werden, wenn der Spieler versucht, den Apfel zu nehmen oder ihn zu essen. Eine Übersicht über die vordefinierten Aktionen und die *„Verben"*, d.h. die Eingaben des Spielers, die die Aktionen auslösen, enthält Kapitel 4 auf Seite 106.

`before` ist der Platz für eine Routine (in eckigen Klammern, aber ohne Namen, deshalb das Semikolon unmittelbar nach der öffnenden Klammer); die auf Aktionen reagiert, die der Spieler mit dem Apfel anstellt. Gibt er den Befehl „Nimm den Apfel", so wird die Routine in der `before`-*property* des Apfels aufgerufen und die auszuführende Aktion auf den Wert `Take` gesetzt. Die Routine kann für jede Aktion einen eigenen Zweig enthalten, in dem festgelegt wird, wie der Apfel auf die Aktion des Spielers reagiert. Im Beispiel ist nur ein Zweig für `Take` enthalten. (`Eat` wird von der Library abgewiesen, wenn der Spieler den Apfel nicht hat.)

Das Programm prüft nun das Attribut `general`; ist es gesetzt, gibt es mit `print "..."` einen einleitenden Text aus und macht nichts weiter. Die Routine gibt – wenn nicht ausdrücklich etwas anderes angegeben ist – den Wert `false` zurück; die Library überprüft, ob der Apfel genommen werden kann und wird ihn dem Spieler geben, weil nichts entgegensteht. Ist aber `general` nicht gesetzt – und zu Beginn ist es das nicht – dann wird eine andere Nachricht ausgegeben (mit der Kurzschreibweise `"..."`, die auch für ein `return true;` am Ende steht!) und die Routine gibt `true` zurück. Das bedeutet aber für Inform soviel wie „Alles fertig – Schluss mit der Aktion", der Zug ist zu Ende, der Apfel bleibt wo er war. (Vorher haben wir allerdings mit dem Befehl `give self general;` das Attribut für den nächsten Versuch gesetzt.)

Auch `Erkenntnis` ist ein Attribut, das gesetzt wird, wenn der Spieler (repräsentiert durch das Objekt `Adam`, wie wir noch sehen werden) den Apfel isst.

`after` funktioniert nach demselben Prinzip wie before, wird aber erst aufgerufen, nachdem die Library geprüft hat, ob die Aktion ausgeführt werden kann und sie auch ausgeführt hat (z.B. bei `Take` den Apfel zum `child` des Spielers gemacht hat oder bei `Eat` den Apfel hat verschwinden lassen). Ein `false` als Rückgabewert bedeutet, dass die Library nun auch ihren vordefinierten Standardtext für die Aktion ausgeben soll; ein `true` als Rückgabewert bedeutet wieder „Alles fertig – Schluss mit der Aktion".

Im Beispiel werden – für ein erfolgreiches `Take` oder `Eat` – Texte mit dem Befehl `"..."` ausgegeben, der ein implizites `return true;` enthält. Nach diesem Text ist also in beiden Fällen Schluss mit der jeweiligen Aktion.

[5] Bitte lesen Sie nochmal die obige Definition: An keiner Stelle folgt nach einem eigenständigen `"..."`-Befehl noch irgend etwas anderes. (Stünde etwas dort, würde es wegen des enthaltenen `return` nie ausgeführt.)

`react_before` funktioniert (ebenso wie eine weitere *property* mit dem Namen `react_after`) in gleicher Weise; nur wird sie nicht nur bei Befehlen aufgerufen, in denen der Apfel direktes Objekt ist („nimm den Apfel") sondern bei allen Befehlen, die im Blickfeld des Apfels[6] stattfinden.

Im Beispiel wird sie benutzt, um das Attribut `general` wieder zurückzusetzen (man beachte die vorangestellte Tilde!), wenn der Spieler einen anderen Befehl gibt als „nimm den Apfel". Dies bewirkt, dass der Apfel nur genommen werden kann, wenn der Spieler seinen Befehl „nimm den Apfel" zweimal unmittelbar nacheinander erteilt.

`~` Die einfache Tilde wird von Inform als Modifikator verwendet. Unmittelbar vor ein Attribut gesetzt, verkehrt sie es ins Gegenteil. **Nota bene:** Der Operator für logisches NOT ist die doppelte Tilde `~~`. Weil wir gerade dabei sind:

`~=` ist ein logischer Operator, der zwei Objekte auf Ungleichheit testet, der Gleichheitsoperator ist logischerweise `==`. Zuweisungen erfolgen mit einem einfachen Gleichheitszeichen. Für eine vollständige Liste der Operatoren siehe 3.1.1.

`noun` ist eine Variable, die das direkte (erste) Objekt des aktuellen Befehls enthält, bei „nimm den Apfel" also den Apfel.

`default` ist ein Schlüsselwort, das für jede andere Aktion steht.

2.4 „... allerlei Bäume, lustig anzusehen ..."

2.4.1 Räume und Bäume aller Klassen.

Die Mitte des Gartens haben wir; jetzt muss aber noch etwas Landschaft außen herum. Weil der Garten überall irgendwie ähnlich aussieht, definieren wir gemeinsame Eigenschaften aller Gartenteile in einer *Klasse*:

```
Class Garten
 with short_name "Im Garten Eden",
      description "Du bist im Garten Eden. Ringsum sind Beete,
                   Obstbäume, Gebüsche; alles ist gepflegt und
                   feierlich. ",
      in_to "Du bist bereits im Garten Eden.",
      u_to Mitte,
      cant_go "Du gehst einige Schritte im Garten umher.",
 has  light;
```

[6] genauer: die stattfinden, während der Apfel im Blickfeld des Spielers ist.

Mit dem Schlüsselwort `Class` wird kein Objekt definiert, sondern eine Art Vorlage für mehrere ähnliche Objekte. Darauf aufbauend können mit wenig Zeilen die tatsächlichen Objekte geschrieben werden, hier z.B der östliche Gartenteil:

```
Object EdenE
 class Garten
 with  s_to EdenSE,
       n_to EdenNE,
       sw_to EdenS,
       nw_to EdenN,
       w_to Mitte;
```

Mit der Zeile `class Garten` „holen" wir alle in der Klasse definierten Eigenschaften (Attribute und *properties*) in unser neues Objekt, ohne sie ein zweites Mal schreiben zu müssen.[7]

Die *properties* auf `_to` legen fest, was passiert, wenn man aus einem Raum in eine bestimmte Richtung (eine der acht Kompasshaupt- und Nebenrichtungen `n, w, s, e` (steht für Osten!)[8], `nw, sw, ne, se, in, out, u` (hinauf) und `d` (hinunter) geht: Steht darin der Name eines anderen Raumobjekts, dann geht es dorthin; steht ein String darin, wird der String ausgegeben und der Spieler bleibt, wo er war. Die *property* `cant_go` hat dieselbe Funktion für jede Richtung, die das Objekt nicht ausdrücklich definiert.

Damit die Bewegung in beide Richtungen funktioniert, muss sie in den Objekten auf „beiden Seiten" festgelegt sein. Das heißt hier, dass auch die Definition von Mitte erweitert werden muss, damit man von dort wirklich in alle Richtungen gehen kann:

```
w_to EdenW,
sw_to EdenSW,
s_to EdenS,
se_to EdenSE,
e_to EdenE,
ne_to EdenNE,
n_to EdenN,
nw_to EdenNW,
```

Eine in der Klasse vordefinierte *property* kann in der Objektdefinition problemlos überschrieben werden. Bestimmte *properties* (z.B. `before` und `after`) sind aber *additiv*, d.h. dass zuerst die in der Objektdefinition angegebenen Regeln abgearbeitet werden,

[7] Dieser Vorgang wird üblicherweise „Erben" genannt (das Objekt erbt alle Eigenschaften der Klasse), was aber eigentlich nicht passt, da die Klasse damit nicht etwa tot ist, sondern beliebig oft verwendet werden kann.

[8] Der Name EdenE ist willkürlich; ich habe den um die Mitte liegenden Gartenteilen solche Namen gegeben, um den Überblick zu behalten.

wenn keine davon ein `true` zurückgegeben hat, werden noch die Regeln der Klassendefinition durchgeprüft.

Bei Attributen gibt es eine Besonderheit: `male` und `female` überschreiben einander und `neuter`; `neuter` überschreibt die beiden anderen aber nicht (man muss also `~male neuter` schreiben, um im Objekt ein von der Klasse geerbtes `male` auszuschlagen).

Weil es so schön ist, folgen noch drei Klassen:

```
Class Baum
with  name 'baum' 'baeume' 'obstbaum' 'obstbaeume',
      dekl 1,
      description "Die Bäume sind lustig anzusehen.
                   Sie tragen Früchte.",
      before [;
               lift, push, pull, empty, burn, cut, squeeze:
                "Du sollst den Garten hüten und nicht in ihm
                 randalieren.";
               search: <<examine self>>;
               climb: "Du hörst die Stimme eines Engels:
                       ~Denk' an den Anteil der Arbeit an der
                       Menschwerdung des Affen!~ und überlegst
                       dir, dass es jetzt noch etwas zu früh ist,
                       zurück auf die Bäume zu klettern.";
             ],
has   scenery male;
```

Neu ist hier der Befehl `<<examine self>>`; es handelt sich wieder um eine Kurzschreibweise; nämlich für die Befehlsfolge `<examine self>; return true;` . Mit `<examine self>` (in einfachen spitzen Klammern) wird eine Aktion ausgelöst und derselbe Ablauf bewirkt, als hätte der Spieler selbst „Untersuche den Baum" eingegeben.

Der vorhin definierte Apfelbaum sollte auch mit der Zeile `class Baum` das allgemeine Verhalten eines Baums vererbt bekommen. (Die `description` überschreibt er allerdings.)

```
Class Obst
with  name 'Frucht' 'Fruechte' 'Obst',
      short_name "Frucht",
      dekl 7,
      before [;
               examine:
                 print_ret
                   (GDer) self, " sieht lecker aus.
                    Du bekommst Appetit.";
               eat, remove, transfer, take, taste:
```

```
                    print_ret
                     "Du pflückst ", (einen) self, ". ",
                      (GEr) self, " ist so appetitlich, dass du
                      nicht widerstehen kannst und ", (ihn) self ,
                      " auf der Stelle verspeist. ", (GEr) self,
                      " ist gut zu essen und schmeckt köstlich.";
                ],
 has  female;
```

und

```
Class Blume
 with short_name "Blume",
      dekl 9,
      name 'bluete' 'blume' 'blueten',
      before [;
        smell: print_ret (GDer) self, " duftet angenehm.";
        touch: "Du fühlst eine zarte Pflanze.";
      ],
      initial [; print_ret (GEin) self, " blüht auf der Wiese.";],
 has  female;
```

2.4.2 Ausgeben von Texten

Der Befehl `print_ret` ist ebenso wie das schon bekannte `"..."` eine weitere Kurzschreibweise für – wer weiß es noch? – `print "..." ; new_line; return true;`

Wir verwenden hier `print_ret` statt `"..."`, weil nicht nur einfach ein String, sondern ein ganzer Satz, bestehend aus mehreren Strings und Ausgaberegeln ausgegeben werden soll (die einzelnen Elemente können in beliebiger Anzahl und Reihenfolge hinter `print_ret` folgen, sie werden durch Kommata getrennt, am Schluss folgt wie immer ein Strichpunkt). Da bei der zweiten Ausgabeanweisung als erstes Element ein in Anführungszeichen geschriebener String kommt, könnte dieser auch in seiner Funktion als Kurzschreibweise für `print_ret` benutzt und also das zweite `print_ret` einfach weggelassen werden.

Ausgaberegeln sind in Klammern geschriebene Funktionsnamen, gefolgt von einem Objekt. (Für eine Übersicht der Ausgaberegeln siehe 3.2.5 auf Seite 89.) Sie bewirken, dass das Objekt mit dem passenden Artikel bzw. passenden Pronomen ausgegeben wird: `(GDer) self` gibt den bestimmten Artikel und den Objektnamen im Nominativ aus, es wird für den Apfel zu „Der Apfel", für die Birne zu „Die Birne" und für das Johannisbrot zu „Das Johannisbrot" (das G am Anfang steht für Großschreibung).

2.4.3 Von der Klasse zum Objekt

Mit diesen Klassen läßt sich der Garten so flott ausstatten, als wären es in Zaubertrank gefallene Eicheln, bzw. Carobs:

```
Object  Johannisbrot EdenE
 class  Obst
 with   short_name "Johannisbrot",
        dekl 1,
        initial "Hier stehen die Johannisbrotbäume.",
        name 'johannisb' 'brot' ,
 has    ~female neuter;
```

bzw.

```
Object  Primel EdenW
 class  Blume
 with   short_name "Primel",
        name 'primel' 'schluessel' 'bluete' 'blueten' 'winzig' 'gelb' 'klein',
        description "Eine kleine Primel, auch Schlüsselblume genannt. Sie hat
                     viele winzige gelbe Blüten.";
```

Beachte: Hier haben wir keinen Pfeil `->` verwendet, um den Ort des Objekts im Spiel festzulegen. Statt dessen ist der *parent* des Objekts – `EdenE` – explizit im Kopf der Definition angegeben. Diese Schreibweise ist weniger fehleranfällig und deshalb vorzuziehen.

2.5 Das Betreten des Rasens ist erlaubt

Im Garten Eden gibt es natürlich paradiesische Liegewiesen.

```
Object  Gras
 with   short_name "Gras",
        dekl 4,
        description
            "Starke, kurze Halme bilden einen dichten Grasteppich.
             Allerlei Getier tummelt sich darin.";
        before [;
         take: "Du pflückst einen Grashalm,
                auf dem ein Käfer sitzt. Du weißt nichts Rechtes
                damit anzufangen und wirfst ihn wieder weg.";
         LookUnder: "Es ist keine Falltür drunter.";
```

```
        ],
        after [;
         enter: "Du streckst dich auf dem Gras aus. Es ist weich und
                 angenehm.";
         exit: "Du stehst wieder auf.";
        ],
        name 'Gras' 'Graeser' 'Halm' 'Wiese' 'Rasen' 'Heu' 'Boden'
             'Teppich' 'Grasteppich' ,
        found_in Mitte EdenN EdenNW EdenSE EdenSW EdenNE EdenN
                 EdenS EdenE EdenW,
        capacity 100,
    has neuter scenery supporter enterable;
```

2.5.1 Begehbare Objekte

Dass man auf dem Gras liegen kann, bewirken die Attribute `supporter` und `enterable`:

Ein Objekt mit der Eigenschaft `supporter` kann mit anderen Objekten „belegt" werden – man kann andere Objekte darauf tun[9] – bis das mit `capacity` angegebene Fassungsvermögen erreicht ist – hier also 100 Objekte. (100 ist die Voreinstellung, so dass die Zeile hier schadlos weggelassen werden könnte. Für eine Liegewiese sind 100 Objekte ja kein Problem. Meist wird man realistischerweise einen niedrigeren Wert wählen.)

Ein Objekt mit der Eigenschaft `enterable` kann auch von der Spielerfigur betreten werden. Dass die Spielerfigur sich tatsächlich ins Gras *legt* (nicht nur draufsteigt oder sich daraufsetzt), ist nicht vorgegeben – diese Illusion bewirken einzig und allein die in der `after`-*property* angegebenen Texte für die Standardaktionen `##Enter` und `##Exit`.[10]

2.5.2 Schiebekulissen

Da das Gras überall gleich aussieht, reicht es völlig aus, im ganzen Spiel nur ein einziges Objekt dafür zu verwenden, das dem Spieler überall dorthin folgt, wo Gras wachsen sollte. Das Gras ist deshalb als Schiebekulisse *(floating object)* implementiert. Dafür sorgt die *property* `found_in`, in der die Namen aller dergestalt begrünten Räume des Spiels stehen. Betritt der Spieler einen neuen Raum, kontrolliert die Library alle `found_in`-*properties* und schiebt die benötigten Kulissen einfach hinterher.

[9] Wie wir im nächsten Abschnitt sehen werden, gibt es auch das Attribut `container` für Objekte, die andere Objekte enthalten können. Seltsamerweise gibt es kein vordefiniertes Attribut für Objekte, *unter* die man andere Objekte legen kann – obwohl in der überwiegenden Mehrzahl aller Spiele entscheidende Objekte (Schlüssel, Schriftrollen etc.) *unter* irgendwelchen anderen Objekten liegen.

[10] Die Inform-Syntax schreibt vor, den Namen von Aktionen zwei Gatter `##` voranzustellen. Davon darf in vielen Fällen abgewichen und es kann der Name der Aktion ohne Gatter geschrieben werden: namentlich in Dispatcherroutinen wie `before` und `after` , im Direktaufruf der Aktion mit spitzer Klammer und in der Definition von Grammatikzeilen (siehe 2.13 auf Seite 51). Das führt praktisch dazu, dass man die Gatter kaum je verwenden muss.

2.6 „. . . ein Strom, zu wässern den Garten. . . ”

Nach der Beschreibung befindet sich mitten im Garten eine Quelle, von denen Flüsse in vier Richtungen laufen. Hier greifen wir nicht zur Schiebekulisse, sondern definieren eine Klasse für die Eigenschaften der verschiedenen Gewässer. Flüssigkeiten sind immer etwas schwierig in der Handhabung, wie man sieht:

2.6.1 Wasser? Klasse!

```
Class  Wasser
 with  description "Das Wasser ist klar und rein. Du siehst bis auf den
                    Grund hinunter. Bunte Fische schwimmen darin umher.",
       name 'Bach' 'Wasser' 'Gewaesser' 'Grund',
       before [;
         search: <<examine self>>;
         lift: "Du spritzt etwas Wasser in die ohnehin schon feuchte
                Luft.";
         touch: "Du greift mit deinen Händen ins klare Wasser. Die
                Fische nehmen Reißaus.";
         take, eat, drink, taste : "Du schöpfst mit den Händen aus dem
                klaren Wasser und trinkst einige Schlucke. Das Wasser
                schmeckt erquickend.";
         smell: "Das Wasser riecht frisch.";
         transfer, empty, emptyT: "Das könnte Ewigkeiten dauern.";
         fill:  "Es ist schon genug Wasser darin";
         enter: if (child(player))
                {
                    print "Du legst erst alles ab, was du bei dir
                           trägst.^";
                    while (child(player))
                           move (child(player)) to limbo;
                }
       react_before [;
         fill: if (noun ~= self) "Du brauchst ein Behältnis für das Wasser.";
         Exit: if (player in self)
               {
                   print "Du gehst wieder an Land";
                   if (child(limbo))
                   {
                       while (child(limbo))
                              move (child(limbo)) to player;
                       print " und nimmst deine Sachen an dich.^";
                   }
                   else
```

```
                    print ".^";
                playerto (parent(self));
                return true;
             }
      swim: if (player notin self)
                <<enter self>>;
            else
                "Du schwimmst ein paar Züge. Das Wasser ist
                 erfrischend.";
      take, putOn, Wear, remove:
            if ((player in self) && (noun in limbo))
                "Du hast ", (den) noun , " doch erst am Ufer abgelegt,
                 damit ", (er) noun, " nicht nass wird.";
      ],
    after [;
      Enter: "Du gehst einige Schritte ins Wasser und lässt dich
              durch das kühle Nass erfrischen.";
      ],
    inside_description [;
      print "Du bist im Wasser.^";
      ],
    add_to_scope [ sache;
      objectloop (sache in limbo)
      {
            AddToScope(sache);
      }
     ],
has   scenery container enterable open;
```

2.6.2 Behälter und ihre Eigenschaften

Das Wasser ist ausgestattet wie ein Behälter – `container` – in den auch die Spielerfigur hinein kann – `enterable` – und zwar – `open` – ohne ihn vorher aufmachen zu müssen. Man kann Behälter auch verschließen und sogar zusperren; dazu aber später mehr beim Thema „Türen”. Solange der Behälter offen ist, ist fällt Licht aus dem Raum in ihn hinein; wenn er geschlossen ist, geht das nur dann, wenn er auch die Eigenschaft `transparent` besitzt. Die `capacity` habe ich hier weggelassen, also ist sie auf 100 gesetzt.

Ein Behälter hat auch eine Innenansicht; sie steht in der *property* `inside_description`. Diese Beschreibung ist hier so geschrieben (mit `print` statt `print_ret`), dass sie nicht `true`, sondern `false` zurückgibt. Dies bewirkt, dass die Beschreibung des Raums *und* die `inside_description` ausgegeben werden; stünde hier nur `"Du bist im Wasser."`, würde implizit `true` zurückgegeben und die Beschreibung des Raums weggelassen werden.

Wieder legt Inform nur fest, dass sich die Spielerfigur im Behälter befindet. Was sie dort macht, ob sie schwimmt, kocht oder eingefroren wird, entscheidet die Autorin.

2.6.3 `return`, richtig und falsch

Beim Lesen und Schreiben von Inform-Code ist es überlebenswichtig, sich immer darüber im Klaren zu sein, ob bzw. welche Codezeile einen Rücksprung mit `true` oder `false` auslöst:[11]

`return true, rtrue`	immer	`true`
`return false, rfalse`	immer	`false`
String in Anführungszeichen	in einer Zuweisung oder nach `print`	ohne Rücksprung
`print_ret`	immer	`true`
String in Anführungszeichen	als Kurzschreibweise für `print_ret`	`true`
Ende einer Routine `],`	unselbständig, innerhalb eines Objekts	`false`
Ende einer Routine `];`	selbständig, außerhalb eines Objekts	`true`

In einer `before`-property bedeutet ein Rücksprung mit `true` soviel wie: Ende der Aktion – nichts passiert weiter. Es ist gängiger Programmierstil, verschiedene logisch aufeinanderfolgende Abbruchkriterien nicht in *eine verschachtelte* `if`-Anweisung zu packen, sondern mehrere `if`-Anweisungen mit – bedingten – Rücksprungbefehlen aufeinanderfolgen zu lassen.[12]

2.6.4 Die Umkleidekabine oder das Verschieben von Child-Objekten

Eine Besonderheit ist die virtuelle unsichtbare Umkleidekabine. Sie ist zunächst nichts als ein freischwebendes Objekt ohne Eigenschaften:

```
Object limbo;
```

Vor dem Betreten des Wassers wird in der `before`-*property* die Aktion `##Enter` abgefangen: Wenn die Spielerfigur Besitztümer bei sich trägt – die Funktion `child(player)` also einen Wert ungleich 0 (`false`) zurückgibt – werden diese Stück für Stück mit einer `while`-Schleife und dem Befehl `move to` solange nach `limbo` verschoben, bis keine mehr da sind. Beim Verlassen des Wassers (Regel für die Aktion `##Exit` in der *property* `react_before`) erhält die Spielerfigur die in `limbo` geparkten Gegenstände auf die gleiche Weise zurück. Versucht er sie zu nehmen (Regel für `##take`), während er im Wasser ist, wird der Versuch zurückgewiesen (zur Erinnerung: `noun` ist das direkte Objekt des vom Spieler eingegebenen Befehls).

[11] Auf den ersten Blick sieht das so logisch aus wie ein Tarifplan der Bahn-AG. Es ist aber genial daraufhin optimiert, möglichst wenig explizite `return`-Befehle schreiben zu müssen. Ein Mathematiker ist ein Mensch, der lieber eine halbe Stunde nachdenkt, als fünf Minuten zu arbeiten.

[12] Beispiele etwa beim Code für die Tür: 2.7 auf Seite 32.

Beachte: die Regeln in `before` werden nur ausgeführt, wenn das Wasser direktes Objekt der Aktion ist („Gehe ins Wasser"); die Regeln in `react_before` werden bereits ausgeführt, wenn das Wasser im Blickfeld ist (die Spielerfigur das Wasser sehen kann). Einige der von `react_before` behandelten Aktionen sind nur sinnvoll, wenn die Spielerfigur tatsächlich im Wasser drin ist, die Abfrage `(player in self)` also `true` zurückliefert.

Beachte: die Programmstellen, an denen die Besitztümer de Spielfigur hin und zurück verschoben werden, enthalten keine Definition eines Rückgabewerts (mit `return true` oder `return false`). (Es handelt sich in beiden Fällen um mit eckigen Klammern `[ ]` eingeschlossene Routinen innerhalb einer Objektdefinition. Solche „unselbständigen" Routinen geben immer `false` zurück, wenn es nicht anders angegeben ist, s.o.) Das führt hier dazu, dass zwar die Besitztümer verschoben werden, der Code aber sonst nicht beeinflusst, wie die Spielfigur baden geht und welche Meldungen von anderen Programmteilen dabei ausgegeben werden.

`add_to_scope` und `objectloop`

Die abgelegten Gegenstände sind zwar ganz woanders, aber für den Spieler trotzdem sichtbar. Dies wird erreicht durch die *property* `add_to_scope`. Scope heißt soviel wie Reichweite oder Blickfeld und steuert, welche Objekte der Spieler sehen kann. Versucht er, z.B. ein Objekt zu untersuchen, das nicht im Blickfeld ist, erhält er die Rückmeldung: „Du kannst nichts dergleichen sehen."

`add_to_scope` steuert, welche Objekte zusammen mit dem Wasser automatisch im Blickfeld sind – egal wo sie sich tatsächlich befinden. Ginge es nur um einen oder mehrere *immer gleiche* Gegenstände, würde es ausreichen, einfach die Namen der Objekte hier aufzuführen. Da wir im voraus nicht wissen können, welche Gegenstände der Spieler bei sich haben wird, wählen wir einen anderen Weg: Für jeden ins Blickfeld zu rückenden Gegenstand wird die Funktion `AddToScope(Gegenstand)` einmal aufgerufen.

Mit der oben verwendeten `while`-Schleife hätten wir hier kein Glück: Ihre Funktionsweise beruht darauf, dass sie die *Child*-Objekte nacheinander entfernt und von selbst aufhört, wenn keines mehr da ist. Macht nichts – `objectloop` ist ohnehin eleganter, um eine Schleife über alle Objekte mit einer bestimten Eigenschaft laufen zu lassen. Zur Syntax bitte nachschlagen: 3.1.3 auf Seite 71. Dort steht auch, warum wir nicht *gleich* `objectloop` für alle Schleifen verwendet haben.

2.6.5 Vom Wasser zum Gewässer

Eine Klasse ist für den Spieler noch nicht sichtbar, wir brauchen ein von ihr abgeleitetes Objekt. Eine Quelle ist schnell herbeigezaubert:

```
Object  Quelle Mitte
 class  Wasser
 with   dekl 9,
        name 'Quelle',
        short_name "Quelle",
 has    female;
```

Vier Bäche? Das ist Luxus; wir brauchen nur einen, den wir wieder als Schiebekulisse ausgestalten:

```
Object  Bach
 class  Wasser
 with   short_name "Bach",
        name 'bach' 'fluss',
        adj "klar",
        dekl 1,
        initial "Ein klarer Bach fliesst durch diesen Teil des
                 Gartens.",
        found_in EdenNW EdenSW EdenNE EdenSE,
 has    male;
```

2.6.6 Vom Gewässer zur Landschaft

Dort, wo ein Bach fließt, sieht der Garten anders aus. Wir definieren eine Klasse `GartenW`, die auf der oben schon eingeführten Klasse `Garten` aufbaut:

```
Class   GartenW
 class  Garten
 with   richtung "[Fehler: Klasse GartenW.richtung geerbt, aber nicht
                  definiert]",
        description [;
           print_ret "Du bist im Garten Eden. Ringsum sind Beete, Obstbäume,
                      Gebüsche; alles ist gepflegt und feierlich. Ein klarer
                      Bach fließt durch diesen Teil des Gartens in ",
                     (string) self.richtung,
                     " Richtung.";
        ];
```

Hier definieren wir „aus dem Stand" eine neue *property* – `richtung` – indem wir sie einfach verwenden und ihr einen Wert zuweisen. In der Klasse ist es noch eine Fehlermeldung, denn jeder Gartenteil soll ja seine eigene Richtung mitbringen. `(string) self.richtung` gibt den Inhalt von `richtung` als Text aus. Was das soll, wird klarer, wenn wir uns ein abgeleitetes Objekt der Klasse ansehen, den nordwestlichen Teil des Gartens:

```
Object  EdenNW
 class  GartenW
 with   richtung "nordwestliche",
        se_to Mitte,
        e_to EdenN,
        s_to EdenW;
```

Hier lautet also die Beschreibung: „... Ein klarer Bach fließt durch diesen Teil des Gartens in nordwestliche Richtung."

2.7 „... und siehe, eine Tür war aufgetan ..."

Verschlossene Türen und versteckte Schlüssel gehören zu einem Textadventure wie die Olive zum Martini. Deshalb gibt es auch in Eden eine verschlossene Tür; sie führt vom östlichen Teil des Gartens – `EdenE` – zu einem Raum, den wir noch nicht gesehen haben und der den schönen Namen `EastOfEden` trägt. Es handelt sich um eine komplizierte Tür: Sie ist auf beiden Seiten (fast) gleich; dafür steht sie nicht an einer Seite des Raums oder – wie gewöhnlich – an einer Wand, sondern mittendrin ohne eine bestimmte Richtung; außerdem bietet sie eine böse Überraschung, wenn der Spieler sie zum ersten Mal öffnet (zum Trost bekommt er Punkte dafür):

```
Object Tuer EdenE
 with  description "Die Tür ist bis auf das Schlüsselloch mit Blumen
                    zugewachsen. Sie steht mitten im Raum, trotzdem
                    kannst du ihre Rückseite nicht sehen.",
       name 'tuer' 'blume' 'blumen' 'zugewachsen' 'schluessel' 'loch'
            'Raum' ,
       each_turn[;
          if (location == EastOfEden)
              self.&name-->6 = 'Garten';
          else
              self.&name-->6 = 'Raum';
          ],
       when_closed "Eine mit Blumen bewachsene Tür steht da.",
       when_open [;
          if (self in EdenE) "Eine Tür führt aus dem Garten heraus in
                              einen weiten, offenen Raum."; ! rtrue!
          "Die Tür führt zurück in den Garten Eden.";
          ],
       door_to [;
          if (self in EdenE) return EastOfEden; ! auch das ist true!
          return EdenE;
          ],
```

```
        door_dir in_to,
        with_key Primel,
        before [;
           lookUnder: "Du siehst viele Blumen.";
           take, lift, transfer: "So kommst du nicht weiter.";
           ],
         react_before [;
            insert, puton: if (self == second)
                           {
                              if (noun==primel)
                                 <<unlock self primel>>;
                              else
                                 "So kommst du nicht weiter.";
                           }
            ],
         after [;
            open: if (self hasnt general)
                  {
                     achieved(0);
                     give self general;
                     move schlange to mitte;
                     "Du öffnest die Tür nur einen Spalt weit, da wird sie
                      dir schon aus der Hand gestoßen. Etwas Großes, rot und
                      schwarz Gefärbtes drückt die Tür auf und dringt in den
                      Garten ein. Es ist ein Tier, das du noch nicht kennst,
                      langgestreckt und ohne Gliedmaßen. Es sieht dich kurz
                      an, zischt, und du fühlst dich mit einem Mal wie eiskalt.
                      -- Dann wendet es sich wieder ab und schlängelt durch
                      das Gras in westlicher Richtung in den Garten.^^
                      Das Tier braucht einen Namen. Du beschließt, es
                      ~Schlange~ zu nennen. ^^
                      Die Tür ist jetzt offen. Du siehst durch die Tür in einen
                      weiten, offenen Raum.";
                   }
            unlock: remove primel;
                    give primel general;
                    "Du hältst die Primel an das Schlüsselloch. Sie schlägt
                     sofort Wurzeln zwischen den anderen Blumen und ist von
                     ihnen nicht mehr zu unterscheiden.
                     ^^Die Tür ist nicht mehr versperrt.";
            ],
         found_in EdenE EastOfEden,
         short_name "Tür",
         dekl 9,
has      female static door openable lockable locked;
```

2.7.1 Die Tür an sich

Zur Tür wird ein Objekt durch das Attribut `door`. `openable` legt fest, dass der Spieler sie auf- und zumachen kann; `lockable` legt fest, dass er sie – mit dem richtigen Schlüssel – auf- oder zusperren kann. Das Attribut `locked` schließlich bedeutet, dass die Tür zu Anfang zugesperrt ist. Die Aktionen dazu heißen `##Open`, `##Close`, `##Lock` und `##Unlock`.

`door_dir` legt die Richtung fest, in die der Spieler gehen muss, um durch die Tür zu gehen (hier nur: `in_to`, d.h. hinein; eine normalere Tür stünde an einer Wand in einer bestimmten Richtung und hätte an dieser Stelle noch etwa `e_to` oder `w_to`). Diese *property* darf man nicht verwechseln mit

`door_to` enthält den Namen des *Raums*, in den der Spieler gelangt, wenn er durch die Tür geht.

`with_key` enthält den Namen des Objekts, mit dem die Tür auf- und zugesperrt werden kann (hier nur einmal, weil die Primel nach dem Aufsperren mit dem Befehl `remove` aus dem Spiel entfernt wird).

`when_closed` und

`when_open` enthalten unterschiedliche Beschreibungen je nach dem Zustand der Tür.

Das Auf- und Zumachen und das Auf-und Zusperren eines `container`-Objekts funktioniert genauso wie bei der Tür. `door_dir` und `door_to` sind dabei natürlich unnötig.

Beachte: Eine Tür muss gar keine Tür sein. Mit dem eben beschriebenen Instrumentarium kann man auch z.B. eine Brücke oder einen Materietransmitter bauen.

2.7.2 Zwei Seiten einer Tür

Im klassischen Adventure dreht sich alles darum, durch die Tür und an den Schatz zu kommen. Wie die Tür auf der Rückseite aussieht, ist nicht so wichtig. Unsere Tür aber hat zwei Seiten:

Zunächst ist sie wieder mit `found_in` als Schiebekulisse implementiert, d.h. sie befindet sich immer in dem Raum, in dem auch die Spielfigur ist. Dies ermöglicht es, entweder mit `(location==EdenE)` oder mit `(self in EdenE)` abzufragen, auf welcher Seite wir gerade sind. (Beides führt hier zum gleichen Ergebnis: `location` ist der augenblickliche Aufenthaltsort der Spielerfigur[13]; mit `in` wird abgefragt, ob `EdenE` der `parent` von `self`, d.h. der Tür ist.) Je nach dem Ergebnis ändern wir den Inhalt von

[13] Jedenfalls dann, wenn es hell ist. Wenn es dunkel ist, ist die Spielerfigur in einem besonderen Ort namens `TheDark` und nur die Variable `real_location` gibt Auskunft darüber, wo sie sein sollte.

`door_to` und die Beschreibung des Raums, der durch die offene Tür zu sehen ist, in der *property* `when_open`. Bei einer „normalen" Tür müssten wir auch noch `door_dir` ändern.

Beachte: Hier ist an zwei Stellen (die ich markiert habe) der oben (2.6.3 auf Seite 29) beschriebene Trick mit (implizitem) `return` angewendet worden; bei „sauberer" Programmierung müsste da jeweils noch ein `else` stehen.

Um das Ganze perfekt zu machen, jonglieren wir auch mit dem Namen der Tür. Der Befehl „Gehe in den Garten" soll den Spieler durch die Tür schicken, wenn er auf der anderen Seite steht. Steht er aber schon im Garten, wäre der Befehl sinnlos und der Spieler würde sich zu Recht aufregen, wenn er mit „Gehe in den Garten" aus dem Garten herausgeschickt würde.

Zu Spielbeginn steht die Tür auf der Gartenseite (in `EdenE`) und der letzte (siebte) Eintrag der `name`-*property* lautet: `'Raum'`. Auf `'Garten'` hört die Tür gar nicht, wohl aber auf „Gehe in den Raum". Geht der Spieler aber durch die Tür, so ändern wir diesen (siebten) Eintrag von `'Raum'` in `'Garten'` und voilá: „Gehe in den Garten" funktioniert; dafür hört die Tür nicht mehr auf `'Raum'`, denn das ist überschrieben worden.

Wie macht man das? Ganz einfach: Intern ist das `name`-*property* ein *array*, d.h. ein Feld aus einzeln (mit dem Operator `-->`) adressierbaren Wörterbucheinträgen. Ein „normales" *property* der Tür kann z.B. mit `tuer.description` (innerhalb des für die Tür geschriebenen Codes einfacher mit `self.description`) aufgerufen werden. Uns interessiert aber nicht die gesamte `name`-*property*, sondern nur ein einzelner (der siebte und letzte) Eintrag. Um dies zu kennzeichnen, muss dem `name` ein `&` vorangestellt werden, also `self.&name`.

Nun kann der siebte Eintrag mit `self.&name-->6` adressiert werden (der erste Eintrag hat den Index 0, der siebte den Index 6).

Wie aber merken wir, ob der Spieler durch die Tür gegangen ist? Wir schauen ganz einfach nach jedem Zug nach, wo er steht und sorgen dafür, dass der Name richtig gesetzt ist. Dafür benutzen wir der Einfachkeit halber die *property* `each_turn` , die eine Routine enthalten kann, die bei jedem Zug ausgeführt wird, wenn die Tür im Blickfeld des Spielers ist.

2.7.3 Die Türe der Pandora

Wenn die Tür zum ersten Mal geöffnet wird, ist das Attribut `general` noch ungesetzt. Dann lassen wir die Schlange in den Garten einbrechen und setzen anschließend `general,` damit das nicht noch einmal passiert. Mit der Routine `Achieved(0)` wird festgehalten, dass der Spieler die erste (bzw. die nullte, auch hier wird beginnend mit Null gezählt) Aufgabe des Spiels gelöst hat und dafür Punkte beanspruchen kann.

2.7.4 noun und second

Wir betrachten nochmal die react_before-*property* und dort die Regel für ##Insert und ##PutOn : Hier werden Handlungen des Spielers wie „Setze die Primel auf die Tür" *abgefangen* (In diesem Befehl wäre die Primel erstes, direktes Objekt (noun) und die Tür zweites Objekt (second).) und *abgeändert* — in den Befehl „Sperre die Tür mit der Primel auf" (auf „Informese" <Unlock Tuer Primel>, der zweite Satz spitze Klammern steht wieder für ein implizites return und statt Tuer kann self stehen).

2.8 „Lasst uns Menschen machen . . . "

2.8.1 Die Spielerfigur

Auch die Spielerfigur selbst ist ein Objekt. Man muss nicht unbedingt ein Objekt für sie definieren, sondern bekommt eine vorgefertigte Figur gestellt. Wer aber will, dass sich die Spielerfigur anders verhält oder dass sie anders aussieht als der Standard, der muss sich selber eine schnitzen:

```
Object  Adam
 with   name 'adam' 'Mensch' 'spieler' ,
        short_name "Adam",
        dekl 0,
        rippe true,
        number,
        description [;
           if (self.rippe)
              {
                  print "Du bist ein Mensch. " ;
                  style bold; print "Der "; style roman;
                  "Mensch. Du siehst so ähnlich aus wie ER, sagt man.";
               }
           else
               if (self hasnt Erkenntnis)
                  "Eine Rippe fehlt. Dafür hast du eine frische
                   Narbe.";
               else
                   if (Feigenblatt hasnt worn)
                      "Du bist nackt!";
                   else
                      "Das Feigenblatt verdeckt deine Blöße nur
                       dürftig.";
           ],
```

```
        add_to_scope koerper,
 has    light proper male animate;
```

Um diese Spielerfigur zu verwenden, muss dann noch – in die Routine `initialise()` – eingefügt werden:

```
ChangePlayer(Adam);
```

Zweck des Ganzen ist es, Adam je nach dem Spielstadium anders aussehen zu lassen: Er hat nach der Erschaffung von Eva ja eine Rippe weniger als vorher; er darf erst dann bemerken, dass er nackt ist, wenn er von der Frucht des Baumes der Erkenntnis gegessen hat und er ist nicht mehr nackt, wenn er sein Feigenblatt trägt.

Für solche Entscheidungen sind boolesche Werte (also `true` und `false`) ganz praktisch: Die *property* `rippe` wird einfach neu eingeführt und mit `true` vorbelegt, solange Adam die Rippe noch hat. Bei Durchführung der bekannten Operation muss man dann `adam.rippe=false;` setzen.

Bei der Erkenntnis – nach der wir im Spiel ziemlich oft fragen und die nicht nur Adam, sondern auch Eva haben kann – gehen wir anders vor und verwenden ein Attribut. Das muss allerdings vorher definiert sein:

```
Attribute Erkenntnis;
```

und kann mit dem Befehl `give` gesetzt (`give eva Erkenntnis;`) und mit den Operatoren `has` und `hasnt` abgefragt werden.

Beachte: Das Schlüsselwort `has` kommt in zwei verschiedenen Bedeutungen vor: zum Einen leitet es in der Objektdefinition die Liste der Attribute ein, die das Objekt hat; zum Anderen kann es in logischen Termen dazu benutzt werden, das Vorhandensein eines Attributs abzufragen: `if ((adam has Erkenntnis) && (eva hasnt Erkenntnis)) ...`

Die *property* number wird von der Library benötigt, wenn das Objekt als Spielfigur eingesetzt werden soll. Ebenso muss die Library wissen, dass Adam ein Lebewesen ist, dazu dient das Attribut `animate`. Das Attribut `proper` bewirkt, dass „Adam” als Eigenname behandelt wird, so dass die Ausgaberoutinen nicht automatisch einen Artikel davorsetzen.

2.8.2 „Das ist doch Bein von meinem Beine” – Körperteile

Das Nachmodellieren einzelner Körperteile ist normalerweise nicht notwendig, es sei denn, die Körperteile hätten eine besondere Funktion im Spiel. An Adam ist in dieser Beziehung nur die Narbe interessant:

```
Object  koerper
 with   short_name "Körper",
        adj "dein",
        dekl 2,
        name 'rippe' 'narbe' 'koerper' 'bauch' ,
        before [;
           examine: return false;
           default: "Dein Körper gehört zu dir.";
           ],
        description [;
           if (adam.rippe)
              "Du siehst nichts Ungewöhnliches.";
           else
              "Eine frische Narbe ist dort, wo vorher eine Rippe
               gewesen ist. Sie scheint gut zu heilen. --
               Sonst siehst du nichts Ungewöhnliches.";
           ],
 has    proper scenery male;
```

Dieser Körperteil ist kein Besitztum von Adam, d.h. das Objekt steht einfach unabhängig im Raum; dass der Spieler es dennoch sehen kann, wird durch die Zeile `add_to_scope koerper` in der Definition von Adam bewirkt. Da es kein *Child* des Objekts Adam ist, wird es beim Schwimmengehen auch nicht abgelegt.

Die `default:`-Regel in der `before`-*property* bewirkt, dass alle Aktionen, die der Spieler mit dem Körper versuchen mag (von Anzünden bis Wegwerfen) nur lakonisch mit „Dein Körper gehört zu dir" abgewiesen werden (zur Erinnerung: Der String steht in Kurzschreibweise für ein `print_ret` und gibt `true` zurück: Ende der Aktion.)

Alle Aktionen? Nein, eine einzelne kleine Aktion, nämlich `##Examine`, muss offen bleiben, sonst bekommt der Spieler ja die Beschreibung der Narbe nicht zu Gesicht. Also fügen wir eine Regel für `##Examine` ein, die *im wahrsten Sinn des Wortes nichts* macht: Sie gibt `false` zurück, die Aktion wird unbeeinflusst weiter ausgeführt und die Beschreibung wird ausgegeben. Stünde statt des `return false` ein `return true`, würde die Regel *auf andere Weise nichts* machen: Ende der Aktion und nichts würde ausgegeben, auch die Beschreibung nicht.

2.8.3 „... und flochten sich Feigenblätter zusammen..."

Für Kleidungsstücke gilt dasselbe wie für Körperteile. Beschränkung auf das Notwendigste sei die Maxime. Das fällt im Garten Eden auch gar nicht schwer:

```
Object  Feigenblatt
 with   short_name "Feigenblatt",
```

```
        dekl 4,
        initial "Dir fällt auf, dass ein Feigenblatt gut geeignet
                 wäre, um deine Blöße zu bedecken.",
        description "Ein Feigenblatt. Grün. Vielleicht ein bisschen
                     zu klein.",
        after [;
           take: give self worn;
                 achieved(5);
                 "Du nimmst das Feigenblatt und machst dir einen
                  Schurz daraus.";
          ],
        before [;
           disrobe, drop, give: "Aber das Feigenblatt ist so kleidsam!";
           ],
        name 'blatt' 'feigenblatt',
has     neuter clothing;
```

Das Attribut `clothing` macht aus einem Objekt ein Kleidungsstück und ermöglicht die Verwendung der Aktionen `##Wear` und `##Disrobe` zum An- und Ausziehen. Dabei setzt die Library automatisch das Attribut `worn`, solange das Kleidungsstück getragen wird.

Das Feigenblatt hat die Besonderheit, dass es sich „von selber" anzieht, wenn der Spieler es nimmt. Deshalb muss hier `worn` explizit gesetzt werden. (Außerdem gibt es Punkte für die sechste gelöste Aufgabe.) Einmal getragen, lässt es sich – nach dem Vorbild eines anderen klassischen Kleidungsstücks[14] – wegen der Regel für `##Disrobe` in der `before`-*property* allerdings nicht mehr ausziehen.

2.9 „Da schied Gott das Licht von der Finsternis . . . "

Das Spielfeld muss angemessen beleuchtet sein, sonst tappt der Spieler im Dunkeln. Die einfachste Methode ist, dem Spieler einfach ein Licht mitzugeben, indem – wie im vorangegangenen Beispiel – das Attribut `light` für die Spielerfigur gesetzt wird.

Wenn der Gegensatz zwischen Licht und Dunkel irgendeine Rolle spielen soll, wird man einigen Räumen (Raumobjekten) das Attribut `light` verleihen und anderen Räumen nicht. Dann braucht der Spieler ein Objekt, das er bei Bedarf mitnehmen kann und das Licht abgibt. Meistens ist das eine Lampe, die der Spieler ein- und ausschalten kann. Das Attribut `switchable` bezeichnet ein mit den Aktionen `##SwitchOn` und `##SwitchOff` ein- und ausschaltbares Objekt. Die properties `when_on` und `when_off` steuern, wie der Schaltzustand sich auf das Aussehen des Objekts auswirkt.

Folgende Lampe ist eine verkleinerte Ausführung der Lampe aus „Adventure":

[14] *„But the brass loincloth is so becoming!"* heißt es in Infocoms „Leather Goddesses of Phobos". (Für Spielerinnen ist es ein *brass bikini*.)

```
Object  brass_lantern
 with   name 'lampe' 'taschenlampe' 'scheinwerfer' 'laterne'
             'licht' 'leuchte' 'messing' 'blank' 'poliert',
        post "aus Messing",
        short_name "Taschenlampe",
        dekl 9,
        when_off "Eine blankpolierte Taschenlampe steht bereit.",
        when_on [; print "Deine Lampe ist hier. Sie leuchtet ";
                   if (self.power_remaining < 30)
                       "nur noch schwach.";                 ! rtrue!
                   "hell.",
           ],
        before [;
           Examine: print "Es ist eine Taschenlampe aus Messing";
                    if (self hasnt light)
                       ". Sie ist ausgeschaltet.";          ! rtrue!
                    if (self.power_remaining < 30)
                       ", die nur noch schwach leuchtet.";  ! rtrue!
                    ", die hell leuchtet.";
           Burn: <<SwitchOn self>>;
           Rub: "Eine elektrische Lampe zu reiben ist nicht allzu
                 erfolgversprechend. Wie dem auch sei, es geschieht
                 dabei nichts aufreibendes ...";
           ],
        after [;
           SwitchOn: give self light;
           SwitchOff: give self ~light;
           ],
        power_remaining 42,
 has    female switchable;
```

Das Original (zu finden bei bei Toni Arnold: `http://www.copyriot.com/tarnold/abent/abent.inform`) hat Batterien, die bei jedem Spielzug Ladung verlieren und ausgetauscht werden müssen.

2.10 „... ein Buch, geschrieben inwendig und auswendig ...”

Auswendig ist nicht allzu schwer:

```
Object  Heft
 with   short_name "Heft",
```

```
        adj "klein",
        dekl 1,
        description "Es ist ein kleines Heft mit nicht allzuvielen Seiten. Auf
                     dem Titel steht: ~Der Abentteuerliche Informissimus
                     Teutsch -- Eine Einführung in die Programmierung mit
                     Inform und der deutschen Library.~ Irgendwie macht es
                     einen unfertigen Eindruck.",
        name 'heft' 'informissimus' 'klein' 'handbuch' 'info',
        before [;
           open, search: "In dem kleinen Heft findest du Informationen über das
                  Spiel."
           close: "Du schließt das kleine Heft.";
        ],
has     neuter;
```

Auch der Befehl „Lies das Heft" führt dazu, dass die `description` ausgegeben wird - Lesen eines normalen Objekts ist gleichbedeutend mit dem Betrachten – `##Examine`. Den Reiz eines Buches macht aber aus, dass verschiedene Dinge darin stehen; dass man also mit Befehlen wie *„Lese die Hinweise im Heft" - Konsultiere das Heft über den Garten"– „Schlage im Heft über das Thema Eden nach"* – an die im Buch selbst enthaltenen Informationen kommt und nicht nur die Beschreibung des Buchdeckels erhält.

Die Library löst für solche Befehle die Aktion `##Consult` aus und übergibt als `noun` das Objekt, in dem gelesen werden soll (hier: `Heft`). Dann aber wird es schwierig.

2.10.1 Parsen selbst gemacht

Den Teil der Eingabezeile, in dem steht, was gelesen werden soll, muss das Programm selbst analysieren. Zu diesem Zweck übergibt die Library eine Variable `consult_words`, die die Anzahl der zu analysierenden Worte enthält und einen Zeiger `consult_from` auf die Eingabezeile, der auf das erste Wort des Textes deutet, der gelesen werden soll, im letzten oben angeführten Beispiel wäre dieser Text *„das Thema Eden"*; `consult_words` wäre gleich 3; der Zeiger `consult_from` weist auf das Wort „das".

Außerdem haben wir noch die Funktion `NextWord()` zur Verfügung, die das nächste Wort aus der Eingabezeile liest und mit dem internen Wörterbuch des Spiels vergleicht. Welches Wort das „nächste" ist, erkennt die Funktion aus der globalen Variablen `wn`. Diese müssen wir – aus `consult_from` – initialisieren, um an der richtigen Stelle anzufangen. Praktischerweise zählt `NextWord()` die Variable `wn` bei jedem Aufruf um 1 hoch.

Die `before`-Regel für die Aktion `##Consult` könnte – für Themen, die bis zu drei Worte lang sind und das Wort *Thema* enthalten können – etwa so aussehen:

```
before [ w1 w2 w3 thema;                    ! vier lokale Variable
```

```
Consult: wn = consult_from;
         w1 = NextWord();              ! erstes Wort des Themas
         w2 = NextWord();              ! evtl. zweites Wort des Themas
         w3 = NextWord();              ! evtl. drittes ...
        if (consult_words==1 && w1~='thema' or 'themen')
           thema = w1;
        else
           if (consult_words==2 && w1=='dem' or 'den' or 'die' or 'der' or 'das' )
           else
              if (consult_words==2 && w1=='thema')
                 thema = w2;
              else
                 if (consult_words==3 && w2=='thema')
                    thema = w3;
                 else "Versuche es mit: ~Lies über <das Thema> im Heft nach~";
```

Beachte: Die in 'einfachem Hochkomma' stehenden Worte sind keine schlichten Zeichenketten, sondern es sind im *dictionary* des Spiels eingetragene Konstanten, intern realisiert als Zeiger auf eine Tabelle mit den tatsächlichen Worten. (Jedes Wort, das im Programmtext in 'einfachen Hochkommas' geschrieben wird, wird in dieses Wörterbuch eingetragen.) Weil es aber konstante Werte sind, können sie in einer `switch`-Anweisung (zur Erinnerung: Kapitel 3.1.3 auf Seite 69)verwendet werden:

```
      switch (thema) {
         'adam', 'mich', 'eva', 'mensch', 'frau':
             "Adam und Eva sind die ersten Menschen. Deutsche Lehrbücher
              fangen immer bei Adam und Eva an.";
         'heft', 'inform' , 'informissimus':
             "Zusammen mit diesem Spiel sollten Sie den ~Abentteuerlichen
              Informissimus Teutsch~ erhalten haben - eine Anleitung und eine
              Kurzreferenz zum Programmieren deutschsprachiger interaktiver
              Belletristik.";
         'vorwort', 'hinweise' 'hilfe':
             "Dies ist interaktive Belletristik.";
         'garten', 'baum', 'Baeume':
             "Im Garten Eden stehen viele Bäume, sie sind lustig anzusehen
              und ihre Früchte sind gut zu essen. Nur die Bäume in der Mitte
              des Gartens sind den Menschen verboten worden.";
         'eden', 'spiel':
             "Dieses Spiel ist ~nur~ das Bei-Spiel zum ~Abentteuerlichen
              Informissimus Teutsch~.";
         default: "Darüber steht nichts in dem kleinen Heft.";
      }
open: ...
```

```
 close: ...
],
```

2.10.2 Parsen mit GInfo.h

Es geht aber auch einfacher. Man kann den Parser der Library die Arbeit machen lassen; dazu muss man ihm aber Objekte vorgeben, die die möglichen „Anfragen" an den Inhalt des Heftes abdecken – und Regeln, wie er diese Objekte verwenden kann. Eine Klasse – `Topic` – für solche Objekte und die notwendigen Regeln enthält die zusätzliche Bibliothek GInfo. Sie definiert ausserdem ein „Aufhängerobjekt" namens `Topics`, mit dem gesteuert werden kann, ob bestimmte Themen gerade „aktuell" sind (dann sind sie *children* von `Topics`) oder nicht (dann kann man sie mit z.B. `remove TMann;` entfernen (und mit `move TMann Topics;` wieder ins Spiel bringen)).

Die einzelnen Objekte enthalten nur ihren eigenen Namen, grammatische Zusatzinformation – und in der `name`-*property* die Worte, mit denen sie vom Spieler angesprochen werden können (`'Thema'` erben sie von `Topic`). Wir lernen hier außerdem eine neue Kurzschreibweise kennen: `Topic TMann` bedeutet dasselbe wie `Object TMann ... class Topic`.

```
Include GInfo;

Topic TMann Topics
 with short_name "Adam",
      name 'Adam' 'Mann' 'Mensch' 'mich' 'mir' 'dich' 'dir',
      dekl 0,
 has  male proper;

Topic TFrau Topics
 with short_name "Eva",
      name 'Eva' 'Frau' 'Mensch' 'Gefaehrtin' 'Gehilfin',
      dekl 0,
 has  female proper;

Topic TSpiel Topics
 with short_name "Programm",
      name 'Spiel' 'Programm' 'Eden' 'Information' 'Informationen' 'info',
      dekl 1;
! neuter erbt das Objekt von der Klasse Topic

Topic THilfe Topics
 with short_name "Hilfe",
      name 'Hilfe' 'Anleitung' 'Einführung',
      dekl 9,
 has  female;
```

Gibt nun der Spieler den Befehl: *„Lies im Heft über Adam nach"*, dann übergibt der Parser wie vorhin die Aktion `##Consult` und als `noun` das Objekt `Heft`; aber jetzt als zweites Objekt des Satzes das Objekt `TMann` in der Variablen `second` und die Programmierung ist ganz einfach:

```
before [;
   Consult:
      switch (second) {
              TMann, Tfrau:
                      "Adam und Eva sind die ersten Menschen.
                       Deutsche Lehrbücher fangen immer bei
                       Adam und Eva an.";
              THeft:  "Zusammen mit diesem Spiel werden Sie in
                       naher Zukunft den ~Abentteuerlichen
                       Informissimus Teutsch~ erhalten ..."
              THilfe: "Dies ist interaktive Belletristik. Mehr
                       erfahren Sie mit ~Hilfe~.";
              TGarten, TSpiel:
                      "Dieses Spiel ist ~nur~ das Bei-Spiel
                       zum ~Abentteuerlichen Informissimus
                       Teutsch~.
              ...
      }
   ...
],
```

Damit es korrekt funktioniert, muss das Heft noch das Attribut `legible` bekommen und nach `Include grammarg;` muss noch der zweite Teil des GInfo-Pakets eingebunden werden mit dem Befehl:

```
Include GinfoG;
```

Mehr über GInfo.h steht in Kapitel 5.1 auf Seite 129.

2.11 Die Bevölkerung des Gartens, `life` und `orders`

Auch in dem schönsten Garten wird es dem Menschen alleine bald fad. Mit einem kleinen Trick wird etwas Belebung vorgetäuscht: bei durchschnittlich jedem dritten Zug wird nach dem Zufallsprinzip eine von acht Meldungen angezeigt, die verschiedene Tiere und ihre Bewegungen im Garten beschreiben. Das ist aber nur Illusion. Wir wollen uns mit echten Lebewesen – d.h. eigenständigen Objekten – beschäftigen.

2.11.1 „Die Erde bringe hervor lebendige Tiere . . . ”

Ein Lebewesen ist zunächst ein Objekt, für das das Attribut `animate` gesetzt ist. Die Library berücksichtigt dann automatisch, dass mit dem Lebewesen verschiedene Aktionen möglich sind, die es für ein unbelebtes Objekt nicht zulässt. Das sind alle Kommunikationsvorgänge (fragen, antworten, etwas zeigen, erzählen, befehlen) sowie ein paar andere Aktionen, die man üblicherweise mit unbelebten Gegenständen nicht macht (angreifen, aufwecken küssen, einen Gegenstand übergeben oder nach ihm werfen). Die mit diesen Handlungen verbundenen Aktionen (für eine vollständige Liste siehe 3.2.7 auf Seite 98) könnten nun normalerweise in der `before`-*property* des Objektes behandelt werden.

Damit das Ganze übersichtlicher wird, gibt es aber eigens für diejenigen Aktionen, die der Spieler nur mit einem Lebewesen vornehmen kann, eine besondere *property*. Sie heißt `life`, funktioniert ansonsten aber genauso wie `before`.

Sehen wir uns also ein einfaches, aber dennoch wirkungsvolles Tier an:

```
Object  schlange
 with   short_name "Schlange",
        name 'schlange' 'schuppen' 'panzer' 'rot' 'schwarz' 'gross',
        adj "rot und schwarz geschuppt",
        dekl 9,
         life [;
          show, ask, askfor, tell: "Sie sieht dich kurz an
                        und du fühlst dich wieder eiskalt.";
          default: "Besser nicht. Das Biest ist unheimlich.";
         ],
        before [;
         take, push, pull, touch, rub, lift, squeeze, taste:
            <<attack self>>;
         smell: "Ein Geruch wie Pech und Schwefel.";
        ],
        description "Das ist die Mutter aller Schlangen. Sie misst
            wenigstens sieben Schritte
            und hat einen Panzer aus roten und schwarzen Schuppen.",
        initial "Eine große Schlange hat sich unter dem Baum der
            Erkenntnis zusammengeringelt.",
  has   animate static female;
```

Beachte: `##attack` wird von `life` behandelt und aufgrund des `default`-Eintrags mit *„Besser nicht. . . ”* beantwortet. Mit dem in der `before`-*property* eingetragenen `<<attack self>>` wird das gleiche Verhalten für alle davor aufgeführten „einfachen” Aktionen bewirkt.

2.11.2 „Darum wird ein Mann seinen Vater und Mutter verlassen. . . ”

Menschen und andere „intelligente” Lebensformen unterscheiden sich für die Inform-Autorin in der Art der Konstruktion nicht vom Tier. Die eigentliche Herausforderung ist es, Objekte für andere Personen außer der Spielerfigur[15] zu erschaffen, die sich im Kontext des Spiels einigermaßen selbständig und glaubhaft verhalten. In der Regel steht dabei das Kommunikationsverhalten dieser Objekte im Vordergrund, also ihre Reaktionen auf die Aktionen `##ask`, `##tell` etc.

In den frühen Tagen der Adventure-Programmierung waren richtige Abenteurer noch richtige Abenteurer und richtige NPC's waren wortkarge Diebe, Trolle oder Piraten, die sich ebenfalls recht eindimensional verhielten. Daher kommt es, dass Inform im Verhältnis zu anderen Features herzlich wenig Hilfestellung zur Programmierung von Gesprächen bietet. Das hat eine Vielzahl von Zusatzbibliotheken auf den Plan getrieben, die aber hier – mit Ausnahme des schon bekannten `GInfo.h` – nicht verwendet werden sollen und unbehandelt bleiben.[16]

Wir beschränken uns auf „einfachste” ask-tell-Kommunikation.[17] Die verwendbare Kommunikationsbandbreite wird dabei gewissermaßen auf die eines Zwei-Wort-Parsers begrenzt, d.h. wir können dem Gesprächspartner der Spielerfigur neben dem Verb nur *ein* Objekt (das Gesprächsthema) mitteilen, das vom Parser her mögliche zweite Objekt ist der Gesprächspartner selbst. Damit wir das Gesprächsthema nicht selbst parsen müssen, verwenden wir für die Gesprächsthemen die gleichen Themenobjekte wie für den Umgang mit Lesestoff, vergleiche oben 2.10.2 auf Seite 43. Sie sind am vorangestellten `T` zu erkennen.

Auf diese Weise ist aber auch schon eine ganze Menge möglich. Machen wir halt' mal einen völlig unbescheidenen Anfang:

```
Object Creator
with   short_name "Gott",
       dekl 4,
       name 'Trurl' 'Gott' 'Allah' 'Herr' 'Schoepfer' 'Jehova'
            ! etcetera ad libitum
       initial "Du fühlst die Gegenwart des ERSTEN.",
       description "Der HErr ist in seinem heiligen Tempel, des HErrn Stuhl
          ist im Himmel. Du kannst IHN nicht sehen und nicht
          beschreiben, aber du weißt, ER ist hier.",
       before [;
         examine, attack,throwAt, wakeOther, answer, ask, tell, order,
```

[15] Im Fachjargon Non-Player-Characters (NPC) genannt.

[16] Die wichtigste Neuerung ist die menügesteuerte Kommunikation: Auf den Befehl „*sprich mit . . .* ” erhält derSpieler eine Auswahl verschiedener vorgegebener Sätze und kann sich für einen entscheiden. Die präsentierte Auswahl verändert sich je nach dem Fortgang des Gesprächs. Das kanonische Beispiel dieser Technik ist „*Photopia*” von Adam Cadre.

[17] Ask-Tell-Kommunikation ist keinesfalls mit „einfach” gleichzusetzen. Eine der komplexesten NPC überhaupt, Galatea im gleichnamigen Spiel von Emily Short, verwendet auch „nur” ask und tell.

```
            askfor, show: return false;
        default: "Du wagst noch nicht einmal den Gedanken
                  daran.";
        ],
```

Das `return false` ist notwendig, um das nachfolgende `default:` nicht *alle* Aktionen „auffressen" zu lassen. Da `before` vor `life` aufgerufen wird, müssen auch die speziellen, sonst für `life` reservierten Aktionen freigehalten werden.

```
orders [;
   give: <<ask self noun>>;
   default: "ER lässt sich nicht befehlen.";
],
```

Die Property `orders` ist ein zweiter Spezialfall für Lebewesen. Hier landen Befehle, die der Spieler dem NPC direkt erteilt. Der einzige hier sinnvolle Befehl lautet *„Herr, gib' mir Eva."* Er wird durch die Konstruktion mit umgeleitet auf den für *„Frage den Herrn über Eva"* bestimmten Programmteil. Aber auch alle anderen *„Gib mir"*-Befehle werden so auf die entsprechenden *„Frage-über"*-Antworten umgeleitet. Ebenso werden weiter unten die *„frage nach"*-Befehle umgeleitet.[18]

```
life [;
   Attack, ThrowAt, Kiss, Love: "Du wagst noch nicht einmal den
         Gedanken daran.";
   WakeOther: "ER schläft nicht.";
   order: "ER lässt sich nicht befehlen.";
   Answer: "Du hast das Gefühl, dass du mit jemand sprichst, der
            dich versteht.";
   Ask: switch (second)
        {
          adam, TMann: "~Ich habe dich heute morgen nach meinem Bilde
             gemacht, aber jetzt bin mir
             gar nicht sicher, ...~ -- ER kommt ins Grübeln.";
          Creator, TEli: "~Ich bin der, der ich bin.
             Namen tun nichts zur Sache.~";
          TEinsamkeit, TFrau:
             ErschaffeEva();
             return true;
```

[18] Eine mögliche Falle für die Autorin ist der Unterschied zwischen Übersetzung und Bedeutung der Aktion `##AskFor`. Die Bedeutung ist *„verlange das Objekt"* im Sinn von *„gib' das Objekt her"*. Die Übersetzung enthält aber insbesondere auch *„frage nach dem Objekt"*, was im Deutschen gern gleichbedeutend mit *„gib' mir Informationen über das Objekt"* verwendet wird. Das ist aber eine ganz andere Aktion, nämlich `##Ask`. Nach einigem Gefiddel mit diesem Problem habe ich meine Version von GInfo.h so umgebaut, dass *„frage nach"* generell als `##Ask` verstanden wird.

```
        TApfel : "~Du sollst von dem Baum in der Mitte
           des Gartens nicht essen.~";
        TObst, TGarten: "~Ich habe den Garten zu deinem
           Aufenthalt gemacht und die Früchte zu deiner Nahrung.~";
        TSchlange :
           "~Es ist besser, wenn du darüber nichts weißt.
            Glaube der Schlange kein Wort!~";
        tuer, TPforte: "~Hmm. - Ich hatte die Pforte des Gartens mit Bedacht
           zugesperrt gelassen.~";
        schwert, TSchwert:
           if (sofa has general) ! Spieler hat das Schwert entdeckt
               "~Waffen sind die Werkzeuge des Bösen. Lass' die
                Finger davon.~";
           else
               "~Schwert? Welches Schwert?~";
        TWasser:
           "~Die Quelle speist die vier Ströme, die vier Ströme
            speisen alle Wasser der Welt.~";
        TAntwort:
           "ER beginnt vom Leben und der Schöpfung zu sprechen
            und dir wird alles klar.";
        TTier:
           "~Du bist es, der eingesetzt wurde, um über alle Tiere zu
            herrschen.~";
        TEngel:
           "~Sie schweben in den höheren Sphären und loben mich
            mit ihrer Musik.~";
        default: "~Fragen, immer nur Fragen ...~";
      };
Tell: switch (second)
      {
        TSchlange, TPforte:
          if (schlange in mitte)
               "~Das musste wohl früher oder später passieren. Ich
                sperre die Pforte lieber wieder zu.~";
          else
               "~Du hast die Schlange also vertrieben? Ja, die
                Gerechtigkeit des Frommen macht seinen Weg eben...~";
        TEinsamkeit, TFrau:
           ErschaffeEva();
           return true;
        default: "Du hast das Gefühl, dass dir jemand zuhört, der
           dich versteht.";
      };
askFor: <<ask self noun>>;
```

```
],

has male animate;
```

2.12 Aktionen abändern

2.12.1 Beten, Singen, Nichtstun

> *„Was ist der Unterschied zwischen einem Geheimrat und einem Wirklichen Geheimrat? —*
> *Der Geheimrat tut nichts. Der Wirkliche Geheimrat tut wirklich nichts."*
> (Autor unbekannt)

Es gibt in Inform zwei Arten von Aktionen; eine davon manipuliert die im Spiel befindlichen Objekte (wie z.B. `##Take`); die andere tut tatsächlich nichts.[19] Gibt der Spieler etwa ein: *„Singe"*, dann wird zwar die Aktion `##Sing` ausgelöst, aber es kommt nur die Standardantwort: *„Dein Gesang ist abscheulich"*. Im Einzelfall kann man dieses Verhalten mit `before`- und `react_before`-Regeln abändern. Es ist aber auch möglich, das Verhalten einer vordefinierten Aktion *insgesamt* abzuändern – mittels eines speziellen Objekts `LibraryMessages`, das zwischen den Befehlen `Include parser;` und `Include verblib;` definiert werden muss.

```
Object LibraryMessages
  with before [;
          wake, sleep: "Sie sagen, es sei der sechste Tag,
                        doch für dich ist es der Erste.
                        Und du bist noch nicht müde.";
          strong: "Das steht so nicht bei Doktor Luther!";
          pray: "~Amen.~";
          sing: Psalm();
       ];
```

`##Strong` ist die Aktion, die bei „starken" Flüchen des Spielers ausgelöst wird.

In der Regel für `##Sing` wird eine – selbständige – Routine aufgerufen, die zufallsgesteuert eine Zeile aus verschiedenen Psalmen „absingt" Die Funktion `random(n)` erzeugt eine Zufallszahl zwischen 1 und n.

[19] Die „amtliche Bezeichnung" ist „Klasse 2" für die die Spielwelt verändernden und „Klasse 3" für die untätigen Aktionen. Klasse 1 sind die „Metaverben", die zwar jede Menge tun, sich dabei aber überhaupt nicht ins Spiel einmischen (z.B. Spielstand speichern, ein Transkript starten o.ä.) Die Auflistung der Aktionen in Kapitel 4 auf Seite 106 enthält Informationen darüber, welche Aktionen zur einen oder anderen Gruppe gehören.

```
[ Psalm; ! Der erste Parameter einer selbständigen Routine ist ihr Name!
  print "Du erhebst deine Stimme und singst: ";
  switch (random(5)) {
      1: "~HErr, unser Herrscher, wie herrlich ist dein Name in
          allen Landen ...~ ";
      2: "~Ich danke dem HErrn von ganzem Herzen und erzähle alle
          deine Wunder ...~ ";
      3: "~Ich freue mich und bin fröhlich in dir und lobe deinen
          Namen ...~ ";
      4: "~Das Gesetz des HErrn ist vollkommen und erquickt die
          Seele ...~ ";
      5: "~Halleluja! Halleluja! Haaa - leluja!~ ";
  }
];
```

Es wäre genauso möglich, hier eine Routine aufzurufen, die „tatsächlich" etwas tut (d.h. eine Veränderung der Modellwelt des Spiels bewirkt).

2.12.2 Schiebung

Hin und wieder stößt man auf Gegenstände, die zu schwer sind, als dass die Spielerfigur sie tragen könnte; die aber mit entsprechendem Kraftaufwand *geschoben* werden können. *„Eden"* enthält nur ein Sofa, das auf dem Fußboden eines Raumes hin- und hergeschoben werden kann. Das ist nichts Besonderes:

```
Object -> Sofa
 with     description "Es ist aus fettem kastanienbraunen Leder und
                       sieht gemütlich aus.",
          before [;
                    take, lift, transfer: "Du kannst das Sofa nur
                       schieben, nicht heben.";
                    pull, push, turn:
                       "Das Sofa gleitet fast ohne Widerstand über den
                        Boden. ";
```

Wollten wir es ermöglichen, das Sofa von einem Raum in den anderen zu schieben, könnten wir die Aktion `##pushdir` (im `before`-*property*) in etwa so definieren:

```
pushdir: if ((location == EastOfEden) && (second == in_obj))
                       "Das Sofa passt nicht durch die Tür."; ! rtrue
```

```
            AllowPushDir();
            rtrue;
```

Der Aufruf der Routine `AllowPushDir()` teilt der Library mit, dass die Spielerfigur – wenn eine gangbare Verbindung existiert – *samt dem Sofa* in den benachbarten Raum verschoben werden kann, wenn der Spieler etwa *„Schiebe das Sofa nach Norden"* eingibt. Wollen wir ansonsten gangbare Wege ausnehmen, muss das vorher abgefangen werden: Im Beispiel kann das Sofa nicht durch die Tür in den Garten geschoben werden, denn wegen des in diesem Fall vorher stattfindenden (impliziten) `rtrue` wird `AllowPushDir()` nicht mehr aufgerufen. Ohne `AllowPushDir()` aber kann der Spieler schuften, was er will: er bekommt das Sofa nicht aus dem Raum.

2.13 Spielen und Erkennen: Neue Verben und Aktionen

Aktionen sind bestimmte, vom Spiel festgelegte Handlungsmuster . Wir wissen (seit 2.12.1 auf Seite 49) bereits, dass es Aktionen gibt, die ausser einer Rückmeldung an den Spieler nichts bewirken (z.B. `##pray`) und solche, die eine Veränderung der Spielwelt bewirken können (z.B. `##take` für das Ergreifen eines Objekts durch die Spielerfigur).

Verben sind Teile der Befehle, die der Spieler eintippt, um Aktionen auszulösen (z.B. *„**Bete**"* oder *„**Nimm** den Apfel")*. Verben und Aktionen sind eng miteinander verbunden.

In der Library sind bereits die wichtigsten Verben und Aktionen vordefiniert (siehe unten 4.2 auf Seite 110). Inform ist jedoch nicht auf die vordefinierten Verben und Aktionen beschränkt.

2.13.1 „... lasst uns herniederfahren und ihre Sprache dortselbst verwirren ..."

Im Himmel finden sich auch Musikinstrumente: Eine Posaune und eine Harfe. Das Verb *„blasen"* ist standardmäßig mit einer Aktion `##blow` verbunden, die zu den Klasse-3-Aktionen gehört, also lediglich einen allgemein gehaltenen Text ausgibt. Das lässt sich für das Objekt `Posaune` gut verwenden:

```
before [; blow: "Tröööööt!";
       ],
```

Die Harfe muss man spielen oder zupfen. Wir brauchen also eine Aktion `##play`, die durch *„Spiele auf der Harfe"* oder *„Zupfe die Harfe"* ausgelöst wird:

```
Verb 'spiel' 'zupf'
    * held                              -> play
    * 'auf' held                        -> play;

[ PlaySub;
    print_ret "Auf ", (dem) noun, " kannst du nichts spielen.";
];
```

Typischerweise deklariert man – nach dem Schlüsselwort `Verb` – nicht nur ein Verb allein, sondern gleich eine Gruppe von Synonymen, hier *„Zupfe"* und *„Spiele"*. Die Schreibweise im Hochkommata funktioniert genauso wie im `name`-*property*, Umlaute und ß müssen also ausgeschrieben werden. Das auslautende -e wird immer weggelassen.

Nach der Synonymliste kommt das eigentlich Interessante: Die Grammatikzeilen, die während des Spiels der Parser verwenden wird, um die Eingabe des Spielers zu analysieren. Schauen wir erstmal nach rechts: da steht nach einem Pfeilsymbol die Aktion, die ausgeführt werden soll, also das Ergebnis der Analyse.

Links steht ein Asterisk als Platzhalter für jedes definierte Synonym, danach folgen Grammatiktoken. Text in Hochkommata steht für sich selbst, `'auf'` also für den eingegebenen Text *„auf"*. `held` steht für ein beliebiges Objekt, das von der Spielerfigur gehalten wird. (Um auf einem Musikinstrument zu spielen, muss man es zuerst in der Hand haben.) Der Parser ordnet also Eingabesätze, die mit *„spiele"* oder *„zupfe"* beginnen und – mit oder ohne zwischengestelltes *„auf"* – als (grammatisches) Objekt des Satzes ein Objekt der Spielwelt haben, das die Spielerfigur gerade in Besitz hat, der Aktion `##play` zu – und übergibt in der Variable `noun` das festgestellte Objekt.[20]

Für eine vollständige Liste der Grammatiktoken siehe 3.2.8 auf Seite 102.

Zu einer neudefinierten Aktion gehört zwingend eine Routine, die das Standardverhalten der Aktion festlegt. Die Verbindung zwischen Routine und Aktion wird durch den Namen hergestellt: Die Routine trägt den Namen der Aktion gefolgt von den drei Zeichen `sub` – hier also `play` und `PlaySub`[21]. Das hier festgelegte Standardverhalten trägt dem Rechnung, dass man auf einem x-beliebigen Gegenstand nichts wird spielen können. Die Spielbarkeit der Harfe wird wie bei der Posaune durch einen Eintrag in der `before`-*property* des Objekts `Harfe` hergestellt, etwa:

```
before [; play: "Plingeli-Plang.";
       ],
```

[20] Der Parser geht sogar noch einen Schritt weiter: Hat die Spielerfigur das Objekt nicht in der Hand, erzeugt er zuerst ein `<take Objekt>`. Ist kein Objekt angegeben, versucht er, im Besitz der Spielerfigur ein geeignetes zu finden. Hat die Spielerfigur nichts in Besitz, fragt er nach.

[21] Das CamelCase in PlaySub verwende ich nur wegen der besseren Lesbarkeit. Groß- und Kleinschreibung ist in Inform bis auf wenige Schlüsselwörter unbeachtlich.

2.13.2 Erkennen – aber wen?

Erkenntnis ist ein zentraler Begriff in *Eden*. Deshalb gibt es auch dafür ein Verb, aber mit zweierlei Bedeutung, je nachdem, ob der Spieler sich selbst oder eine andere Person erkennen will. Das eine ist ein eher philosophisch begründeter Vorgang, das andere – zumindest nach dem Sprachgebrauch Martin Luthers – etwas ganz anderes. Dazu kommt, dass geeignete vordefinierte Verben schon auf andere Aktionen zeigen: z.B. ist das Verb *„liebe"* zusammen mit einigen dafür wesentlich besser geeigneten Verben (*„küsse")* für die Aktion `##kiss` definiert (siehe 4.2 auf Seite 119). Beide Probleme lassen sich lösen:

```
Verb 'erkenn'
    * creature                          -> Love
    * 'dich'/'mich' 'selbst'            -> Gnothi;
```

`creature` ist das Grammatiktoken für ein Lebewesen (alle Objekte mit dem Attribut `animate`). Der Schrägstrich zwischen `'dich'` und `'mich'` zeigt an, dass diese beiden Token alternativ stehen können, der Parser erkennt also *„erkenne dich selbst"* und *„erkenne mich selbst"* als identische Eingabesätze.

```
Extend only 'lieb' replace
    * creature                          -> Love;
```

Das Schlüsselwort `extend` ermöglicht die Erweiterung eines Verbs, d.h. die Definition neuer Grammatikzeilen für dieses Verb. Mit dem Zusatz `only` wird klargestellt, dass nur die ausdrücklich aufgeführten Synonyme in ihrer Bedeutung erweitert werden sollen; stünde kein `only` da, würde die gesamte ursprüngliche Liste, also auch *„küssen"* etc. auf die Aktion `##love` umgeleitet. Das ginge zu weit.

Das Schlüsselwort `replace` schließlich sorgt dafür, dass die neue Grammatikzeile anstelle der vorher definierten verwendet wird. Die Verbindung von *„liebe"* mit der Aktion `##kiss` wird also völlig unterbrochen. Andere Möglichkeiten böten hier die Schlüsselwörter `first` (die neudefinierten Zeilen werden vom Parser vor den alten durchgesehen) und `last` (die neuen Grammatikzeilen werden an die alten angehängt).

Die vordefinierten Routinen für die neuen Aktionen bieten nichts überraschendes:[22]

```
[ LoveSub;
     print_ret (Gder) noun, " ist dir nicht zur Gefährtin bestimmt.";
];

[ GnothiSub;
    "Das ist leichter gesagt als getan ...";
];
```

[22] `LoveSub` ist im Original viel länger, da ich den ganzen Code hineingeschrieben habe, der eigentlich bei Eva hätte stehen müssen.

2.13.3 Letzte Feinheiten

Ein zusätzliches Synonym zu einer Verbendefinition hinzuzufügen ist so einfach wie Blumen zu pflücken:

```
verb 'pflueck' = 'nimm';
```

Ein anderes Problem macht etwas mehr Arbeit. Die Obstbäume im Garten sehen für den Spieler so aus, als trügen sie Obst. Das Obst ist aber – wegen der besseren Sichtbarkeit – ein selbständiges Objekt und nicht `child` des jeweiligen Baumes. Das bringt den Parser in Verwirrung, wenn ein Satz wie *„Pflücke einen Pfirsich vom Obstbaum"* eingegeben wird – denn er parst dann nach der Aktion `##remove`, die normalerweise als zweites Objekt einen `supporter` oder `container` erfordert (siehe 4.2 auf Seite 121). Ein Obstbaum ist aber weder das eine noch das andere, denn die Früchte liegen weder auf ihm drauf noch in ihm drin.

Das kann uns indessen nicht schrecken: Erst definieren wir zwei Funktionen, die `true` oder `false` zurückgeben, wenn die (implizit übergebene) Variable `noun` auf Obst bzw. auf einen Obstbaum zeigt:[23]

```
[ ObstP;
    if ((noun ofclass(Obst)) || (noun==Apfel)) rtrue;
    else rfalse;
];

[ BaumP;
    if (noun ofclass(Baum)) rtrue;
    else rfalse;
];
```

Schon können wir die Grammatik für *„nehme"* und alle verbundenen Synonyme ergänzen und *„Pflücke einen Pfirsich vom Obstbaum"* wird korrekt erkannt.

```
extend 'nimm'
    * noun = ObstP 'von'/'aus'/'vom' noun = BaumP -> Remove;
```

Das Konstrukt `noun = ObstP` bildet *ein* Grammatiktoken, das für alle Objekte steht, für die die Funktion `ObstP` den Rückgabewert `true` hat (also für alle Obstarten und den Apfel). Mutatis mutandis steht das Token `noun = BaumP` für alle Bäume.

[23] Der heilige Nikolaus (Wirth) möge mir die explizite Schreibweise der Funktionen mit `if...else` verzeihen! Aber Hand aufs Herz, liebe Leserin: Weißt du, ohne oben in 2.6.3 auf Seite 29 nachzuschauen, ob die Routinen implizites `true` oder `false` zurückgeben würden?

2.14 Allerlei Dämonen

Dämonen (*daemons*) heißen in UNIX-Betriebssystemen Dienstprogramme, die unabhängig von Aktionen des Benutzers im Hintergrund laufen. Zu den bekanntesten gehört der *cron-daemon*, der dafür sorgt, Programme in bestimmten Zeitabständen auszuführen.

2.14.1 Ausführung eines Dämons bei jedem Spielzug

Inform ermöglicht es, für jedes Objekt einen eigenen Dämonen zu schreiben: nämlich als eine Routine in der *property* `daemon` des Objekts. Diese Routine wird mit dem Befehl `StartDaemon(Objekt);` gestartet; sie läuft dann bei jedem Spielzug einmal ab, bis sie mit `StopDaemon(Objekt);` wieder schlafen gelegt wird. Ein solcher Dämon des Objekts `Eva` sorgt etwa dafür, dass Eva der Spielerfigur Adam im ganzen Bereich des Gartens (d.h. dort, wo Gras wächst wie oben 2.5 auf Seite 25 beschrieben) nicht von der Seite weicht:

```
daemon [;
    if (Eva notin parent(player))
        if (Gras in parent(player))
        {

            if (Eva in Gras)
            {
                move Eva to parent(player);
                "Auch Eva steht von der Wiese auf.";
            }
            else
            if (parent(Eva) ofclass Wasser)
            {
                move Eva to parent(player);
                "Auch Eva kommt aus dem Wasser.";
            }
            else
            {
                move Eva to parent(player);
                "Eva folgt dir nach.";
            }
        }
        else
        {
            if (player in Gras)
            {
```

```
                move Eva to Gras;
                "Eva legt sich zu dir ins Gras.";
            }
            if (parent(player) ofclass Wasser)
            {
                move Eva to parent(player);
                print_ret "Auch Eva steigt in ", (den)
                    parent(player),".";
            }

        }
    ],
```

Weitere Dämonen sorgen dafür, dass der Apfel nicht einfach liegengelassen werden kann und (in Verbindung mit einem Zähler und einer `switch`-Anweisung) für die sich über mehrere Züge erstreckenden Verhaltensfolgen der Schlange und des Yehova-Objekts.

2.14.2 Ich spüre deine Nähe . . .

Besonders praktisch ist ein Dämon, der automatisch startet, wenn der Spieler in die Nähe des Objekts kommt und ebenso automatisch wieder anhält, wenn er sich entfernt hat. Dafür ist die *property* `each_turn` vorgesehen. Eine dort enthaltene Routine wird bei jedem Zug ausgeführt, falls das Objekt in Sichtweite *(scope)* des Spielers ist. Wir haben diesen Mechanismus schon in 2.7.2 auf Seite 34 dazu eingesetzt, bei jedem Zug die Gartentüre auf die richtige Seite zu bringen. Man kann damit aber mehr machen, z.B. eine unwiderstehlich impertinente Versuchung programmieren:

```
each_turn [;
    if ((Apfel in Eva) && (Eva has erkenntnis))
       "^Eva hält in ihrer Hand den Apfel vom
        Baum der Erkenntnis. Sie bietet ihn dir an: ~Iss auch du
        von dem Apfel! Wir werden wie ER sein und Gut und Böse
        unterscheiden können!~";
],
```

Diese *property* des Objekts `Eva` sorgt dafür, dass Eva dem Spieler bei jedem Zug den Apfel anbietet, solange Eva in Sichtweite der Spielerfigur ist.

2.14.3 „Und da die sieben Tage vergangen waren . . . ”

Eine weitere Unterart der Dämonen ist der Timer. Er zählt von einem Anfangswert bei jedem Zug um 1 herunter und führt die mit ihm verbundene Routine des Objekts

(enthalten in der *property* `time_out`) genau einmal aus, nämlich wenn er abgelaufen ist. Ein Timer sorgt z.B. dafür, dass Eva sich unaufhaltbar zur Schlange hingezogen fühlt:

```
time_out [;
    if (Eva notin Mitte)
    {
        move Eva to Mitte;
        print "Eva wird plötzlich unruhig. Ihr Blick wirkt nach
           innen gekehrt, als lausche sie einer Stimme, die nur
           sie hören kann.^^
           Schließlich entfernt sie sich mit großen Schritten
           zur Mitte des Gartens hin.^";
    }
],
```

Dieser Timer muss nicht unbedingt im Objekt `Eva` eingebaut sein. Da es sich ja eigentlich um die magische Anziehungskraft der Schlange handelt, habe ich ihn in das Objekt `Schlange` gesetzt. Gestartet wird er also mit dem Befehl `StartTimer(Schlange,13);` und zwar bereits bei Evas Erschaffung. Er schlägt nach 13 Zügen zu und holt Eva zur Schlange in die Mitte des Gartens.[24] Während er läuft, kann die Autorin den Stand in `Schlange.time_left` nachsehen. Ein laufender Timer kann mit `StopTimer(Objekt);` auch wieder angehalten werden.

2.15 „... daß sein ganzes Alter ward neunhundertunddreißig Jahre, und starb."

Spoilerwarnung: Lösen Sie *Eden*, bevor Sie diesen Abschnitt lesen.

2.15.1 Der Herr sieht alles (und gibt Punkte)

Aus dem Rätselspiel stammt die Konvention, Handlungen des Spielers mit Punkten zu belohnen und eine Gesamtwertung anzugeben. Die einfachste Vorgehensweise ist es, für das Einsammeln bestimmter Gegenstände (Schätze) und für das Betreten bestimmter Räume Punkte zu vergeben. Die entprechenden Objekte (Räume und bewegliche Objekte) werden einfach mit dem Attribut `scored` versehen. Die insgesamt erreichbaren Punkte müssen in der Konstante `MAX_SCORE` angegeben werden. Den Rest (Buchhaltung und Ermitteln der Gesamtpunktzahl) besorgt die Library ganz von alleine.

[24] Das eigentliche „Verführungswerk" erledigt die Schlange dann standesgemäß mit ihrem Dämon. Ihr Dämon sorgt anschließend auch dafür, dass sie sich (samt Dämon) selbst aus dem Spiel entfernt.

Wem das nicht reicht, der kann außerdem noch einzelne zu lösende Aufgaben *(tasks)* definieren. *Eden* hat deren 8, numeriert von 0 bis 7. Jede Aufgabe bis auf eine wird mit 7 Punkten bewertet, eine nur mit einem Punkt[25], so dass sich eine Gesamtsumme von 50 Punkten ergibt. Das erfordert folgende Definitionen:

```
Constant TASKS_PROVIDED;              ! schaltet die Aufgabenrechnung ein
Constant NUMBER_TASKS = 8;            ! Anzahl der Aufgaben
Constant MAX_SCORE = 50;              ! höchste erreichbare Punktzahl
Array task_scores -> 7 7 7 7 7 1 7 7; ! Der letzte lausige Punkt bei Aufgabe 5
```

Die Bewertung erfolgt durch Aufruf der Funktion `achieved()` mit der Nummer der jeweils gelösten Aufgabe als Argument, wie wir es oben 2.7 auf Seite 32 beim Türöffnen gesehen haben. Die Library achtet auch hier darauf, dass Punkte bei Wiederholung der Aktion nicht doppelt vergeben werden.

Das Ganze ist aber öde, wenn wir dem Spieler nicht mitteilen können, für welche Aktionen er die Punkte bekommen hat. Für die scored-Objekte macht das die Library automatisch; für die Aufgaben ist etwas Arbeit nötig, nämlich das Vorbereiten einer Routine mit dem Namen `PrintTaskName` und geeigneten Ausgabetexten:

```
[ PrintTaskName task_number;
    switch (task_number)
 {
     0: "für das Erfinden eines neuen Namens."; ! 'Schlange', s.o.
     1: "für das Finden einer zweiten Primel.";
     2: "als Trost für den Verlust einer Rippe.";
     3: "für das Erkennen deines Weibes Eva.";
     4: "für die Erkenntnis von Gut und Böse.";
     5: "für den letzten lausigen Punkt";
     6: "für das Überwinden einer Grenze.";
     7: "für die Rose.";
     }
];
```

2.15.2 Tod und Spielende

Ob die Spielerfigur am Leben ist oder nicht, entscheidet die Variable `deadflag`. Im „Normalbetrieb" steht sie auf `0`, entsprechend dem Wert `false`. Es genügt, der Variablen den Wert `1` oder `true` zuzuweisen, um das Spiel unrühmlich zu beenden. Zweckmäßigerweise sollte noch irgend ein sinniger Text nach dem Muster *„Du trittst auf eine Schlange, die ausprobiert, ob sie sich für die unsanfte Behandlung revanchieren kann. Sie*

[25] Das ist der in vielen Spielen zu findende (bzw. traditionell nur schwer zu findende) „letzte lausige Punkt".

kann." ausgegeben werden. Die Library sorgt dann nur noch für eine zentrierte, mit Sternchen versehene Todesnachricht und beendet den Spieldurchlauf nach Ablauf des aktuellen Spielzugs.

Der Spieler will das Spiel aber gewinnen. Auch das läuft seltsamerweise über `deadflag` und zwar, indem man es auf den Wert `2` setzt. (Zweckmäßigerweise sollte noch irgend ein sinniger Text nach dem Muster *„Auf deinem Konto liegt jetzt der gesamte Schatz der Chubakka Maru."* ausgegeben werden. Die Library sorgt dann nur noch für eine zentrierte, mit Sternchen versehene *Gewinnmitteilung* und beendet den Spieldurchlauf.)

Das reicht für das klassische Todesfallen-und-Schatzsucher-Adventure völlig aus. In *Eden* allerdings gibt es keine Todesfalle, dafür aber mehrere verschiedene Enden, von denen keines „gewonnen" oder „gestorben" ist. Das ist kein Problem: `deadflag` kann auch auf andere Werte ab `3` aufwärts gesetzt werden. Dann gibt es keine vorgefertigte Todes- oder Gewinnbenachrichtigung; die abschließende Grußkarte darf man sich in einer Routine namens `DeathMessage` selber anfertigen (Zentrieren und Sternchen macht die Library aber dennoch):

```
[ DeathMessage;
    if (deadflag == 3) print "Du hast deine Chance vertan.";
    if (deadflag == 4) print "Du hast dein Weib Eva verraten.";
    if (deadflag == 5) print "Du bist zur Hölle gefahren.";
    if (deadflag == 6) print "Du bist im Garten Eden geblieben";
    if (deadflag == 7) print "Du hast deine Bestimmung erfüllt.";
];
```

Zum Schluss noch ein Beispiel für das Setzen von `deadflag` mit der Ausgabe eines – mehr oder weniger – sinnigen Textes:

```
deadflag = 6;
"^Du erkennst dein Weib Eva und ihr werdet eine Ewigkeit
 lang glücklich und zufrieden im Paradies leben.";
```

Beachte: Der Spielzug wird – einschließlich des auszugebenden Textes – noch ausgeführt. Erst nach vollständiger Ausführung wird `deadflag` kontrolliert, so dass es durchaus *vor* dem Abschlusstext gesetzt werden darf.

2.16 Eden ungekürzt

Ursprünglich hatte ich das Spiel „Eden – ein interaktiver Anfang" als reines Programmierbeispiel für den vorliegenden Text entworfen. Dabei ist es nicht geblieben, denn ich habe es beim Textfire.de-Wettbewerb 2002 als Beitrag eingereicht. Also gilt auch für Eden: *„This tale grew in the telling"* – zwar ist keine sechsbändige Geschichte von

Mittelerde daraus geworden, es ist noch immer eine IF-Kurzgeschichte – aber das erzählerische Moment einerseits und andererseits die Notwendigkeit, das Spiel unter Zeitdruck lange vor dem „Informissimus" fertigzustellen, führen zu zwei wesentlichen didaktischen Mängeln:

* „Eden" enthält weit mehr Code als es für ein Beispiel notwendig oder tunlich gewesen wäre. Die Programmbeispiele in den vorhergehenden Abschnitten sind deshalb alle auf das Wesentliche „zurückgeschnitten" worden.
* „Eden" enthält in Überfülle Code, der in letzter Minute hinzugefügt wurde, nicht die beste Lösung für das Problem darstellt, nicht sauber durchdacht, manchmal wahrscheinlich sogar überflüssig, kurz: alles andere als „beispielhaft" ist. (Insofern ist es ein *realistisches* Beispiel.)

Ich wollte ja eigentlich den Quelltext von Eden für die Veröffentlichung überarbeiten und vielleicht sogar mit „Noweb" in publizierbare Form bringen. Nur wäre dann dieser Text nie erschienen – jedenfalls nicht zusammen mit dem Quellcode. Vor die Entscheidung gestellt, entweder den Quellcode entgegen der Ankündigung wegzulassen, die Veröffentlichung des Informissimus auf den St.-Nimmerleins-Tag zu verschieben oder mich als schlechter Programmierer zu outen, habe ich dann doch die letzte Möglichkeit gewählt.

Sei's drum. Zum Quelltext gehören drei Dateien, die Sie entweder schon haben oder die unter `http:/if.frob.de/downloads/` zu finden sind: eden.inf (das Hauptprogramm), eden-hints.inf (das Hilfemenü) und eden-hints.raw (die Quelldatei dafür). In Abwandlung eines alten Spruchs: *„Es sind die schlechten Programmierer, die die guten Bücher schreiben – die guten Programmierer schreiben nur Programme!"*

3 Der Baum der Erkenntnis: Inform-Kurzreferenz

Dies ist nur eine Kurzreferenz ohne den Anspruch auf Vollständigkeit. Wer mehr wissen will, braucht das „Designer's Manual" von Graham Nelson. Die vierte Auflage vom Mai 2001 (zitiert als DM[4]) ist als pdf-Datei erhältlich: `http://www.ifarchive.org/if-archive/infocom/compilers/inform6/manuals/designers_manual_4.pdf`.

Inform-Code und Schlüsselwörter habe ich im Folgenden in `blauer Schreibmaschinenschrift` gesetzt, Platzhalter innerhalb von Codebeispielen stehen in LILA KAPITÄLCHEN.

3.1 Inform

3.1.1 Elementare Syntax

„Inform sieht seinen Quellcode als eine Auflistung von Dingen, die es sich ansehen soll und die durch Strichpunkte ; getrennt sind." (DM[4], p. 7)

Einfache Datentypen und Literale

Byte ist die kleinste adressierbare Speichereinheit; es hat 8 Bits und kann ein Zeichen *(Charakter)* oder eine Zahl zwischen 0 und 255 darstellen. Meistens arbeitet man mit der Zusammenfassung zweier Bytes zu einem *Word*.

Word hat 16 Bits (im Z-Code, unter Glulx sind es 4 Bytes - 32 Bits) und wird für die Darstellung von Adressen, Zeigern (Wörterbucheinträgen) oder Zahlen benutzt.

Zahlen (ein *Word*) können dezimal (`-32768` bis `+32767`), hexadezimal (`($0` bis `$FFFF`) oder binär (`$$0` bis `$$1111111111111111`) angegeben werden.

Charakter ist ein einzelner Buchstabe in einfachem Hochkomma: `'a'`

String ist eine Zeichenkette aus bis zu 4000 Zeichen in Anführungszeichen. Eine Tilde `~` wird beim Ausgeben in ein Anführungszeichen, ein Caret `^` in eine Zeilenschaltung umgesetzt. Die Tilde selbst erhält man mit `@@126`, das Caret (zu deutsch „Dächle") mit `@@94`, den Backslash `\` mit `@@92`, den Klammeraffen selbst mit `@@64`.

Latin-1-Zeichen sind zulässig, es ist nicht mehr notwendig, deutsche Sonderzeichen besonders zu behandeln. Ausnahme: Im `name`-Property müssen Umlaute und ß als ae oe ue ss transkribiert werden.

Wörterbucheinträge *(dictionary words)* haben neun[1] signifikante Zeichen (Achtung: Umlaute und ß zählen als zwei!) und werden in einfache Hochkommas eingeschlossen: `'Haus'`. Ein Wörterbucheintrag mit Länge 1 muss von einem `Character` unterschieden werden durch eine der Schreibweisen `"a"` (nur im `name`-*property* zulässig, veraltet, bitte nicht mehr benutzen) oder besser `'a//'`. Intern werden Wörterbucheinträge als Zeiger vom Typ `word` verarbeitet; der Zeiger verweist auf den Eintrag der Zeichenkette im Wörterbuch. Wörterbucheinträge können auch Leerstellen und Satzzeichen enthalten; solche Einträge werden aber vom Parser bei der Eingabe nicht erkannt, weil er Leerstellen etc. als Trennzeichen verwendet.

Aktionen erkennt Inform (und der Leser) an zwei vorangestellten Gattern: `##Take` oder - in eine Anweisung eingeschlossen - an den umschließenden spitzen Klammern: `<take babelfisch>`

Bezeichner müssen mit einem Buchstaben beginnen, können Ziffern und den Unterstrich enthalten und bis zu 32 Zeichen lang sein.

Groß- und Kleinschreibung sind irrelevant[2] mit zwei Ausnahmen:

1. Bestimmte Schlüsselwörter (die Anweisungen) müssen immer klein geschrieben sein: `box break continue do until font (on off) for give if else jump move...to new_line objectloop print print_ret remove return rfalse rtrue spaces string style (roman bold underline reverse fixed) switch default while`
2. Die Namen von Dateien, die mit `include` eingebunden werden sollen, müssen korrekte Groß-/Kleinschreibung aufweisen, wenn das verwendete Betriebssystem (wie alle vernünftigen Betriebssysteme) Groß-/Kleinschreibung unterscheidet.

Konstante, Variablen und Arrays

Konstante und Variablen sind nicht typisiert, d.h. sie können jeden Datentyp (insbesondere auch den Typ Object) enthalten.

Konstante werden mit dem Schlüsselwort `Constant` definiert; bei der Definition kann ein Wert zugewiesen werden:

[1] Auch in Glulx sind es – aus Kompatibilitätsgründen mit der „alten" Inform-Library – nicht mehr als neun.
[2] Ich verwende an vielen Stellen CamelCase, weil es die Lesbarkeit verbessert.

```
Constant NAME_DER_KONSTANTE;
Constant NAME_DER_KONSTANTE = AUSDRUCK;
```

Inform kennt bereits die Konstanten `false` (=0) und `true` (=1); logisch wahr ist aber auch jeder andere von 0 verschiedene Wert. Außerdem noch `nothing` (=0) und `null` (=-1). `Null` steht für ein undefinierte Aktion oder Property, `nothing` für „kein Objekt".

Variable Globale Variable werden wie Konstante definiert, jedoch mit dem Schlüsselwort `Global`. Lokale Variable können innerhalb von Objekten (als *properties*, s.u.) und im Kopf von Routinen definiert werden.

Arrays Mit dem Schlüsselwort `array` können Felder bestimmter Datentypen eingerichtet werden. Je nach der Elementgröße (Byte oder Word) und nach der verwendeten Adressierung verwendet Inform vier Arraytypen. Bei der Adressierung ist zu unterscheiden zwischen „einfachen" Arrays, die bei Länge N von 0 bis N-1 referenziert werden und Arrays, die zum Speichern von Text gedacht sind und bei Länge N von 1 bis N adressiert werden, wobei im „nullten" Element Längeninformationen gespeichert sind zu Beginn vorbelegt mit der Länge N des Arrays). Allen Arrays ist gemeinsam, dass sie eine bei der Definition festgelegte und später nicht mehr änderbare Länge haben.

Die Definition eines Array erfolgt nach folgendem Muster:

```
Array NAME_DES_ARRAY PFEIL DATEN;
```

im Einzelnen:

`Array` das Schlüsselwort `Array`

`NAME_DES_ARRAY` ein Bezeichner für den Namen

`PFEIL` für jeden der vier Typen unterschiedlich:

- **`->`** für ein von 0 bis n-1 adressiertes Array aus Bytes *(byte array)*
- **`-->`** für ein von 0 bis n-1 adressiertes Array aus Words (2 Bytes lang, s.o.) *(word array)*
- **`string`** für ein von 1 bis n adressiertes Array aus Bytes *(string array)*
- **`table`** für ein von 1 bis n adressiertes Array aus Words *(table array)*

`DATEN` entweder eine einzelne Zahl, die dann die Länge des zu definierenden Array festlegt;
oder eine Liste von Werten (Ausdrücken), mit denen die einzelnen Elemente des Array vorbelegt werden (die Länge ist dann gleich der Länge dieser Liste)
oder schließlich ein String, der zeichenweise auf die einzelnen Elemente des Array verteilt wird (die Länge ist dann gleich der Länge dieses Strings).

Der Zugriff auf ein Array erfolgt mit NAME_DES_ARRAY->AUSDRUCK für die Typen *Byte* und *String* bzw. mit NAME_DES_ARRAY-->AUSDRUCK für die Typen *Word* und *Table*. Die Auswertung des Ausdrucks muss eine Zahl innerhalb der Arraygrenzen ergeben.

Die `name`-*property* eines Objekts (3.2.7 auf Seite 98) besteht tatsächlich aus einem *word array,* in dem Referenzen auf Wörterbucheinträge abgelegt sind. Zur Adressierung der `name`-*property* als Array muss ein `&` vorangestellt werden, z.B:

```
objekt.&name-->4='Haus';
```

Operatoren

Zuweisungsoperator ist das einfache Gleichheitszeichen `=`.

Inform kennt binäre Operatoren für die Grundrechenarten `+ - * /` und die modulo-Funktion `%`; außerdem bitweises and `&` und or `|`.[3] Unäre Operatoren sind das einfache Minuszeichen, das bitweise not `~` und die Pre- und Postfixoperatoren `++` und `--`.[4]

Inform kennt die normalen binären Vergleichsoperatoren gleich `==`, ungleich `~=`, größer `>`, größer oder gleich `>=`, kleiner `<`, kleiner oder gleich `<=`, logisches and `&&` und logisches or `||`; außerdem ein unäres logisches not `~~`.

Spezielle binäre Vergleichsoperatoren sind:

`ofclass` testet das links stehende Objekt auf Zugehörigkeit zur rechts angegebenen Klasse

`in` und

`notin` testen das links stehende Objekt darauf, ob es *child* des rechts stehenden Objekts ist oder nicht

`provides` testet, ob das links stehende Objekt die rechts angegebene *property* besitzt;

`has` und

`hasnt` testen, ob das links angegebene Objekt das rechts angegebene *attribute* besitzt oder nicht.

[3] Bitweise heißt, dass jedes einzelne Bit zweier (bei *and* und *or*) bzw. eines (bei *not*) Wertes mit dem entsprechenden Operator verknüpft wird: Sei `x=$$0010110;` dann ergibt `~x` den Wert `$$1101001`. Anders das *logische not*: `~~x` ergibt den Wert `false` (jeder von `0` verschiedene Wert ist `true`).

[4] Die Pre- und Postfixoperatoren addieren (`++`) bzw. subtrahieren (`--`) jeweils 1 zum/vom Wert einer Variablen und weisen der Variablen das Ergebnis wieder zu. (Kürzer: Sie inkrementieren bzw. dekrementieren die Variable um den Wert 1.) Dies geschieht entweder vor der Auswertung des gesamten Ausdrucks (wenn der Operator als Prefix *vor* der Variablen steht) oder danach (wenn er als Postfix *nach* der Variablen steht).

`x=i++;` weist erst `x` den Wert von `i` zu und erhöht danach `i` um `1`. `y=--i;` vermindert zuerst `i` um `1` und weist danach `y` den neuen Wert zu.

Eine weitere Besonderheit ist das Schlüsselwort **or**, mit dem auf der rechten Seite eines Vergleichsoperators ganze Listen gebildet werden können:

```
if (messer in tisch or schrank or schublade) ...
```

Logische Ausdrücke werden immer von links nach rechts abgearbeitet und abgebrochen, sobald das Ergebnis feststeht.

Ausgewertet werden Operatoren in der Reihenfolge ihrer Priorität, ansonsten von links nach rechts:

Der Superclass-Operator `::`
Der Punkt `.` als Auswahloperator in `Objekt.property`
Die Klammern `()` im Aufruf einer Routine
`.&` und `.#` als Auswahloperatoren eines *property array*
`++` und `--`, die Prä- und Postfixoperatoren
Das unäre Minuszeichen `-`
Die Pfeile `->` und `-->` zur Referenzierung eines Arrayelements
Die mathematischen Punktrechnungsoperatoren `*`, `/` und `%`, sowie die bitweisen Operatoren `&`, `|` und `~`
Die mathematischen Strichrechnungsoperatoren `+` und `-`
`or` für die Auflistung logischer Alternativen
Alle oben beschriebenen logischen Operatoren und Vergleichsoperatoren mit Ausnahme von
`&&`, `||` und `~~`
Der Zuweisungsoperator `=`
Das Komma `,` als Reihungsoperator für verschiedene aufeinanderfolgende Zuweisungen.

3.1.2 Objekte und Klassen

Alles, was der Spieler zu sehen bekommt ist ein Objekt: Räume, Gegenstände, Lebewesen – auch die Spielerfigur selbst.

Klassen sind vordefinierte Muster für mehrere Objekte, die dann die vordefinierten Eigenschaften „ihrer" Klasse erben.

Alle Objekte, die sich in einem Raum befinden, sind zusammen mit dem Raum selbst Knoten in einem Baum, dem *object tree*. Die Anfangsposition der Objekte in diesem Baum wird normalerweise bei der Definition der Objekte festgelegt; sie kann sich im Spiel dynamisch verändern. Ein Objekt ohne Elternobjekt *(parent)* ist entweder ein Raum oder gerade nicht im Spiel.

Definition von Objekten und Klassen

Ein Objektdefinition enthält bzw. kann enthalten:

* den Namen einer Klasse, deren Eigenschaften das Objekt erbt oder das Schlüsselwort `Object`
* (optional) einen oder mehrere Pfeile `->`, die anzeigen, dass das Objekt im *object tree* unterhalb des zuletzt definierten Objekts steht, das *mit einem Pfeil weniger* definiert wurde (Objekte mit gleichviel Pfeilen stehen also auf gleicher Ebene, sie sind Geschwister, *siblings*);
* einen Bezeichner als Namen des Objekts;
* (optional) in Anführungszeichen den *extended name*, d.h. den Namen, der vom Spiel ausgegeben wird – **im Deutschen macht das nur bei Räumen Sinn, deren Name nicht dekliniert werden muss; der *extended name* von Objekten wird in der Regel aus den *Properties*** `adj, short_name` **und** `post` **zusammengesetzt.**
* (optional und nur, wenn nicht implizit durch Pfeile angegeben) den Namen des Objekts, das im *object tree* einen Knoten weiter oben sitzt, also das Elternobjekt *(parent)*;
* (optional) eine mit dem Schlüsselwort `Class` eingeleitete Liste von Klassen, deren Eigenschaften das Objekt erbt;
* (optional) eine mit dem Schlüsselwort `with` eingeleitete Liste von (öffentlichen) Eigenschaftsnamen *(properties)* und den dazugehörenden Eigenschaften d.h. von Variablen und ihren Werten; jeweils mit einem abschließenden Komma,
* (optional) eine weitere mit dem Schlüsselwort `private` eingeleitete Liste von Eigenschaftsnamen *(properties)* und den dazugehörenden Eigenschaften; jeweils mit einem abschließenden Komma,
* (optional) eine mit dem Schlüsselwort `has` eingeleitete Liste von gesetzten oder ungesetzten Attributen;
* einen Strichpunkt zum Schluss.

Die mit `class, with, private` und `has` eingeleiteten Abschnitte können in beliebiger Folge stehen, sie werden dann durch Kommata getrennt.

Eine Klassendefinition wird begonnen mit dem Schlüsselwort `Class` gefolgt vom Namen der Klasse und optional einer Zahl in runden Klammern. Diese Zahl gibt an, wieviele Instanzen der Klasse später dynamisch erzeugt werden können. Danach können – wiederum in beliebiger Reihenfolge – mit `class, with, private` und `has` eingeleitete Abschnitte folgen.

Die Funktion `metaclass()` liefert die Klasse eines Objekts zurück.

Erbschaften und Erbfolge

Die mit `with` und `private` definierten Eigenschaften sind Variablen, die samt ihren in der Klassendefinition definierten Werten vererbt werden. Auf sie kann jederzeit zugegriffen werden, indem die Bezeichner des Objekts und des *Properties* mit einem Punkt verknüpft werden: `OBJEKTNAME.PROPERTY` . Statt des Objektnamens kann *innerhalb* der Objektdefinition selbst auch `self` verwendet werden. Für Code in Klassendefinitionen ist das unumgänglich, weil die Klasse ja nicht weiß, wie ihre Instanz später heißen wird.

Im Normalfall handelt es sich bei diesen Eigenschaften um Routinen. Sie können auch mit Parametern aufgerufen werden: `x = Topf.Kochen(Apfel);` bedeutet, dass man an das Objekt `Topf` eine in dessen *Property* `Kochen` definierte Nachricht mit dem Inhalt `Apfel` schickt und den Rückgabewert in der Variablen `x` ablegt (unterstellt, wir hätten eine sinnvolle *Property* `Kochen` für den `Topf` definiert, könnte das dann etwa ein Objekt `Kompott` sein).

Werden vordefinierte Eigenschaften der Klasse in der Objektdefinition nochmals mit Werten belegt, so werden die in der Klasse vordefinierten Eigenschaften überschrieben, **wenn nicht vor ihrer ersten Verwendung die Eigenschaft als additiv deklariert wurde:**

```
Property additive NAME_DER_NEUEN_PROPERTY;
```

Neue *Common properties* lassen sich auf dieselbe Weise definieren:

```
Property NAME_DER_NEUEN_PROPERTY;
```

Bei additiven *Properties* – in der Regel sind es Routinen – wird *zuerst der zuletzt definierte Teil* ausgeführt. Gibt dieser `false` zurück, wird in der „Ahnenreihe" eine Klasse weiter zurückgegangen, solange bis `true` zurückgegeben wird oder die oberste Klasse erreicht ist. Additiv sind die vordefinierten *common Properties* `after, before, describe, each_turn, life, name, time_out`.

Attribute werden immer additiv vererbt. Dies führt bei `male, female, neuter` zu einem kleinen Problem; siehe 3.2.6 auf Seite 92.

An den Code der überschriebenen *Properties* der Superklasse kommt man mit dem Superclass-Operator `::` heran: Sei z.B. `Dampftopf` von der Klasse `Topf` hergeleitet, habe aber die Property `Kochen` überschrieben, so kann der Dampftopf mit

```
Dampftopf.Topf::Kochen(Apfel)
```

den Apfel genauso kochen wie jeder andere Topf auch. Zumeist wird der Superclass-Operator aber verwendet werden, um innerhalb der *Property*, die man gerade überschreibt, den ursprünglichen Code der Klasse aufrufen zu können.

Traversieren des *object tree*

Zur Ermittlung der Umgebung eines Objekts dienen die Funktionen

`parent()` liefert das im Baum übergeordnete Objekt (oder `nothing`, wenn es keines gibt)

`children()` liefert die Zahl der nachgeordneten Elemente (kann auch 0 sein)

`child()` liefert das erste nachgeordnete Objekt (oder `nothing`),

`sibling()` liefert das nächste gleichgeordnete Objekt (oder `nothing`),

`IndirectlyContains(``WALD,BLATT`**`)`** prüft, ob `WALD` im *object tree irgendwo* oberhalb von `BLATT` steht; der Rückgabewert ist kein Objekt, sondern vom Typ *boolean*.

Manipulieren des *object tree*

Verändert wird der Baum durch die Befehle

`move` `MEIN_OBJEKT` **`to`** `NEUER_PARENT`**`;`** setzt ein Objekt samt allen seinen Kindern an eine neue Stelle.

`remove` `MEIN_OBJEKT`**`;`** nimmt das Objekt samt allen seinen Kindern aus dem Spiel; löscht es aber nicht, es kann jederzeit mit `move ... to` wieder eingesetzt werden.

Nachrichten an verschiedene Klassen und dynamische Erzeugung von Objekten

`OBJEKT.PROPERTY(AUSDRUCK1,AUSDRUCK2,...);` ist der Aufruf einer in einer *property* eines Objekts enthaltenen Routine,

`ROUTINE.call(AUSDRUCK1,AUSDRUCK2,...);` ist in gleicher Weise der Aufruf einer selbständigen Routine.

`STRING.print();` gibt den String aus und

`STRING.print_to_array(NAME_DES_ARRAYS);` kopiert ihn in einen Array.

`KLASSE.remaining()` gibt an, wieviele Instanzen einer Klasse noch mit

`KLASSE.create()` dynamisch ins Leben gerufen werden können. Damit das überhaupt geht, muss bei der Klassendefinition eine Zahl größer 0 angegeben worden sein.

`KLASSE.destroy(OBJEKT)` löscht das Objekt und gibt den Platz wieder frei; nur mit `create` erzeugte Objekte können gelöscht werden!

`KLASSE.recreate(OBJEKT)` setzt ein Objekt auf seine Anfangswerte zurück.

`KLASSE.copy(OBJEKT1,OBJEKT2)` kopiert alle Eigenschaften der Klasse von Objekt 1 nach Objekt 2.

3.1.3 Schleifen und andere Konstrukte

if und else

```
if (BEDINGUNG)
     ANWEISUNGSBLOCK1
else
     ANWEISUNGSBLOCK2
```

Ist die Bedingung `true`, wird ANWEISUNGSBLOCK1 ausgeführt, sonst ANWEISUNGSBLOCK2. Der `else`-Zweig ist optional, er bezieht sich immer auf das letzte `if`. (Abweichende Konstruktionen sind durch 'Blocken' natürlich möglich.)

Ein ANWEISUNGSBLOCK besteht entweder aus einer einzelnen, mit Semikolon terminierten Anweisung oder aus mehreren `{` in geschweifte Klammern eingeschlossenen `}` Anweisungen. Die Bedingung muss immer in *runden* Klammern stehen!

switch

```
switch (AUSDRUCK)
{
    KONSTANTE1: ANWEISUNGEN ... ;
    KONSTANTE2, KONSTANTE3: ANWEISUNGEN ... ;
    KONSTANTE4 to KONSTANTE5: ANWEISUNGEN ...;
    ...
    default: ANWEISUNGEN ... ;
}
```

Der Ausdruck wird ausgewertet, entspricht sein Wert einer der Konstanten, wird der zugehörige Anweisungszweig ausgeführt; sonst der `default`-Zweig. Der `default`-Zweig ist optional. Der auszuwertende Ausdruck muss in runden Klammern stehen. Die Anweisungen der einzelnen Zweige müssen nicht 'geblockt' werden.

Schleifen

Eine Schleife bewirkt die wiederholte Ausführung eines Befehls oder einer Befehlsfolge. Die Schleifenkonstrukte folgen wie `if` und `switch` ebenfalls der C-Syntax (mit Ausnahme des Inform-spezifischen `objectloop`). Jede Schleife kann mit dem Befehl `break;` unmittelbar verlassen werden. Der Befehl `continue;` springt unmittelbar an den Schleifenanfang und beginnt mit dem nächsten Durchlauf. Die Schleifenbedingung steht immer in runden Klammern, der *Anweisungsblock* ist auch hier entweder eine einzelne, mit Semikolon abgeschlossene Anweisung oder eine in geschweifte Klammern eingeschlossene Folge solcher Anweisungen.

while definiert eine Schleife mit Anfangsbedingung, die ausgeführt wird, *solange* die Auswertung der Bedingung `true` ergibt:

```
while(AUSDRUCK) ANWEISUNGSBLOCK
```

do ... until definiert eine Schleife mit Endbedingung, die ausgeführt wird, *bis* die Auswertung der Bedingung `true` ergibt:

```
do ANWEISUNGSBLOCK until (AUSDRUCK);
```

for definiert eine Zählschleife (d.h. eine Schleife, bei der die Anzahl der Wiederholungen durch eine Zählvariable gesteuert wird) nach dem von C bekannten Muster (Vorsicht, Doppelpunkt statt Strichpunkt!):

```
for (INITIALISIERUNG : BEDINGUNG : VERÄNDERUNG) ANWEISUNGSBLOCK
```

`INITIALISIERUNG` enthält üblicherweise einen bei Beginn einmalig auszuführenden Zuweisungsbefehl an eine Variable (Zählvariable), die zur Steuerung der Schleife dient, z.B. `i=0`

`BEDINGUNG` Die `for`-Schleife wird ausgeführt, solange die Auswertung der Bedingung `true` ergibt. Beispiel: `i<= 100`

`VERÄNDERUNG` Eine Anweisung, mit der die Variable für jeden Durchlauf verändert wird (wobei man darauf achten sollte, dass auf diese Weise irgendwann die Abbruchbedingung erfüllt wird ;-). Beispiel: `i++`

objectloop

definiert eine Schleife über alle im Spiel definierten Objekte oder über eine Auswahl dieser Objekte:

```
objectloop (SELEKTIONSAUSDRUCK) ANWEISUNGSBLOCK
```

Besteht der SELEKTIONSAUSDRUCK einfach nur aus dem *Namen einer Variable*, so wird der Anweisungsblock für *jedes Objekt im Spiel* einmal ausgeführt, wobei im Anweisungsblock mit eben dieser Variable auf das aktuelle Objekt zugegriffen werden kann.

Der Selektionsausdruck kann auch bestehen aus einer *Bedingung mit einem Variablennamen auf der linken Seite*. Dann wird der Anweisungsblock für *jedes Objekt, das diese Bedingung erfüllt,* einmal ausgeführt; wiederum referenziert der Variablenname im Anweisungsblock das gerade aktuelle Objekt.

Beachte: Eine Konstruktion, die auf einer Bedingung mit `in` aufbaut und die so gefundenen Objekte bewegt, sägt den Ast ab, auf dem sie sitzt. Statt:

```
objectloop (x in player) remove x; ! SO NICHT!
```

schreibt man also:

```
while (child(player)) remove child(player);
```

3.1.4 Anweisungen, Routinen und Funktionen

Anweisungen

Zuweisung mit `=` Inform verwendet das einfache Gleichheitszeichen für Zuweisungen, das doppelte `==` ist ein Vergleichsoperator. Der Compiler warnt bei möglichen Verwechslungen, aber man sollte sich nicht darauf verlassen! Mehrfache Zuweisungen sind möglich:

```
VARIABLE1 = VARIABLE2 = AUSDRUCK;
```

Anweisungsfolgen Jede Anweisung muss mit einem Semikolon `;` abgeschlossen werden. Ausnahme: Die Definition einer *Property* eines Objekts wird mit einem Komma abgeschlossen; das letzte Semikolon vor diesem Komma entfällt. Umgekehrt wird die gesamte Objektdefinition mit einem Semikolon beendet, vor diesem abschließenden Semikolon braucht es kein Komma.

Das Komma `,` ist eine Möglichkeit, eine Folge aus mehreren Anweisungen zu schreiben: `ANWEISUNG1, ANWEISUNG2;` gilt syntaktisch als nur eine Anweisung.

Anweisungsblock Ein Anweisungsblock besteht entweder aus einer einzelnen Anweisung oder aus mehreren aufeinanderfolgenden Anweisungen, die in geschweifte Klammern `{}` eingeschlossen sind.

Rückgabewert Eine Anweisung hat einen Rückgabewert; bei Zuweisungen ist das das Ergebnis.

Der Rückgabewert von Routinen kann mit der Anweisung `return AUSDRUCK;` explizit angegeben werden. *Beachte:* Bei Ausführung von `return` erfolgt auch der Rücksprung aus der Routine.

`return;` ohne Zusatz bedeutet immer `return true;` und sollte der besseren Lesbarkeit wegen so oder abgekürzt als `rtrue;` geschrieben werden. Das Gegenstück dazu ist der Befehl `return false;` abgekürzt `rfalse;`

Wird der Rückgabewert nicht explizit gesetzt, haben selbständig definierte Routinen immer `true;` unselbständige (als *property* eines Objekts definierte) Routinen immer `false` als Rückgabewert.

Routinen (Unterprogramme)

sind Anweisungsfolgen mit bis zu 15 (oder ohne) lokalen Variablen und einem Namen. Sie werden in eckige Klammern `[ ]` eingeschlossen. Im einzelnen besteht eine Routinendefinition aus folgenden Elementen:

```
[ NAME VARIABLENLISTE; ANWEISUNGEN ]
```

`[` einer eckigen Klammer zu Beginn

`NAME` Nur bei selbständig, d.h. außerhalb eines Objekts definierten Routinen ist der erste nach der Klammer folgende Bezeichner der Name der Routine (unter dem sie auch wieder aufgerufen werden kann, s.u.) . Besondere Regeln gelten für Aktionsroutinen: Für jede (vom Benutzer neu definierte) Aktion, z.B. `##MachIrgendwas` muss eine (selbständig definierte) Routine mit dem Namen der Aktion und einem angehängten `Sub` (hier also: `MachIrgendwasSub`) definiert sein.

Ist eine Routine unselbständig, d.h. als *property* eines Objektes definiert, ist kein Name anzugeben, die Routine erhält automatisch den Namen der *property* und ist mit der Nachricht `OBJEKT.PROPERTY` aufrufbar.

`VARIABLENLISTE` optional, Bezeichner für bis zu 15 lokale Variable, die entweder mit 0 vorbelegt sind oder beim Aufruf als Parameter übergeben werden.

`;` ein Semikolon schließt den Kopf der Routine ab (auch wenn weder ein Name noch lokale Variable angegeben sind, das Semikolon muss geschrieben werden!) und es folgen die

ANWEISUNGEN in beliebiger Anzahl und Länge, gefolgt von einer abschließenden

] eckigen Klammer am Ende. (Semikolon oder Komma danach nicht vergessen.)

Wird der Rückgabewert nicht explizit gesetzt, haben selbständig definierte Routinen immer `true`; unselbständige (als *property* eines Objekts definierte) Routinen immer `false` als Rückgabewert. (Man kann das gar nicht oft genug sagen!)

Aufruf von Routinen und Funktionen

Routinen werden aufgerufen mit ihrem Namen gefolgt von einer Parameterliste in runden Klammern. Die Parameterliste kann leer sein oder bis zu sieben durch Kommata zu trennende Argumente enthalten. Die Argumente werden der Reihenfolge nach auf die lokalen Variablen der Routinendefinition abgebildet. Rekursion ist zulässig. Der Aufruf kann als Funktion innerhalb einer Zuweisung erfolgen, muss dies aber nicht. Beispiele:

```
x = ackermann (y, z); ! (Vom Gebrauch wird abgeraten!)[5]
streik.posten();
```

Dispatcherroutinen

Ein besonderes Format gilt für die Routinen in den *properties* `after, before, life, orders, react_after` und `react_before`. Diese Routinen legen fest, wie ein Objekt sich zu verschiedenen Handlungen (Actions) des Spielers verhält, die mit ihm oder in seiner Nähe stattfinden.

Sie werden gewissermaßen so geschrieben, als wäre ein `switch(Action)` vorangestellt:

```
[OPTIONALE LISTE LOKALER VARIABLER;

    AKTION1: ANWEISUNGEN ... ;
    AKTION2, AKTION3: ANWEISUNGEN ...;
    ...
    default: ANWEISUNGEN ... ;
],
```

Auch hier müssen die Anweisungen in den einzelnen Zweigen nicht 'geblockt' werden. Die Aktionen werden ohne einleitendes `##` geschrieben.

[5] Die Ackermann-Funktion (in Inform nicht vordefiniert!) ist ein Beispiel für eine zweifach rekursive Funktion. Ihr bekanntester „Seiteneffekt" ist der enorme Verbrauch an Systemressourcen für vergleichsweise kleine Werte von x und y.

3.1.5 Der richtige Ausdruck

`print` und `print_ret`

Der grundlegende Ausgabebefehl ist `print` gefolgt von einer durch Kommata getrennten Liste aus Zeichenketten und/oder Objekten oder Ausdrücken mit vorangestellter Regel. Die *Regel* – in runden Klammern vor das Objekt oder den Ausdruck gestellt – legt fest, ob z.B. die Adresse eines ihr folgenden Objekts oder aber sein Name im Nominativ Singular mit vorangestelltem Artikel ausgegeben werden soll:

```
print (GDer) Objektname, " hat als Objekt intern die Nummer ", Objektname;
```

Für die häufig benötigte Folge `print ... ; new_line; return true;` gibt es die abgekürzte Schreibweise `print_ret ... ;` oder – noch kürzer und wenn die zu druckenden Daten mit einem String anfangen – `"...";` d.h. einfach der zu Beginn stehende String, optional wie `print` gefolgt von der Liste der weiter zu druckenden Daten. Die meisten auf den ersten Blick unmotiviert herumstehenden Zeichenketten eines Inform-Programms sind bei näherem Hinsehen solcherart abgekürzte `print_ret`-Befehle.

Im auszugebenden String haben zwei Zeichen besondere Bedeutung: das Caret `^` wird als Zeilenumbruch und die Tilde `~` als Anführungszeichen ausgegeben.

(allgemeine) Regeln für den Ausdruck

`(number)` AUSDRUCK Das Ergebnis des Ausdrucks als ausgeschriebene Zahl

`(char)` AUSDRUCK Das Ergebnis des Ausdrucks als Zeichen

`(string)` ADRESSE Der String an dieser Adresse

`(address)` ADRESSE Der Wörterbucheintrag mit dieser Adresse

`(name)` OBJEKT Der `short_name` des Objekts

`(`ROUTINE**`)`** AUSDRUCK Das Ergebnis der mit *Ausdruck* als Parameter aufgerufenen *Routine*. Nicht verwechseln mit ROUTINE`(`AUSDRUCK`)`, das zwar den selben Text, aber obendrein noch den Rückgabewert der Routine (meist 1, entsprechend `true`) ausgibt.

Die deutsche Library enthält eine ganze Reihe solcher Regeln für die grammatisch wohlgeformte Ausgabe der Namen von Objekten. Dazu später 3.2.5 auf Seite 89.

sonstige Ausgabebefehle

new_line; schaltet eine neue Zeile

spaces AUSDRUCK**;** gibt soviele Leerzeichen aus, wie der Ausdruck angibt

box gefolgt von einem oder mehreren Strings erzeugt eine gesonderte Ausgabe dieser Strings (zentriert in einem invertierten Fenster über dem normalen Text)

font off; schaltet vom normalen Ausgabefont auf einen nicht proportionalen Font um

font on; schaltet die (normale) Proportionalschrift wieder ein (falls vom Interpreter unterstützt!)

style bold; schaltet auf Fettschrift

style underline; schaltet Unterstreichung ein

style reverse; schaltet invertiertes Video ein

style roman; ist normale Schrift ohne Auszeichnung.

Beachte: Alle style-Befehle werden von Glulx nicht unterstützt.

3.1.6 Miscellen

string N **"**EIN_HÄUFIG_GEBRAUCHTER_STRING**"** Mit Werten von 0 bis 31 für N kann so eines von 32 Stringmakros belegt werden. (Glulx: 64)

@00 ... @31 gibt eine solches Makro innerhalb eines Strings wieder aus.

inversion; gibt die Versionsnummer des Compilers aus.

Release AUSDRUCK**;** definiert die 'laufende Nummer' des Spiels (üblicherweise 1 für die erste veröffentlichte Fassung etc.)

Serial "DATUM**";** definiert die Seriennummer; *Datum* ist eine sechsstellige Zahl, default der Tag der letzten Änderung der Quelldatei.

Statusline score; (Voreinstellung) und

Statusline time; ändern das Aussehen der Statuszeile.

quit; Beendet das Programm. Sofort.

.MARKE**;** Ein Bezeichner mit vorangestelltem Punkt markiert eine Stelle des Programms als Ziel für *(geflüstert)* duweißtschonwas. Die Marke ist lokal zu der Routine, in der sie steht.

jump MARKE; *(noch leiser geflüstert:)* duweißtschonwas . . .[6]

save MARKE; Speichert den kompletten Programmstatus und springt dann zur Marke, wenn

restore MARKE; erfolgreich ausgeführt wird. Im Normalfall braucht man das nicht, weil das Abspeichern und Laden von Spielständen von der Library erledigt wird, der Spieler muss nur die Aktionen ##Save und ##Restore auslösen.

read TEXTPUFFER N; und

read TEXTPUFFER PARSERPUFFER; lesen die Eingabe zeichenweise in den TEXTPUFFER, einen *byte array* der Länge N (TEXTPUFFER->0 enthält N; TEXTPUFFER->1 die Zahl der eingelesenen Zeichen, die ab TEXTPUFFER->2 enthalten sind); die zweite Variante trennt den eingelesenen Text bereits in Wörterbucheinträge; PARSERPUFFER->1 enthält die Zahl der gelesenen Worte, deren Wörterbucheinträge (oder 0, wenn das Wort nicht bekannt war) stehen in PARSERPUFFER-->3, PARSERPUFFER-->7, ... , PARSERPUFFER-->2*i-1 ...

3.1.7 Compileranweisungen

message "TEXT_DER_NACHRICHT" gibt während des Übersetzens eine Nachricht aus

System_file; gibt für diese Datei keine Compilerwarnungen aus und ermöglicht das „Ausblenden" einzelner Unterprogrammdefinitionen einer Bibliotheksdatei mit dem Befehl:

Replace NAME_EINER_BIBLIOTHEKSROUTINE; Diese Direktive muss vor dem Einbinden der Bibliothek angegeben werden und bewirkt, dass eben diese Routine der Bibliothek nicht mit eingebunden wird, so dass sie nachher noch mit eigenem Code definiert werden kann.

Ifdef BEZEICHNER; **Ifndef** BEZEICHNER; **Iftrue** AUSDRUCK; **Iffalse** AUSDRUCK; beginnen einen Abschnitt, der nur dann kompiliert werden soll, wenn die Bedingung zutrifft;

Ifnot; (optional) beendet diesen Abschnitt und beginnt einen alternativen Abschnitt, der nur dann kompiliert wird, wenn die vorherige Bedingung *nicht* zutrifft;

Endif; beendet die bedingte Kompilierung.

[6] Hier geht es um die *Dunkle Seite* der Programmierkunst: Die Marke ist ein Sprungziel, zu dem mit dem Befehl jump gesprungen werden kann. Die Verwendung von Sprungbefehlen führt in etwa 1% der Fälle zu eleganten Lösungen und in den anderen 99% zu Komplikationen in der Form von schwer lesbarem Code bis hin zu völliger Verwirrung. Wenn du also nicht wußtest, was das Flüstern sollte, bist du jetzt gewarnt.

Default BEZEICHNER AUSDRUCK**;** definiert eine Konstante mit dem Namen Bezeichner (und optional mit dem Wert des Ausdrucks), *aber nur dann*, wenn es sie nicht schon gibt.

Stub BEZEICHNER ANZAHL**;** definiert in ähnlicher Weise einen Unterprogrammkopf mit der in Anzahl angegebenen Zahl lokaler Variabler (maximal 3), falls es eine Routine mit diesem Namen noch nicht gibt.

Include "QUELLDATEI**";** bindet eine Datei aus dem Bibliotheksverzeichnis ein; die Endungen .h und .inf können weggelassen werden.

Include ">QUELLDATEI**";** bindet eine Datei aus dem aktuellen Verzeichnis ein.

Link "MODULDATEI**";** bindet ein vorkompiliertes Modul ein (das hat schon lange keiner mehr gemacht; es diente ursprünglich zur Zeitersparnis bei der Kompilierung langer Spiele, aber heute sind die Computer so schnell, dass es den Aufwand nicht lohnt).

Abbreviate "STRING1**" "**STRING2**" ...;** gibt bis zu 64 Strings an, die durch Abkürzungen ersetzt werden sollen. Benötigt Compilerschalter `-e`. Die besten Abkürzungen lassen sich mit `-u` ermitteln.

ICL-Befehle

Der Compiler wird normalerweise nach folgendem Muster aufgerufen:

```
inform ICL_BEFEHLE EINGABEDATEI AUSGABEDATEI
```

Die Angabe einer Ausgabedatei ist optional. ICL steht für Inform Command Language und umfasst folgende Möglichkeiten:

-SWITCHES Eine mit einem einfachen Strich eingeleitete Folge von Compilerschaltern (siehe nächsten Abschnitt)

(DATEINAME**)** In runden Klammern der Name einer Datei, die ihrerseits ICL-Kommandos (eines pro Zeile) enthält

compile EINGABEDATEI AUSGABEDATEI Die Anweisung, eine weitere Datei zu kompilieren (Ausgabedatei optional).

+include_path=VERZEICHNIS**,**VERZEICHNIS**...** Die Anweisung, Bibliotheksdateien aus einem bestimmten Verzeichnis zu laden

+module_path=VERZEICHNIS**,**VERZEICHNIS**...** Die Anweisung, vorkompilierte Module aus einem bestimmten Verzeichnis zu laden.

In gleicher Weise können der `source_path` für Quelldateien, der `code_path` (für das Kompilat) und auch der `icl_path` für weitere Kommandodateien angegeben werden.

`+language_name=german` damit die deutschen Varianten der Bibliotheksdateien geladen werden

In gleicher Weise können der `transcript_name` für die mit -t erzeugte Textdatei und der `debugging_name` für die mit -k zu schreibende Debuggerinformation verändert werden.

`$large`, `$huge` und `$small` sind drei verschiedene Speichermodelle je nach der Größe des zu erstellenden Spiels; `$large` ist voreingestellt. Angeblich soll mit `$list`mehr über die Details des jeweiligen Speichermodells zu erfahren sein, aber meine Version des Compilers reagiert nicht darauf. So kann ich auch nicht mit `$NAME=WERT` einzelne dieser Einstellungen ändern.

Compilerschalter

Es gibt zwei Arten von Compilerschaltern: solche, die eingeschaltet (`-a`) oder mit der Tilde ausgeschaltet (`-~a`) werden (markiert mit ein/aus) und solche, an die eine Zahl angehängt werden muss (markiert mit *). Die Schalter werden ohne Punkt und Komma einfach hintereinandergesetzt, z.B. bedeutet `-a~bcd2`, dass a und c eingeschaltet, b ausgeschaltet und d auf 2 gesetzt wird. Innerhalb der zu übersetzenden Datei können ebenfalls Compilerschalter mit dem Befehl `switches LISTE_VERSCHIEDENER_SCHALTER;` gesetzt werden.

-d* (Für englische Schreibmaschinisten) automatisches Eliminieren doppelter Leerzeichen: 0 - niemals, 1 - nach Punkten, 2 - nach Punkten, Frage- und Ausrufungszeichen

-e ein/aus Deklarierte Abkürzungen (abbreviations) benutzen

-g* Trace für Unterprogrammaufrufe: 0 - keine, 1 - außerhalb von System files, 2 - alle

-i ein/aus In der Datei gesetzte Switches ignorieren

-k ein/aus Ausgabe der Informationen für das Debuggen mit Infix; schaltet auch -D automatisch ein.

-r ein/aus Der gesamte Text des Spiels wird in eine Textdatei geschrieben

-v* 3 bis 8 um Spieldateien im Format .z3 bis .z8 zu erzeugen (default 5)

-C* 0 bis 9 für den Zeichensatz der Quelldatei; 0 bedeutet ASCII, 1 bis 9 bedeuten ISO 8859-1 bis -9 (default 1)

-D ein/aus die Debugging-Verben werden mit übersetzt

-F* 0 or 1 temporäre Dateien werden angelegt, um Speicherplatz zu sparen

-G ein/aus (nur beim Biplattform-Compiler): übersetzt nicht ins Z-Code-Format, sondern ins Glulx-Format. Das wird hier nicht weiter beschrieben.

-M ein/aus als Modul kompilierten

-S ein/aus Strenge Fehlerprüfung zur Laufzeit (schließt automatisch -D mit ein)

-U ein/aus vorkompilierte Bibliotheksmodule einbinden

-X ein/aus den Infix Debugger einbinden

-a ein/aus Trace der Assemblerbefehle (ohne Hexdump)

-c ein/aus ausführlichere Fehlermeldungen

-f ein/aus Ermittelt die tatsächliche Einsparung bei der Verwendung von Abkürzungen

-h* ein/1/2 Hilfe zur Compilerbenutzung

-j ein/aus Konstruierte Objekte auflisten

-l ein/aus Alle übersetzten Anweisungen auflisten

-m ein/aus Speicherbenutzung angeben

-n ein/aus Nummern der *properties*, attributes und actions angeben

-o ein/aus Offsetadressen ausgeben

-p ein/aus Compilerfortschritt prozentual anzeigen

-q ein/aus Obsolete Befehle nicht monieren

-s ein/aus Statistik ausgeben

-t ein/aus Trace der Assemblerbefehle (mit Hexdump)

-u ein/aus mögliche Abkürzungen ermitteln (kann länger dauern)

-w ein/aus Warnungen aus

-x ein/aus Je ein Gatter # für 100 übersetzte Zeilen ausgeben

-y ein/aus Trace der Linkerinformationen

-z ein/aus Speicheraufteilung der Z-Maschine angeben

-E* Fehlermeldungen im Stil von Acorn (0), Microsoft (1) oder Mac (2)

3.2 Die deutsche Library

In weiten Teilen – Arbeitsweise, Funktionen, Attribute, *Properties*, Aktionen – sind die deutsche und die englische Library übereinstimmend. Ich beschränke mich nicht auf die deutschen Eigenheiten, sondern versuche, die Library, so wie man sie für das deutschsprachige Programmieren braucht, als vollständiges Ganzes darzustellen. Rein englischsprachige Eigenheiten können mir dabei schon mal unter den Tisch gefallen sein.

3.2.1 Konstante

AMUSING_PROVIDED ist normalerweise nicht definiert. Wird sie definiert, so wird der Spieler am Ende eines gewonnenen Spiels gefragt, ob er noch unterhaltsame Dinge wissen möchte; falls ja, wird die Routine `Amusing()` aufgerufen.

DEATH_MENTION_UNDO ist normalerweise nicht definiert. Wird sie definiert, so wird Undo nach Spielende bei den möglichen Optionen (neben Restore, Restart, Quit und evtl. Amusing) mit aufgeführt.

DEBUG ist normalerweise nicht definiert. Wird es definiert, so werden die Debuggerkommandos aktiviert.

HEADLINE = "TEXT"; *muss* definiert werden, gibt den Untertitel am Spielanfang aus.

MANUAL_PRONOUNS ist normalerweise nicht definiert. Wird es definiert, so werden Pronomina nicht mehr „automatisch” gesetzt.

MAX_CARRIED = AUSDRUCK; Voreinstellung 100. Legt fest, wieviele Objekte die Spielerfigur auf einmal tragen kann.

MAX_SCORE = AUSDRUCK; Voreinstellung 0. Die maximal erreichbare Punktezahl.

MAX_TIMERS = AUSDRUCK; Voreinstellung 32. Legt die Höchstanzahl gleichzeitig aktiver *timer* und *daemons* fest.

NO_PLACES ist normalerweise nicht definiert. Wird es definiert, so werden die Aktionen `##Objects` und `##Places` nicht eingebunden.

NUMBER_TASKS = AUSDRUCK; Voreinstellung 1. Die Anzahl der zu erfüllenden Aufgaben.

OBJECT_SCORE = AUSDRUCK; Voreinstellung 4. Punktezahl, die der Spieler für das erste Aufnehmen eines Objekts mit dem Attribut `scored` erhält.

R_NEU ss und ß werden nach den neuen Rechtschreibregeln verwendet.

ROOM_SCORE = AUSDRUCK; Voreinstellung 5. Punktezahl, die der Spieler für das erste Betreten eines Raums mit dem Attribut `scored` erhält.

SACK_OBJECT = OBJEKT; Ein Objekt – mit dem Attribut `container` – in dem das Spiel automatisch Gegenstände verstaut, die der Spieler nicht mehr halten kann.

Story = "TEXT"; *muss* definiert werden, gibt den Titel am Spielanfang aus.

TARGET_GLULX oder

TARGET_ZCODE werden vom Compiler gesetzt, je nachdem, ob das Spiel in Z-Code oder Glulx übersetzt wird (Compilerschalter `-G`).

TASKS_PROVIDED ist normalerweise nicht definiert. Wird es definiert, so wird das *Scoring* von Objekten/Räumen auf erfüllte Aufgaben erweitert.

USE_MODULES ist normalerweise nicht definiert. Wird es definiert, so können vorkompilierte Bibliotheksdateien verwendet werden.

WITHOUT_DIRECTIONS ist normalerweise nicht definiert. Wird es definiert, so werden die normalen Richtungen des `Compass` außer *in* und *out* deaktiviert; die Autorin muss eigene Richtungen festlegen.

WORDLENGTH ist 2, wenn für Z-Code, und 4, wenn für Glulx übersetzt wird. Aus Kompatibilitätsgründen sollte man diese Konstante verwenden, wenn man Speicherarithmetik betreibt.

3.2.2 Variable

action Die gerade ausgeführte Aktion

actor Die Figur, die die Aktion ausführt, das kann nach einer *order* auch ein NPC sein.

compass ist ein `container`, der die verfügbaren Richtungen enthält, also im Normalfall `d_obj, e_obj, in_obj, n_obj, ne_obj, nw_obj, out_obj, s_obj, se_obj, sw_obj, u_obj, w_obj`.

deadflag ist im Spiel immer 0; wird es auf 1 gesetzt, „stirbt" die Spielerfigur, bei 2 ist das Spiel als gewonnen beendet, alle Werte von 3 aufwärts sind frei für alternative Enden verwendbar.

inventory_stage steuert den Ablauf von Routinen der *properties* `invent` und `list_together`

keep_silent ist voreingestellt auf `false`, wenn es auf `true` gesetzt wird, geben die meisten Aktionen der Klasse 2 im Erfolgsfall keine Rückmeldung

LibraryMessages ist ein im Bedarfsfall von der Autorin – vor Einbinden von `VerbLib` – zu definierendes Objekt, das eine `before`-Routine enthält, die für die in ihr abgefangenen Aktionen die Standardantworten der Library überschreibt.

location Der Raum, in dem die Spielerfigur ist, wenn sie Licht hat. Hat es kein Licht, ist `location==thedark` und der Raum ist nur in der `real_location` zu finden.

notify_mode ist voreingestellt auf `true`, wird es auf `false` gesetzt, erfolgt keine Meldung bei einer Veränderung des Punktestands

noun Das direkte Objekt der aktuellen Aktion.

player Die Spielerfigur.

real_location Der Raum, in dem sich die Spielerfigur tatsächlich befindet, wenn es dunkel und `location` auf `thedark` gesetzt ist.

score Der aktuelle Punktestand.

second Das zweite, meist indirekte Objekt der aktuellen Aktion.

selfobj die vordefinierte Spielerfigur. Der Zugriff sollte aber immer indirekt auf die Variable `player` erfolgen.

self Das Objekt, dass die gerade verarbeitete Nachricht (*message*) empfangen hat

sender das Objekt, das die Nachricht gesendet hat; oft `InformLibrary`, kann auch `nothing` sein

task_scores ist ein *byte array*, der der Reihe nach die Punktzahlen für das aufgabenbasierte *scoring* enthält

thedark ein obskurer Raum, der zur `location` (aber nicht zum *parent* der Spielerfigur) wird, wenn es dunkel ist.

the_time Die Uhrzeit im Spiel, gemessen in Minuten nach Mitternacht

turns Zahl der schon gespielten Züge

wn („word number") zeigt an, im wievielten Wort des Eingabezeile der Parser steht (erstes Wort==1). Die Routine `NextWord()` holt dieses Wort und addiert 1 zu wn.

3.2.3 Funktionen

Achieved(AUSDRUCK**);** Eine der punktbringenden Aufgaben ist erledigt.

AddToScope(GERUFENES_OBJEKT**);** Wird in der `add_to_scope`-*property* eines Objekts aufgerufen. Das solcherart gerufene Objekt kommt dann zusammen mit dem rufenden Objekt in und aus dem Blickfeld *(scope)*.

AfterRoutines() ruft innerhalb einer Aktionsroutine die `after`-routinen aller Objekte auf, die sich mit der Aktion im after-Stadium befassen wollen. Siehe auch 3.2.8 auf Seite 100.

AllowPushDir(GESCHOBENES_OBJEKT**)** ist in einer before-Routine im Zweig für ##PushDir zu verwenden, um das Schieben des Objekts in einen anderen Raum zu ermöglichen, siehe 3.2.7 auf Seite 95.

Banner() gibt das „Banner", also die Titelinformationen des Spiels, aus.

CDefArt(OBJEKT**)** gleichbedeutend mit print (GDer) OBJEKT

ChangeDefault(PROPERTY,AUSDRUCK**);** Ändert die Vorbelegung eines *common property*.

ChangePlayer(OBJEKT,FLAG**)** Wechselt mit der Spielerfigur in das neue Objekt. Wenn das Flag auf true gesetzt ist, kommt bei der Raumbeschreibung der Zusatz „als OBJEKT" um zu kennzeichnen, welche Person der Spieler gerade steuert.

DefArt(OBJEKT**)** gleichbedeutend mit print (der) OBJEKT

DictionaryLookup(BYTE_ARRAY,LÄNGE**)** Ein im BYTE_ARRAY->0 bis ->LÄNGE-1 abgelegtes Wort wird im Wörterbuch gesucht. Rückgabewert ist der Zeiger auf den Eintrag oder false.

DoMenu("TEXT**",**ROUTINE1,ROUTINE2**)** Ruft ein Menüsystem auf den Schirm. Überholt durch menu.h.

DrawStatusLine() Gibt die Statuszeile aus

EnglishNumber(AUSDRUCK**)** Gibt die Zahl in – deutschen – Worten aus.

GetGNAOfObject(OBJEKT**)** Ermittelt den gender-number-animation - Wert des short name eines OBJEKTS..

HasLightSource(OBJEKT**)** Prädikat. Testet, ob das OBJEKT light hat und gibt true zurück, wenn ja.

InDefArt(OBJEKT**)** gleichbedeutend mit print (ein) OBJEKT

IndirectlyContains(OBJEKT1,OBJEKT2**)** Prädikat, das testet, ob OBJEKT2 im object tree irgendwo unterhalb von OBJEKT 1 steht und true zurückmeldet, falls dies der Fall ist.

IsSeeThrough(OBJEKT**)** Prädikat, das true zurückgibt, wenn das OBJEKT Licht durchlässt; also entweder transparent, supporter, oder enterable hat, solange es sich nicht um einen closed container handelt.

Locale(OBJEKT,"ÜBERSCHRIFT","ZWISCHENÜBERSCHRIFT"**)** Gibt die volle Beschreibung eines OBJEKTS aus, das dabei behandelt wird, als ob es ein Raum wäre, d.h. zunächst wird die Beschreibung ausgegeben; ist diese Beschreibung leer, folgt die ÜBERSCHRIFT, wenn nicht, die ZWISCHENÜBERSCHRIFT. Danach folgen die in OBJEKT enthaltenen Objekte, deren Anzahl auch der Rückgabewert der Funktion ist.

LoopOverScope(ROUTINE**,**SPIELFIGUR**)** Führt für jedes Objekt im Blickfeld (scope) der SPIELFIGUR die ROUTINE einmal aus (mit dem jeweiligen Objekt als Parameter). Ist keine SPIELFIGUR angegeben, wird automatisch die Spielerfigur (player) angenommen.

LTI_Insert(POSITION**,**CHARACTER**)** Schiebt den CHARACTER an POSITION in den Standardeingabepuffer (library text array)

MoveFloatingObjects() Aktualisiert alle Schiebekulissen („Floating objects") und sollte nach jeder Änderung eines Kulissenobjekts aufgerufen werden.

NextWord() Holt das nächste Wort aus der Eingabezeile und inkrementiert wn. Rückgabewert ist das Wort oder false, wenn das Wort nicht im Wörterbuch eingetragen ist oder die Eingabe erschöpft ist.

NextWordStopped() Holt das nächste Wort aus der Eingabezeile und inkrementiert wn. Rückgabewert ist das Wort oder false, wenn das Wort nicht im Wörterbuch eingetragen ist oder -1, wenn die Eingabe erschöpft ist.

NounDomain(OBJEKT1**,**OBJEKT2**,**TYP**)** ist eine grundlegende Routine des Parsers. Ihr werden zwei OBJEKTE übergeben, in deren Bereich gesucht werden soll (üblicherweise actor und location); außerdem als TYP eine der Konstanten NOUN_TOKEN, HELD_TOKEN oder CREATURE_TOKEN. Die Routine sucht dann, ob ab der aktuellen Position (wn) ein passendes Objekt zu finden ist und liefert dieses Objekt zurück; nothing, wenn nichts gefunden wird und REPARSE_CODE, wenn der Parser beim Spieler „nachfragen" musste.

ObjectIsUntouchable(OBJEKT**,**FLAG**)** Prädikat, das true zurückgibt, wenn zwischen der Spielerfigur und dem Objekt sich eine Barriere – ein geschlossener container – befindet. Seiteneffekt ist eine Nachricht an den Spieler. Das FLAG kann weggelassen werden, wird es auf true gesetzt, wird die Benachrichtigung unterdrückt.

OffersLight(OBJEKT**)** Prädikat, das true zurückliefert, wenn das OBJEKT (Raum oder container) Licht enthält.

ParseToken(TOKENTYPE**,**TOKENDATA**)** Das ist die „general parsing routine" der Library, die die Eingabe auf ein erwartetes token überprüft. Für den Anwender zugänglich sind nur zwei verschiedene Tokentypen, nämlich ELEMENTARY_TT mit den möglichen Einträgen für Tokendata: NOUN_TOKEN, HELD_TOKEN, MULTI_TOKEN, MULTIHELD_TOKEN, MULTIEXCEPT_TOKEN, MULTIINSIDE_TOKEN, CREATURE_TOKEN und NUMBER_TOKEN. Außerdem kann der Tokentyp sein: SCOPE_TT; wobei Tokendata eine „scope routine" sein muss. Rückgabewerte sind GPR_FAIL (nichts gefunden); GPR_PREPOSITION (Übereinstimmung, aber keine Daten); GPR_NUMBER eine gefundene Zahl, GPR_MULTIPLE bei einer gefundenen Übereinstimmung mit mehreren Objekten und GPR_REPARSE wenn mit dem Parsen von vorn angefangen werden muss – oder aber das gefundene Objekt.

PlaceInScope(OBJEKT) Das OBJEKT wird ins Blickfeld (*scope*) gerückt. Normalerweise verwendet in Routinen für add_to_scope oder dem Grammatiktoken scope=ROUTINE.

PlayerTo(OBJEKT,FLAG) versetzt bzw. teleportiert den Spieler in das OBJEKT (einen Raum oder ein enterable). Ist das FLAG false, wird als erstes ein ##Look im neuen Raum ausgelöst; bei true wird nichts ausgegeben und bei 2 wird eine abgekürzte Raumbeschreibung ausgegeben, wenn der Raum visited hat.

PronounNotice(OBJEKT) Setzt alle grammatisch zulässigen Pronomen auf das OBJEKT.

PronounValue(PRONOMEN) Ermittelt die aktuelle Belegung eines (als Wörterbucheintrag anzugebenden) PRONOMENS, nothing, wenn das Pronomen nicht besetzt ist.

ScopeWithin(OBJEKT) Sinnvollerweise nur in *„scope Routinen"* benutzbar und auch dort nur, wenn die Libraryvariable scope_stage auf 2 gesetzt ist. ScopeWithin() rückt den Inhalt von Objekt ins Blickfeld; d.h. zunächst die Child-Objekte und dann rekursiv deren Inhalt, wenn sie durchsichtig sind.

SetPronoun(PRONOMEN,OBJEKT) Bewirkt, dass das OBJEKT vom Spieler mit dem PRONOMEN angesprochen werden kann.

SetTime(TIME,RATE) Für Spiele, in denen die Zeit gemessen werden soll (nicht die Echtzeit, sondern eine fiktive Zeit, die nur für das Spiel gilt): Der Ausdruck für TIME setzt die Uhr auf eine Anzahl von Minuten nach Mitternacht; RATE gibt an, wieviel Minuten bei jedem Zug verstreichen; wenn ein negativer Wert angegeben wird, wieviel Züge auf eine Minute kommen. 0 bedeutet, dass keine Zeit verstreicht.

StartDaemon(OBJEKT) Aktiviert den Dämon eines OBJEKTS

StartTimer(OBJEKT,AUSDRUCK) Startet den Timer eines OBJEKTS und setzt den Anfangswert auf AUSDRUCK; der Timer wird in jedem Zug um 1 heruntergezählt.

StopDaemon(OBJEKT) Hält den Dämon des OBJEKTS an.

StopTimer(OBJEKT) Hält den Timer des OBJEKTS an

TestScope(OBJEKT,ACTOR) Prädikat, das testet, ob ein OBJEKT im Blickfeld (scope) eines bestimmten ACTOR sich befindet; wird kein anderer ACTOR angegeben, wird der Test für die Spielerfigur ausgeführt.

TryNumber(WORDNUM) versucht, das Wort an der angegebenen Stelle (WORDNUM ist ein Wert von wn) als Zahl zu parsen, gibt -1000 zurück, wenn es schiefgeht, ansonsten die gefundene Zahl. Zahlen über 10000 werden auf 10000 gekappt.

UnsignedCompare(A,B) Zum Vergleich zweier Speicheradressen im Bereich 0..65535, denn > vergleicht nur mit Vorzeichen. Resultat 1 wenn A > B, 0 wenn A==B und -1 (bzw. 65535) wenn A<B.

WordAddress(WORDNUM) WORDNUM ist eine Wortposition der Eingabezeile (vgl. wn) – Rückgabewert ist der *byte array*, in dem der Text des Wortes zeichenweise abgelegt ist.

WordInProperty(WORT,OBJEKT,PROPERTY) Prädikat, das testet, ob das WORT in einem PROPERTY des OBJEKT enthalten ist; das PROPERTY muss einen Array von Wörterbucheinträgen enthalten; üblicherweise wird name verwendet.

WordLength(WORDNUM) WORDNUM ist eine Wortposition der Eingabezeile (vgl. wn) – Rückgabewert ist die Länge des Worts an dieser Position.

WriteListFrom(OBJEKT,STIL) Gibt eine Liste des OBJEKTS und aller folgenden Objekte der gleichen Ebene (der *siblings*) aus. Alle Objekte innerhalb eines Objekts X werden gelistet mit WriteListFrom(child(X), STIL). Der STIL wird durch Summieren aus folgenden Konstanten gebildet:

NEWLINE_BIT Zeilenschaltung nach jedem Eintrag

INDENT_BIT Einträge einrücken

FULLINV_BIT Vollständiges Inventar

ENGLISH_BIT Ganze Sätze bilden

RECURSE_BIT Rekursiv nach üblichen Regeln arbeiten

ALWAYS_BIT Immer rekursiv abarbeiten

TERSE_BIT Knapper Stil

PARTINV_BIT Nur kurzes Inventar

DEFART_BIT Bestimmten Artikel verwenden

WORKFLAG_BIT Nur Objekte mit workflag werden gelistet

ISARE_BIT Einleitendes „ist” bzw. „sind” ausgeben

CONCEAL_BIT Objekte mit concealed oder scenery werden weggelassen

YesOrNo() Prädikat, das true rückmeldet, wenn der Spieler eine Ja/Nein-Frage mit Ja beantwortet hat. Die Frage selbst wird von YesOrNo() nicht behandelt, sie muss vor Aufruf der Routine gestellt worden sein.

3.2.4 Einhänger (entry point routines)

Diese Routinen haben zwar feste Namen, ihr Inhalt kann aber vom Benutzerprogramm frei definiert werden. Sie werden im Regelfall jedoch nicht vom Benutzerprogramm, sondern aus der Library heraus aufgerufen. Dort sind sie an festgelegten Stellen im Spielablauf „eingehängt”; wenn man an diesen Stellen etwas bestimmtes tun will, schreibt man es in die entsprechende Routine. Die einzige, die zwingend benutzt werden muss – um die Spielerfigur mit location = RAUM_ODER_ENTERABLE; an einen definierten Anfangsort zu setzen – ist initialise().[7]

[7] und nicht initialize!

AfterLife() Wird aufgerufen, wenn deadflag einen anderen Wert als false (lebendig) oder 2 (gewonnen) annimmt und bietet die Gelegenheit, orangefarbenen Rauch zu verblasen und deadflag wieder auf false zu setzen.

AfterPrompt() Wird nach dem Eingabeprompt (aber vor dem Einlesen der Eingabe) aufgerufen. Üblicherweise der Ort, an dem Zitatboxen ausgegeben werden, weil dann sichergestellt ist, dass sie nicht wegscrollen. Der Prompt selber kann durch einen Eintrag für die Aktion Prompt in der Dispatcherroutine LibraryMessages.before (siehe 3.2.2 auf Seite 81) geändert werden.

Amusing() Wenn die Konstante AMUSING_PROVIDED definiert ist, ist hier der Ort, nach einem gewonnenen Spiel noch mehr oder weniger amüsante Zusatzinformationen unterzubringen.

BeforeParsing() Wird aufgerufen, nachdem die Eingabezeile gelesen wurde, Eingabepuffer und Parsingtabellen initialisiert und wn auf 1 gesetzt wurde. Die Routine darf die Tabellen verändern und muss wn nicht zurücksetzen.

ChooseObjects(OBJEKT,c) Wird vom Parser beim Parsing von multi-tokens oder bei Mehrdeutigkeiten aufgerufen.

Für multi-tokens gilt: Ist c false, möchte der Parser das OBJEKT von der Liste ausschließen, ist c true, möchte er es einschließen. Die Routine sollte false zurückgeben, um die Entscheidung des Parsers zu akzeptieren, 1, um das OBJEKT in die Liste einzuschließen und 2 um es auszuschließen.

Ist c==2, sucht der Parser Hilfe beim Auflösen einer Mehrdeutigkeit, die Routine sollte einen numerischen Wert zwischen 0 (OBJEKT völlig ungeeignet) und 9 (OBJEKT sehr gut geeignet) zurückgeben. Um dies zu entscheiden, steht die vom Parser beabsichtigte Aktion in der Variablen action_to_be.

DarkToDark() Wird aufgerufen, wenn der Spieler versucht, von einem dunklen Ort an einen anderen unbeleuchteten Ort zu gelangen; in frühen Adventures trat hier meist ein Monster namens *Grue* auf und sorgte für einen angemessenen Grund, deadflag auf true zu setzen.

DeathMessage() Für die Fälle von deadflag>=3 kann hier eine angemessene Nachricht zum Spielende gesetzt werden (nur die letzte, fettgedruckte und in Asterisks eingeschlossene Zeile, nicht der Text davor!)

GamePostRoutine() Eine Dispatcherroutine, analog zu after. Wird aufgerufen, wenn alle anderen after-Routinen die Aktion haben passieren lassen, siehe 3.3 auf Seite 104.

GamePreRoutine() Eine Dispatcherroutine, analog zu before. Wird vor allen anderen before-Routinen aufgerufen.

Initialise() Muss zu Spielbeginn einmal aufgerufen werden und dabei location auf einen Anfangswert (Raum oder enterable) setzen. Üblicherweise wird hier

auch der Begrüßungstext ausgegeben. Unmittelbar danach folgt das „Banner" des Spiels (Titel, Untertitel, Angaben zur Version von Compiler und Library). Gibt `initialise()` den Wert `2` zurück, wird die Ausgabe des Banners unterdrückt und sollte mit der Routine `Banner()` alsbald nachgeholt werden.

`InScope()` wird in zwei Situationen aufgerufen, wenn die Library „nachsieht", was im Blickfeld *(scope)* ist: Einmal mit dem Flag `et_flag` auf `true`, dann kann festgelegt werden, welche Objekte ein `each_turn` ausführen sollen. Ist das Flag `false`, wird normal nach einem *grammar token* geparst. Beide Male können Objekte mit `ScopeWithin()` und `PlaceInScope()` hinzugefügt werden. Die Auswirkung dieser Routine ist dramatisch vom Rückgabewert abhängig: `false` bedeutet, dass die Library danach noch alle „normal" im Blickfeld befindlichen Objekte hinzufügt; `true` bedeutet, dass nach `InScope()` abgebrochen wird und nur die hier explizit ins Blickfeld gerückten Objekte sich darin befinden.

`LookRoutine()` Wird am Ende jeder `##Look`-Aktion, d.h. am Ende jeder Raumbeschreibung aufgerufen.

`NewRoom()` Wird – noch bevor eine Raumbeschreibung ausgegeben wird – aufgerufen, wenn sich der Raum verändert, entweder infolge einer erfolgreichen Bewegung oder eines `PlayerTo()`.

`ParseNoun(OBJEKT)` bietet die Gelegenheit, in das Parsen der `name`-*property* des übergebenen `OBJEKTES` einzugreifen. Der Rückgabewert ist wie bei einer `parse_name`-Routine die Anzahl der übereinstimmendes Wörter, `false`, wenn keine Übereinstimmung gefunden wurde und `-1`; wenn die Routine die Entscheidung an den Parser zurückgibt; `wn` muss dann auf das Wort zeigen, an dem der Parser weitermachen soll (normalerweise mit `1`, aber nicht notwendigerweise).

`ParseNumber(TEXT,N)` Eine Gelegenheit, sich in das Parsen von Zahlwörtern einzumischen. Der zu parsende Text befindet sich in einem array `TEXT` mit der Länge `N`. Rückgabewert ist eine Zahl zwischen 1 und 10000; `false` bedeutet, dass keine Zahl erkannt wurde.

`ParserError(PE)` Zum Umdefinieren der Parser-Fehlermeldungen; mit `PE` wird eine Fehlernummer übergeben; der Rückgabewert ist `true`, wenn `ParserError()` eine angemessene Meldung selbst ausgegeben hat; `false` wenn die vordefinierte Meldung ausgegeben werden soll. Die Fehlerkonstanten und die vordefinierten Antworten (kursiv geschriebene Teile werden mit dem betreffenden Teil der nicht verstandenen Eingabe ausgefüllt) sind:

`STUCK_PE` Diesen Satz verstehe ich nicht.

`UPTO_PE` Ich habe dich nur soweit verstanden: . . .

`NUMBER_PE` Diese Zahl verstehe ich nicht.

`CANTSEE_PE` Du kannst nichts dergleichen sehen.

`TOOLIT_PE` Es sieht so aus, als hättest du zu wenig gesagt!

NOTHELD_PE Du trägst *das Objekt* nicht bei dir!

MULTI_PE [Mit diesem Verb kannst du nicht mehrere Objekte ansprechen.]

MMULTI_PE [Du kannst pro Zeile nur einmal mehrere Objekte ansprechen.]

VAGUE_PE Ich verstehe nicht, was mit *„dem Pronomen"* gemeint ist.

EXCEPT_PE Das ist nichts, worauf du dich in diesem Spiel beziehen musst.

ANIMA_PE Das kannst du nur mit belebten Objekten machen.

VERB_PE Dieses Verb verstehe ich nicht.

SCENERY_PE Ein flüchtiger Blick darauf sagt dir, dass es sich um nichts wichtiges für das Spiel handelt.

ITGONE_PE Im Augenblick ist *„es" (das Objekt)* nicht zu sehen.

JUNKAFTER_PE [Ich habe nicht verstanden, wie das endete.]

TOOFEW_PE Nur *Anzahl* davon sind ansprechbar.

NOTHING_PE Nichts zu tun!

ASKSCOPE_PE (Ausgabe der verwendeten `scope routine`)

PrintRank() Wird nach der Ausgabe der Punktezahl aufgerufen und ermöglicht Nachrichten im Stil von „... was Dir den Rang eines Amateurabenteurers verleiht."

PrintVerb(v) Wird aufgerufen, wenn der Parser in einer Nachfrage („Du möchtest ... aber was, womit...?") den Namen eines Verbs ausgeben soll und bietet eine Gelegenheit, dies zu ändern. `v` ist die Wörterbuchadresse des Verbs. Ein Rückgabewert von true bedeutet, dass die Routine selbst ein Verb ausgegeben hat; false bedeutet, dass der Parser das Verb ausdruckt, das er von Anfang an verwenden wollte (`v`).

PrintTaskName(N) Druckt den Text zur Aufgabe Nummer `N` aus (wird für `##Fullscore` benötigt). Üblicherweise ein `switch(n)`-Statement.

TimePasses() wird nach jedem Spielzug aufgerufen, der „Zeit" im Spiel kostet, also nicht bei Aktionen von Metaverben.

UnknownVerb(WORD) Wird aufgerufen, wenn der Parser ein unbekanntes Verb findet. Dies kann hier abgefangen und ein anderes Verb zurückgegeben werden, das der Parser stattdessen benutzen soll (z.B. 'hilf'). Wird false zurückgegeben, macht der Parser weiter und gibt eine „unbekanntes Verb"-Fehlermeldung aus.

3.2.5 Ausgaberegeln für Artikel und Pronomen

Diese Regeln sind tatsächlich ganz normale Routinen, die ein bestimmtes Objekt übergeben bekommen und dann im gewünschten Deklinationsfall und mit dem gewünschten Artikel oder – anstelle des Objektnamens – ein Pronomen als druckbaren *String*

ausgeben. Der String ist aber nicht der Rückgabewert, sondern der Seiteneffekt dieser Routinen! Deshalb müssen sie – innerhalb eines `print`-Befehls – so geschrieben werden:

```
print (REGEL) OBJEKT
```

Denn die „normale" Schreibweise

```
print REGEL(OBJEKT) ! SO NICHT!
```

führt zu keiner Fehlermeldung (diese Syntax ist für Routinen ja zulässig); gibt aber außer dem gewünschten String auch noch den Rückgabewert der Routine aus – meistens `true`, das als `1` gedruckt wird. Wenn in der Ausgabe eines Spiels unmotivierte Einsen stehen, sollte man nach einem solchen Fehler suchen.

Alle Regeln haben den Namen des gewünschten – grammatisch – männlichen Artikels oder Pronomens, der oder das ausgegeben werden soll; ob die weibliche oder neutrale Form gedruckt wird, hängt von den entsprechenden Attributen – `male, female, neuter` – des Objekts ab, ob Singular oder Plural verwendet werden, vom Attribut `pluralname`. Dekliniert wird außer dem `short_name` des Objekts auch jedes mit `adj` deklarierte Adjektiv; was unter `post` steht, wird nicht dekliniert.

Artikel[8]

Diese Regeln geben einen Artikel gefolgt vom korrekt deklinierten Namen des Objekts aus.

Artikel	Nominativ	Genitiv	Dativ	Akkusativ
bestimmt	(der)	(des)	(dem)	(den)
großgeschrieben	(GDer)	(Gdes)	(GDem)	(GDen)
unbestimmt	(ein)	(eines)	(einem)	(einen)
großgeschrieben	(GEin)	(GEines)	(GEinem)	(GEinen)
verneinend	(kein)	(keines)	(keinem)	(keinen)
großgeschrieben	(GKein)	(GKeines)	(GKeinem)	(Gkeinem)
nur dekliniert, kein Artikel	(_er)	(_es)	(_em)	(_en)

Hat das Objekt das Attribut `proper` (d.h. sein Name ist ein Eigenname), so wird grundsätzlich kein Artikel ausgegeben, vor einem Adjektiv aber wird der direkte Artikel ausgegeben.

Hat die *property* `article` des Objekts einen Eintrag, so wird dieser statt des unbestimmten Artikels vorangestellt, auch wenn es ein leerer String ist. Hat der Eintrag den Wert der Library-Konstante `definit` so wird immer der bestimmte Artikel verwendet.

[8] In der englischen Library gibt es nur die Regeln (the) und (a) sowie (IsOrAre).

Pronomen

Diese Regeln geben nur ein Pronomen, nicht aber den Namen des Objekts aus:

Pronomen	Nominativ	Genitiv	Dativ	Akkusativ
einfach	`(er)`	`(seiner)`	`(ihm)`	`(ihn)`
großgeschrieben	`(Ger)`	`(GSeiner)`	`(GIhm)`	`(GIhn)`
demonstrativ	`(dieser)`	`(dieses)`	`(diesem)`	`(diesen)`
großgeschrieben	`(GDieser)`	`(GDieses)`	`(GDiesem)`	`(Gdiesen)`

Kopulae

`(isorare)` gibt „ist" oder „sind" aus. Es kann aber auch einfach `(ist)` geschrieben werden. Analog gibt es `(hat)` für „hat" oder „haben" und `(wird)` für „wird" oder „werden". Das häufig vorkommende `(GEr) OBJEKT (isorare) OBJEKT` wird abgekürzt zu `(GEristSiesind)OBJEKT.`

Endungen von Verben

`(endT)` und `(endEt)` werden nach dem Stamm eines Verbs verwendet und evaluieren je nach Singular oder plural des folgenden Objekts zu t/en bzw. zu et/en.

Unregelmäßige Verben können mit der Konstruktion `singplur(OBJEKT,"SINGULARFORM","PLURALFORM")` „erschlagen" werden.

3.2.6 Attribute (attributes) von Objekten

Ein Attribut ist etwas, das ein Objekt entweder haben kann oder nicht. Es handelt sich also technisch um Flags bzw. boole'sche Werte, die gesetzt sein oder nicht gesetzt sein können.

Es können zusätzliche Attribute mit dem (globalen) Befehl

```
Attribute BEZEICHNER;
```

definiert werden. Inform kann maximal 48 Attribute verarbeiten.

In der Objektdefinition werden Attribute mit `has` gesetzt. Sie können später jederzeit mit `give` neu gesetzt oder geändert werden: `give Fliegenpilz edible` macht ihn essbar; zurückgesetzt werden sie mit der Tilde: `give Hintertuer ~locked` sperrt die Tür auf.

Getestet werden Attribute mit `has` und `hasnt`.

absent wird *zusammen mit dem Befehl* `remove` gebraucht, um Schiebekulissen, sogenannte „*floating objects*" verschwinden zu lassen, d.h. Objekte, die einmal deklariert und mit `found_in` (3.2.7 auf Seite 97) von mehreren Räumen aus gesehen werden können. (`remove` beseitigt das gerade vorhandene Exemplar; `absent` sorgt dafür, dass es beim nächsten Raumwechsel des Spielers nicht automatisch wieder hereingeschoben wird.)

animate kennzeichnet ein Lebewesen. Objekte mit diesem Attribut erkennt der Parser als `creature`.

clothing kennzeichnet ein Kleidungsstück, das mit *##disrobe* und *##wear* aus- und angezogen werden kann.

concealed versteckt das Objekt vor dem Spieler, ein ausdrückliches *##examine Osterei* findet es aber.

container ist ein Objekt in das man etwas hineintun (triggert *##Receive*) oder aus dem man etwas herausnehmen kann (triggert *##LetGo*. Kann nicht gleichzeitig das Attribut `supporter` besitzen.

door kennzeichnet nicht nur Türen, sondern auch Brücken. Wenn die Eigenschaft `door_to` richtig eingestellt ist, macht es aus einer *##Enter*–Aktion ein *##Go*.

edible Das Objekt ist essbar. Es reagiert auf *##eat*. *##Vomit* musst du selber definieren!

enterable macht das Objekt für den Spieler begehbar, aber nur, wenn das Objekt auf dem Boden steht.

female, male, neuter legen das grammatische Geschlecht eines Objekts fest. Das muss – anders als bei der englischen Library – auch bei unbelebten Objekten gemacht werden. Beachte: Erbt das Objekt von einer Klasse `male` oder `female`, ist aber selbst `neuter`, so muss das geerbte Geschlechtsmerkmal mit `~male` oder `~female` explizit gelöscht werden.

general verleiht dem Objekt keine besondere Befehlsgewalt, sondern kann für beliebige Zwecke verwendet werden.

light beleuchtet einen Raum, macht ein anderes Objekt zur Lichtquelle. `give player light` verbannt zuverlässig alle *Grues* aus dem Spiel.

lockable macht ein Objekt (insb. Türen, Container) verschließbar und für *##Lock* und *##Unlock* empfänglich. Der Spieler braucht den richtigen Schlüssel (definiert mit `with_key`) zum Aufschließen.

locked sorgt dafür, dass ein Objekt, das `lockable` hat, auch wirklich verschlossen ist.

moved zeigt an, ob das Objekt vom Spieler schon bewegt wurde.

on kennzeichnet ein eingeschaltetes `switchable` Objekt.

open kennzeichnet ein offenes `openable` Objekt.

openable ist ein Objekt, das mit *##Open* und *##Close* auf- und zugemacht werden kann, wenn nicht `locked` gesetzt ist. Wenn die Eigenschaften `when_open` und `when_closed` gesetzt sind, dann wird das Objekt unterschiedlich beschrieben, je nachdem, ob es offen oder zu ist.

pluralname kennzeichnet ein Objekt, dessen Name im Plural steht. Paradoxerweise muss aber der `short_name` dennoch im Singular angegeben werden – das Umformulieren macht die Library! Achtung bei umlautender Pluralform (die muss angegeben werden)!

Inkurs: ein altbekanntes Objekt[9] könnte z.B. so definiert werden:

```
Object SewerPipes Wellhouse
 with  name 'abfluss' 'rohr' 'rohre' 'abflussro' 'zwei' 'zwoelfzoe',
       dekl 1,
       short_name "Abflussrohr",
       adj "zwölfzöllig",
       article "zwei",
       description "Sie sind zu eng zum Durchschlufen.
                    Der einzige Ausgang ist im Westen.",
 has   neuter pluralname scenery;
```

Ein weiteres Beispiel: Die `Apfelblueten` in *Eden*.

proper kennzeichnet den Namen des Objekts als Eigennamen, so dass nie ein Artikel davor gesetzt wird.

scenery kennzeichnet ein Dekorationsstück: Ein solches Objekt ist automatisch `static` und wird nicht mit in die Ausgabe von *##Look* aufgenommen (weil es in der `description` des übergeordneten Objekts, i.d.R. des Raums, schon erwähnt wurde).

scored vergibt Punkte: bei einem Raum `ROOM_SCORE` Punkte für das erste Betreten, bei einem anderen Objekt `OBJECT_SCORE` Punkte für das erste An-sich-nehmen.

static macht ein Objekt unbeweglich, d.h. unempfindlich gegenüber *##Remove*, *##Turn*, *##Pull*, *##Push* und *##Take*.

supporter kennzeichnet ein Objekt, auf das man mit *##PutOn* Sachen draufstellen (und wieder wegnehmen) kann. Es reagiert auf *##Receive* und *##LetGo*.

switchable ist ein Objekt, das ein- und ausgeschaltet werden kann. Wenn die Eigenschaften `when_on` und `when_off` gesetzt sind, dann wird das Objekt unterschiedlich beschrieben, je nachdem, ob es ein- oder ausgeschaltet ist.

[9] die Wasserrohre aus dem Brunnenhaus von „Adventure", Crowther/Wood 1972.

`talkable` kennzeichnet Objekte, die kein `animate` haben, die aber trotzdem gegenüber Worten empfänglich sind (z.B.: *Mikrofon, hallo*).

`transparent` ist ein `container`, dessen Inhalt sichtbar ist.

`visited` ist ein Raum, in dem der Spieler schon gewesen ist.

`workflag` ist ein intern verwendetes Flag.

`worn` zeigt an, ob ein mit `clothing` gekennzeichnetes Kleidungsstück gerade getragen wird.

3.2.7 Eigenschaften (*properties*) von Objekten

Properties sind das eigentliche Fleisch der Objekte. Es gibt *common properties*, die alle Objekte von der Klasse `Object` geerbt haben. Daneben kann jedes Objekt weitere *properties* besitzen, die einfach in dem mit `with` eingeleiteten Teil der Objektdefinition (oder der Klassendefinition) zwischen den anderen *properties* definiert werden. *Common properties* sind für jedes Objekt definiert, d.h. ein (lesender) Zugriff ist möglich; ein schreibender aber nur, wenn sie in der Objektdefinition explizit belegt worden sind. Alle nachfolgenden *properties* der library sind *common*; *additiv* sind sind nur die, bei denen es angegeben ist. Wenn nicht anders angegeben, können die *properties* anstelle eines anderen Datentyps (in der Regel eines Objekts oder eines als Nachricht an den Spieler auszugebenden String) eine Routine enthalten, die einen solchen Datentyp zurückgibt oder einen String ausgibt.

`add_to_scope` Objekte, die automatisch zusammen mit dem definierten Objekt in (und aus) dem Blickfeld (*scope*) geraten. Hier steht entweder eine Liste von einzelnen Objekten oder eine Routine, in der die einzelnen Objekte mit `AddToScope()` oder `ScopeWithin()` ins Blickfeld bewegt werden.

`adj` **(deutsche Besonderheit)** enthält ein oder mehrere Adjektive (mehrere Adjektive als Liste, jedes in eigenen Anführungszeichen), die das Objekt im auszugebenden Text näher beschreiben. Sie werden mitdekliniert. *Beachte*: Es empfiehlt sich, solche Adjektive im `name`-*property* zu wiederholen; der Parser „schaut" nur in `name`, nicht in `adj`.

`after` **(additiv)** Eine Routine, die für alle *erfolgreichen* Aktionen der Klasse 2 aufgerufen wird, die in einem Raum stattfinden oder mit dem Objekt als `noun` ausgeführt werden (z.B. `##Take Apfel` landet in der `after`-Routine des Apfels und des Raums, in dem die Aktion stattfindet; aber nur dann, wenn der Apfel tatsächlich genommen werden konnte!). Zum Format siehe 3.1.4 auf Seite 73. Rückgabewert ist entweder `false` (die Library arbeitet weiter und gibt die vordefinierte Nachricht aus) oder `true` (keine weitere Ausführung, was unterstellt, dass die `after`-Routine schon alles erledigt hat).

article hier steht ein objektspezifischer unbestimmter Artikel, der statt „ein" verwendet werden soll, wenn das Objekt (z.B. eine Flüssigkeit) nicht zählbar ist (z.B. „viel", „wenig" ...) oder ein leerer String, was bedeutet, dass nie ein unbestimmter Artikel ausgegeben wird. Mit der Konstante definit kann die Ausgabe eines bestimmten Artikels erzwungen werden. *Beachte:* Für Objekte mit dem Attribut proper wird dieser Eintrag ignoriert; bei Objekten mit dem Attribut pluralname ist er nicht vonnöten (vor diesen setzt die Library nie ein „ein" ein ;-). Für ein komplett anderes Beispiel siehe den Inkurs bei pluralname 3.2.6.

articles ein array mit den Artikeln des Objekts (in der deutschen Library obsolet)

before **(additiv)** Eine Routine, die für alle Aktionen aufgerufen wird, die in einem Raum stattfinden oder mit dem Objekt als noun ausgeführt werden (z.B. ##Take Apfel landet in jedem Fall in der before-Routine des Apfels und des Raums, in dem die Aktion stattfindet und die Routinen haben die Chance, zu entscheiden, ob der Apfel gegessen werden kann oder nicht). Zum Format siehe 3.1.4 auf Seite 73. Rückgabewert ist entweder false (die Library arbeitet weiter, prüft, ob die Aktion evtl. Erfolg haben, durchgeführt werden und ins after-Stadium gelangen kann und gibt die vordefinierte Nachricht aus) oder true (keine weitere Ausführung, was unterstellt, dass die before-Routine schon alles erledigt hat bzw. dass nichts mehr geht.). Sonderfälle gibt es für

##Go wenn das Objekt ein Fahrzeug ist, in dem sich die Spielerfigur befindet. Die Rückgabewerte von before haben dann folgende Wirkung:

0 (false**)** lässt die Bewegung nicht zu und meldet dies.

1 (true**)** bewegt Spieler und Fahrzeug.

2 lässt die Bewegung nicht zu und meldet nichts.

3 lässt die Bewegung zu und meldet nichts (dies sollte auch der Rückgabewert sein, wenn die Ortsveränderung selbst programmiert wird).

##PushDir wenn das Objekt erfolgreich in eine Richtung (d.h. in einen anderen Raum) geschoben werden soll, muss die before-Routine AllowPushDir() aufrufen und true zurückgeben.

cant_go Enthält für einen Raum die Nachricht, die ausgegeben wird, wenn der Spieler versucht, in eine nicht mit d_to etc. definierte Richtung zu gehen.

capacity Die Zahl der Objekte, die ein container oder supporter fassen kann (Voreinstellung 100). Wird diese *property* für ein Spielerobjekt definiert, legt sie die Zahl der Objekte fest, die dieses Spielerfigur tragen kann (Voreinstellung in der Konstante MAX_CARRIED).

changing_gender Wenn ein Objekt diese Property hat, kann hinter einzelnen Wörterbucheinträgen der name-property ein vom grammatischen Geschlecht des Objekts abweichendes female, male oder neuter angegeben werden. Wird das Objekt im Spiel mit einem so als abweichend gekennzeichneten Synonym ange-

sprochen, wird das abweichende Geschlecht für nachfolgende Ausgaben berücksichtigt. Das vermeidet unschöne Begebenheiten wie *„Mach' das Tor auf."* – *„Sie* (nämlich *die* Tür) *öffnet sich."*

d_to und die *properties* für die anderen Richtungen: **e_to, in_to, n_to, ne_to, nw_to, out_to, s_to, se_to, sw_to, up_to, w_to** legen für Räume und begehbate Objekte fest, in welche Richtungen und wohin sie verlassen werden können. Sie sind vorbelegt mit `false`, was gleichbedeutend ist mit keinem Ausgang. `true` bedeutet ebenfalls kein Ausgang, aber ohne Benachrichtigung. Ein String ist gleichbedeutend mit keinem Ausgang, wobei der String ausgegeben wird und deshalb eine Nachricht enthalten sollte, warum es nicht weitergeht. Wenn diese *properties* ein Objekt, also einen anderen Raum oder eine Tür enthalten, existieren Weg und Ziel in dieser Richtung. Sie können auch eine Routine enthalten, die dann ihrerseits ein Objekt (Tür oder Raum), true, false oder einen String zurückgibt.

dekl **(deutsche Besonderheit)** Die Deklination wird festgelegt durch die Eigenschaft `dekl`, gefolgt von einer Zahl, die angibt, zu welchem Deklinationssystem das Objekt (der `short_name` des Objekts) gehört. `dekl 0` bedeutet, dass das Objekt gar nicht dekliniert wird.

1. Genitiv Singular mit -s, Plural mit -e, Dativ Plural mit -en: der Tag, das Jahr
2. Genitiv Singular mit -s, Dativ Plural mit -n: der Apfel, das Segel
3. Genitiv Singular mit -s, Plural mit -en: der Staat, das Auge
4. Genitiv Singular mit -s, Plural mit -er, Dativ Plural mit -ern: der Wald, das Bild
5. Genitiv Singular mit -s, Plural mit -s: der Opa, das Deck
6. Nominativ Singular ohne Endung, in allen anderen Fällen -en: der Mensch
7. Plural mit -e, Dativ Plural mit -en: die Kraft
8. nur im Dativ Plural mit -en: die Mutter
9. alle Fälle im Plural mit -en: die Frau
10. alle Fälle im Plural mit -s: die Oma

Dekliniert wird jedes Wort im `short_name`, außerdem ein mit `adj` definiertes Adjektiv. Nicht mitdekliniert wird der mit `post` nachgestellte Zusatz. Umlautungen im Plural (Apfel->Äpfel) kann die Library nicht.

daemon Eine Hintergrundroutine (Dämon), die in jedem Spielzug einmal ausgeführt wird, wenn sie aktiv ist. Der `daemon` eines `OBJEKTS` wird aktiviert mit `StartDaemon(OBJEKT)` und angehalten mit `StopDaemon(OBJEKT)`.

describe **(additiv)** eine Routine, die vor `description` aufgerufen wird (und regelmäßig besondere Zusätze zur Beschreibung ausgibt). Ist der Rückgabewert `false`, macht die Library mit der `description` weiter, ist er `true`, wird die `description` nicht mehr ausgegeben.

description enthält die ausführliche Beschreibung eines Raums oder die mit ##Examine abfragbare Beschreibung eines anderen Objekts.

door_dir enthält eine der Richtungen d_to etc. (oder eine Routine mit diesem Rückgabewert) und gibt damit für Türen an, in welche Richtung (hinein) sie führen, d.h. wohin man gehen muss, um in die Tür zu kommen. Auch die Objekte d_obj etc. des compass können mit door_dir umgeleitet werden.

door_to legt fest, ob und wohin es aus einer Tür heraus geht. Mögliche Inhalte wie d_to.

each_turn **(additiv)** Eine Routine, die nach jedem Zug aufgerufen wird, solange das Objekt im Blickfeld *(scope)* des Spielers ist.

found_in legt für Schiebekulissen *(floating objects)*, die sich mit dem Spieler von Raum zu Raum bewegen, fest, in welchen Räumen sie zu sehen sind. Es enthält entweder eine Liste von Räumen, un denen die Kulisse vorhanden sein soll und von Objekten, in deren Gegenwart sie vorhanden sein soll; oder aber eine Routine, die true liefert, wenn die Kulisse da sein soll und false, wenn nicht. Ist das Attribut absent gesetzt, bleibt found_in ohne Wirkung.

grammar eröffnet die Möglichkeit, Objekten mit den Attributen animate oder talkable eine eigene, von der allgemeinen Grammatik der Library abweichende Grammatik zu geben (Standardbeispiel: der mit Sprachbefehlen steuerbare Navigationscomputer interpretiert „HAL, nimm Kurs auf Jupiter" nicht als ##Take-Befehl, sondern als besondere Anweisung). Die Routine kann auf den Variablen verb_word (das vom Parser als Verb erkannte Wort) und verb_wordnum (seine Position in der Eingabe) aufbauen. Für den Rückgabewert gibt es mehrere Möglichkeiten:

- **false** nichts gefunden, der Parser soll alleine weitermachen;
- **true** alles erledigt, action, noun und second gesetzt;
- **'Verb'** Ein Verb (sein Wörterbucheintrag), dessen Grammatikzeilen der Parser verwenden soll (wenn z.B. im obigen Beispiel „Kurs" ein Objekt ist, das auf ##SetTo reagiert, könnte man 'stell' zurückgeben);
- **-'Verb'** (der negative Wert des Wörterbucheintrags): wie 'Verb' verfahren, aber anschließend die normalen Grammatikzeilen für das tatsächlich geschriebene Verb (verb_word) prüfen.

initial Für einen Raum: Die Beschreibung bei Ankunft des Spielers in diesem Raum. Für ein Objekt: Die Beschreibung des Objekts in der Raumbeschreibung solange das Objekt nicht moved hat. Sollte für Türen, Container und switchable Objekte nicht verwendet werden (siehe stattdessen unten when_closed etc.).

inside_description Die Raumbeschreibung, wenn sich der Spieler in diesem Objekt aufhält.

invent Hier kann nur eine Routine stehen, die dann bei Erstellung eines Inventars zweimal aufgerufen wird (mit der Variablen `inventory_stage` auf `1` oder auf `2`). Beim ersten Aufruf ist für das Objekt noch nichts ausgegeben, die Routine kann `false` zurückgeben und sich nicht einmischen oder den Inventareintrag vollständig ausgeben und `true` zurückgeben. Der zweite Aufruf erfolgt, wenn die Library schon den Namen des Objekts ausgedruckt hat („eine Taschenlampe"); die Routine kann dann Zusätze schreiben („kurz vor dem Erlöschen") und `true` (Schluss des Eintrags, keine weitere Ausgabe) oder `false` zurückgeben (die Library macht weiter).

life **(additiv)** Eine Dispatcherroutine, die ausschließlich bei `animate`-Objekten gebraucht wird und wie eine `before`-Routine aufgebaut ist, jedoch ausschließlich Regeln für die speziellen Aktionen `##Attack`, `##Kiss`, `##WakeOther`, `##ThrowAt`, `##Give`, `##Show`, `##Ask`, `##Tell`, `##Answer`, `##Order` enthält.

list_together steuert, ob das Objekt in Listen (Inventar, Raumbeschreibungen) zusammen mit anderen ähnlichen Objekten in einer Gruppe zusammengefasst wird: Enthält es eine Zahl, werden alle Objekte mit dieser Zahl zusammengefasst. Enthält es einen String (z.B. „Brücken") werden alle Objekte mit diesem String zusammengefasst und eine Zeile mit der Anzahl und dem String vorangestellt („Sieben Brücken:").
Komplizierter wird es mit einer Routine: Diese wird zweimal aufgerufen, wobei die Variable

name **(additiv)** enthält ein oder mehrere Wörter (in Hochkommata), die in das Wörterbuch des Parsers eingetragen werden und unter denen der Parser das Objekt wiedererkennt. Hier werden der kleine rostige und der große silberne Schlüssel auseinandergehalten. `name` ist ein `array` aus Wörterbucheinträgen, eine Routine kann hier nicht stehen (dafür gibt es `parse_name`).

Umlaute und ß müssen hier in ae oe ue und ss aufgelöst werden, sonst schluckt der Parser die Wörter nicht!

Falls Synonyme ein anderes grammatisches Geschlecht als das eigentliche Objekt aufweisen, kann dieses Geschlecht (`male/female/neuter`) ohne Hochkomma hinter dem jeweiligen Synonym angegeben werden. Das Objekt muss in diesem Fall die *Property* `changing_gender` besitzen. So wird sichergestellt, das das Spiel das korrekte Pronomen verwendet und auf: *„Nimm die Frucht"* mit *„Du hast sie nicht."* und nicht etwa mit *„Du hast ihn* [nämlich den Apfel] *nicht."* antwortet.

Für Räume soll `name` anderes bewirken: in der Beschreibung der Szenerie verwendete Wörter werden dem Parser bekannt gemacht, um häßliche Meldungen á la Ich-sehe-das-Objekt-nicht-das-ich-dir-eben-erst-beschrieben-habe zu vermeiden und durch „Das ist nicht wichtig" o.ä. zu ersetzen.

Das funktioniert leider in der deutschen Fassung nur eingeschränkt, nämlich ohne Berücksichtigung der Artikel: *„Untersuche Raum"* führt zu der als `SCENERY_PE` dafür vorgesehenen Nachricht. *„Untersuche den Raum"* aber zur unerwünschten Fehlermeldung `CANTSEE_PE`.

number Frei zur beliebigen Verwendung.

orders Eine Dispatcherroutine eines Objekts mit `animate` oder `talkable`. Hier wird festgelegt, wie dieses Objekt auf Befehle („Eva, gib mir den Apfel") reagiert.

parse_name enthält eine Routine, die die Arbeit des Parsers unterstützt: Sie sollte mit `NextWord()` die Eingabezeile wortweise lesen und ihr Ergebnis (Zahl der passenden Wörter der Eingabe) zurückmelden. Findet sie dabei Pluralbezeichnungen 'ihres' Objekts, sollte sie die Variable `parser_action` auf `##PluralFound` setzen. Den Eingabezeiger `wn` muss die Routine dabei nicht konservieren.
Wird die Routine mit `parser_action==##TheSame` aufgerufen, kann sie prüfen, ob die Objekte `parser_one` und `parser_two` unterscheidbar sind oder nicht.
Mögliche Rückgabewerte sind:

- **false** Die Eingabe passt nicht auf das Objekt.
- **n >= 1** Die Zahl der übereinstimmenden Worte in der Eingabe
- **-1** normaler Aufruf: Die Aufgabe wird an den Parser zurückgegeben. Aufruf mit `##TheSame`: Die Objekte sind nicht unterscheidbar.
- **-2** Aufruf mit `##TheSame`: Die Objekte sind verschieden.

plural Die Pluralbezeichnung eines mehrfach vorkommenden Objekts, z.B. "Kieselsteine".

post **(deutsche Besonderheit)** Ein nachgestellter Namensbestandteil, der nicht dekliniert wird, z.B.: die Wasserpfeife *des Großwesirs*

react_after Eine Dispatcherroutine, die im `before`-Stadium aufgerufen wird, wenn das Objekt im Blickfeld (scope) ist.

react_before Eine Dispatcherroutine, die im `after`-Stadium aufgerufen wird, wenn das Objekt im Blickfeld (scope) ist.

short_name Der – auszugebende – Name des Objekts im Singular. Mit `dekl` muss noch angegeben werden, ob und wie er dekliniert werden soll. Für Pluralobjekte ist das Attribut `pluralname` zu setzen.

suffixes enthält ein Array oder eine Routine, die unregelmäßige Deklinationsendungen angibt (`dekl` muss auf 0 gesetzt sein). Ein Array besteht aus 4 Strings für die Endungen der 4 Fälle, eine Routine bekommt als ersten Parameter den Fall (als eine der Konstanten `Nom Gen Dat` oder `Akk`) übergeben.

`short_name_indef` Der `short_name`, wenn er mit einem unbestimmten Artikel benutzt wird (in der deutschen Library obsolet)

time_left Der aktuelle Stand des Timers. *Beachte:* Der Startwert wird nicht hier, sondern mit der Routine `StartTimer()` zugewiesen.

time_out Die Routine, die ausgeführt wird, wenn der Timer des Objekts abgelaufen ist.

when_closed, when_open Unterschiedliche zusätzliche Beschreibungen für eine Tür oder einen container. Sie werden nur zusammen mit der ausführlichen Beschreibung des Raumes ausgegeben (d.h. bei ##Look).

when_off, when_on Unterschiedliche zusätzliche Beschreibungen für ein wegen switchable ein- und ausschaltbares Objekt. Sie werden nur zusammen mit der ausführlichen Beschreibung des Raumes ausgegeben (d.h. bei ##Look).

with_key Name eines Objekts, in der Regel eines Schlüssels, mit dem ein Objekt mit dem Attribut lockable vom Spieler auf- und zugesperrt werden kann.

3.2.8 Aktionen und Verben

"Verben" sind die – deutschen – Verben im grammatischen Sinn, die der Parser versteht und in Aktionen umsetzt. Ein Verb kann verschiedene Aktionen auslösen, je nachdem, wie der vom Verb eingeleitete Satz aussieht, insbesondere, welche Präpositionen er aufweist.

"Aktionen" sind die – in englischer Sprache definierten und hier mit zwei führenden Gattern versehenen – Befehle, die das Spielprogramm ausführt und die im Programm verwendet werden können. Eine Aktion kann durch viele verschiedene Verben ausgelöst werden.

Definition von Aktionen

Eine Aktion mit dem Namen ##AKTIONSNAME wird definiert, indem man eine – selbständige – Routine mit dem Namen AKTIONSNAMESub schreibt und wenigstens ein Verb so definiert, dass es diese Aktion auslöst.

Für eine Aktion der Klasse 3 – die im allgemeinen „nichts" macht, als Text auszugeben – genügt eine Routine mit einem Befehl, der eine passende Standardantwort ausgibt, z.B. für eine Aktion ##Xyzzy:

```
[ XyzzySub;
    "Nichts geschieht.";
];
```

Eine Aktion der Klasse 2, die tatsächliche Veränderungen bewirkt; Objekte bewegt oder ihren Zustand verändert etc., muss – falls die beabsichtigte Veränderung stattgefunden hat, der Befehl des Spielers also „erfolgreich" war – die Routine AfterRoutines() aufrufen; typischerweise als *vorletzte* Anweisung wie folgt:

```
if (AfterRoutines()) rtrue; [10]
```

[10] auch `return AfterRoutines();` sollte funktionieren.

und danach – als *letzte* Anweisung – den für den erfolgreichen Abschluss der Aktion standardmäßig vorgegebenen Text ausgeben (der auf diese Weise übersprungen wird, wenn eine `after`-Routine `true` zurückgibt). So erst wird das für die *properties* `after` und `react_after` beschriebene Verhalten erzielt.

Aufruf von Aktionen

Aktionen kann nicht nur der Spieler auslösen, sondern auch die Autorin. Dafür wird der Name der Aktion – ohne Gatter – optional gefolgt von ein oder zwei Objekten (dem noun und dem second der Aktion) – in spitze Klammern eingeschlossen:

```
<Dig Acker Spaten> ;
oder
<<Dig Acker Spaten>>;
```

Die zweite Variante (mit Doppelklammern) beinhaltet ein implizites `return true;` nach Ausführung der Aktion.

***Beachte*:** Der Parser wird dabei umgangen. Ob die Aktion Sinn macht, insbesondere ob die Objekte überhaupt im Blickfeld *(scope)* sind, muss hier die Autorin entscheiden.

Definition von Verben

Sie besteht aus dem Schlüsselwort `verb` gefolgt von einer Liste von Synonymen (als Stamm der Befehlsform, d.h ohne auslautendes „e”, in Hochkommata gefasst); gefolgt von einer oder mehreren Grammatikzeilen. Jede Grammatikzeile wird eingeleitet von einem Asterisk `*` als Platzhalter für das Verb oder eines der Synonyme, optional gefolgt von einem Satz aus Literalen (die meist Präpositionen darstellen) und anderen Tokens, die die Objekte des Satzes repräsentieren; danach einem Pfeil `->` gefolgt vom Namen der auszulösenden Aktion und eventuell dem Schlüsselwort `reverse`. Der Parser geht später die Grammatikzeilen von oben nach unten durch und hält beim ersten erfolgreichen Vergleich an.

Ein (nicht nachahmungsfähiges) Beispiel:

```
verb 'tret' 'tritt' 'steig' 'kletter'
    *                       -> VagueGo
    * 'in' container         -> Enter
    * 'auf' supporter        -> Enter
    * 'auf'/'ueber' noun     -> Climb
    * 'mit' held 'auf' noun -> Climb reverse;
```

Der Schrägstrich zwischen *auf* und *über* bedeutet, dass diese zwei Präpositionen alternativ stehen können. Die Angabe von `reverse` macht nur Sinn bei Grammatikzeilen, die eine Aktion mit einem direkten und einem indirekten Objekt verwenden: Normalerweise folgt erst das erste, i.d.R. direkte Objekt (das `noun`) dann das zweite Objekt (`second`). Ist aufgrund des verwendeten Satzbaus die Reihenfolge umgekehrt, muss `reverse` angegeben werden.

Metaverben – die nicht als Spielzüge zählen, keine Zeit verstreichen lassen und kein *'each turn'* auslösen – werden definiert, indem das Schlüsselwort `meta` nach `verb` und vor der Synonymliste eingefügt wird.

Token (Zeichen) der Grammatik

`'TEXT_IN_HOCHKOMMATA'` eben dieses Textliteral; mehrere Literale können mit einem Schrägstrich verbunden werden und gelten dann alternativ.

`noun` jedes beliebige Objekt im Blickfeld (*scope*)

`held` Ein Objekt, das die ausführende Person hält (das muss nicht notwendigerweise der Spieler selbst sein!)

`multi` ein oder mehrere Objekte im Blickfeld (*scope*)

`multiheld` ein oder mehrere Objekte, die die ausführende Person hält

`multiexcept` ein oder mehrere Objekte im Blickfeld (*scope*), außer dem angegebenen Objekt

`multiinside` ein oder mehrere Objekte im Blickfeld (*scope*), die innerhalb eines anderen Objekts sind

`NAME_EINES_ATTRIBUTS` jedes beliebige Objekt im Blickfeld (*scope*), bei dem dieses Attribut gesetzt ist

`creature` jedes beliebige Objekt im Blickfeld (*scope*) mit dem Attribut `animate`

`noun=``NAME_EINER_ROUTINE()` Jedes Objekt im Blickfeld, für das die Routine den Rückgabewert true liefert. Das noun wird an die Routine übergeben in der globalen Variable `noun`. Häufig von der Library benutzt in `noun=ADirection`, das prüft, ob das Objekt des Satzes eine gangbare Richtung darstellt.

`scope=``NAME_EINER_ROUTINE()` Jedes Objekt im durch die Routine definierten Blickfeld (für ein Beispiel siehe `GinfoG.h).` Die Routine wird dreimal aufgerufen, wobei die Variable `scope_stage` nacheinander auf `1`, `2` und `3` gesetzt ist. In den verschiedenen Stadien sollte sie arbeiten wie folgt:

1. Rückgabewert ist `true`, wenn ann dieser Stelle wie in einem `multi`...-token mehrere Objekte akzeptiert werden sollen, `false`, wenn nur ein Objekt zulässig ist.

2. mit den Routinen `ScopeWithin()` und `PlaceInScope()` werden Objekte ins Blickfeld gesetzt. Rückgabewert ist `true`, wenn nur diese Objekte im Blickfeld sein sollen; `false`, wenn danach noch die Library alle Objekte hinzunehmen soll, die ohnehin im Blickfeld sind (es ist unschädlich, wenn auf diese Weise ein Objekt zweimal aufgeführt wird).
3. das Objekt wurde nicht erkannt; die Routine darf eine eigene Fehlermeldung ausgeben.

`number` Eine Zahl zwischen 1 und 10.000, für Zahlen bis zwanzig auch ausgeschrieben.

`special` eine Zahl (wie `number`) oder ein (einzelnes) Wort.

`NAME_EINER_ROUTINE()` jeder Text, den diese Routine akzeptiert. Die Routine muss den Eingabestrom an dieser Stelle selbständig weiter parsen und einen von mehreren möglichen vordefinierten Werten zurückgeben (siehe DM[4], p. 226 f.)

`topic` jeder beliebige Text

`hinein` Überliest 'hinein' und 'rein'.

`heraus` Überliest 'heraus', 'herunter' und 'raus'.

`xhinweg` Überliest 'weg', 'fort', 'hinweg' und 'hinfort'.

`xhinein` Liest 'hinein' und 'rein' und ersetzt es mit dem zuletzt benutzten Objekt .

`xheraus` Liest 'heraus', 'herunter' und 'raus' und ersetzt es mit dem zuletzt benutzten Objekt .

`xdamit` Liest 'damit' etc. und ersetzt es mit dem zuletzt benutzten Objekt.

`xdarauf` Liest 'darauf', 'drauf', 'herauf' und 'drauf' und ersetzt es mit dem zuletzt benutzten Objekt.

Umdefinieren von Verben

Neue Synonyme lassen sich mit `verb =` einfach hinzufügen und „erben" alle bereits definierten Grammatikzeilen:

```
verb 'NEUES_SYNONYM' 'NOCH_EIN_SYNONYM' ... = 'SCHON_DEFINIERTES_VERB';
```

Soll die Grammatik eines bereits definierten Verbs abgeändert werden, können neue Grammatikzeilen mit dem Schlüsselwort `extend` definiert werden:

```
extend 'SCHON_DEFINIERTES_VERB' MODUS
```

gefolgt von den neuen Grammatikzeilen. `MODUS` ist dabei eines der Schlüsselwörter

`first` Die neuen Zeilen werden vor allen anderen eingefügt.

`last` Die neuen Zeilen werden erst nach allen schon vordefinierten Zeilen für dieses Verb eingefügt.

`replace` Die bisher für dieses Verb definierten Zeilen werden gelöscht und durch die neuen Zeilen ersetzt.

Obwohl hier nur *ein* Verb und keine Synonymliste angegeben werden kann, gilt die Definition für *alle Synonyme* des nach `extend` angegebenen Verbs. Will man die Grammatik nur für bestimmte Synonyme aus der Liste erweitern, schreibt man:

```
extend only 'EIN_VERB' 'NOCH_EIN_VERB' MODUS
```

wiederum gefolgt von den neuen Grammatikzeilen.

3.3 Und es geschah also: Ausführungsreihenfolge

Eine `AKTION` mit einem bestimmten `DIREKTEN_OBJEKT` (`noun`) wird in folgenden Schritten abgearbeitet.

1. Das `before`-Stadium
   a) Aufruf der `GamePreRoutine()`
   b) Aufruf von `player.orders()`
   c) Aufruf der `react_before` aller im Blickfeld (*scope*) befindlichen Objekte
   d) Aufruf der `before`-Routine des Raums
   e) Aufruf von `DIREKTES_OBJEKT.before()`
2. Das *during*-Stadium (Library oder selbst definierte Routine der `AKTION`);

   nur wenn die `AKTION` erfolgreich ist bzw. wenn `AKTIONSub()` (d.h. die selbst definierte Aktionsroutine) `AfterRoutines()` aufruft, folgt:
3. Das `after`-Stadium
   a) Aufruf der `react_after` aller im Blickfeld *(scope)* befindlichen Objekte einschließlich der Spielerfigur (`player`)

b) Aufruf der `after`-Routine des Raums

c) Aufruf von `DIREKTES_OBJEKT.after()`

d) Aufruf der `GamePostRoutine()`

Beachte: In allen Stadien von 1 und 3 (`before` und `after`) wird die weitere Ausführung abgebrochen, sobald auch nur eine der aufgerufenen Routinen `true` zurückgibt.

4 Nimm und iss: Die Standardaktionen und ihre Verben

Hier sortiere ich nach Aktionen und liste die sie auslösenden Verben und Grammatikzeilen auf. Eine nach Verben geordnete Aufstellung ist unmittelbar in der Library-Datei `germang.h` enthalten.

Die Aktionen sind in vier Gruppen aufgeteilt: Gruppe 1 umfasst die „Metaverben" und das Vokabular für die Fehlersuche; Gruppe 2 die von Anfang an wirksamen Aktionen; Gruppe 3 die nur vorbereiteten Aktionen (deren Auswirkungen mit `before` erst definiert werden müssen) und Gruppe 4 die nur implizit durch andere Aktionen aufgerufenen „fake actions". Gruppen 2 und 3 bilden den eigentlichen "Wortschatz" des Spielers, weshalb ich sie auch zusammengefasst habe.

Beachte: Der Parser trennt alle Verben erstmal von einer eventuellen Imperativendung -e. Beim Lesen der nachfolgenden Auflistung muss man sich also immer ein -e dazudenken.

4.1 Metaverben

4.1.1 Spielsteuerung

##FullScore

meta 'punkt' 'punktestand' 'scor'

* 'voll' -> FullScore

meta 'voll' 'fullscor'

* -> FullScore
* 'punkte' -> FullScore

##LMode1

meta 'knapp' 'normal'

* -> LMode1

##LMode2

meta 'ausfuehrlich' 'lang'

* -> LMode2

##LMode3

meta 'superknapp' 'superkurz' 'kurz'

* -> LMode3

##NotifyOff

meta 'punkt' 'punktestand' 'scor'

* 'off'/'aus'/'ausschalten' -> NotifyOff

meta 'notify' 'meldungen'

* 'off'/'aus'/'ausschalten' -> NotifyOff

##NotifyOn

meta 'punkt' 'punktestand' 'scor'

* 'on'/'an'/'ein'/'einschalten' -> NotifyOn

meta 'notify' 'meldungen'

* 'on'/'an'/'ein'/'einschalten' -> NotifyOn

##Objects

meta 'objekt' 'objects' 'gegenstaend' 'ding' 'sachen'

* -> Objects

##Places

meta 'ort' 'places' 'plaetz' 'raeum'

* -> Places

##Score

meta 'punkt' 'punktestand' 'scor'

* -> Score

##Quit

meta 'q//' 'quit' 'stirb' 'end' 'by' 'abbruch' 'beend' 'schluss'

* -> Quit

##Restart

meta 'neustart' 'wiedergeburt' 'neu'

* -> Restart

##Restore

meta 'lad' 'laden' 'load' 'restor'

* -> Restore

##Pronouns

meta 'pronomen' 'fuerwort'

* -> Pronouns

##Save

meta 'speicher' 'sicher' 'sav' 'speichern'

* -> Save

##ScriptOff

meta 'skript' 'script' 'transkript' 'transcript' 'mitschrift'

* 'off'/'aus' -> ScriptOff

##ScriptOn

meta 'skript' 'script' 'transkript' 'transcript' 'mitschrift'

* -> ScriptOn

* 'on'/'an'/'ein' -> ScriptOn

##Verify

meta 'filecheck' 'verify'

* -> Verify

##Version

meta 'version'

* -> Version

##Places und ##Objects können mit dem Compilersymbol `NO_PLACES` ausgeschaltet werden. Das nun folgende Debug-Vokabular wird nur einkompiliert, wenn die Compilerschalter `-S` und/oder `-D` „eingeschaltet" sind:

4.1.2 Debugging-Verben

##ActionsOff

meta 'actions'

* 'off' -> ActionsOff

##ActionsOn

meta 'actions'

* -> ActionsOn
* 'on' -> ActionsOn

##ChangesOff

meta 'changes'

* 'off' -> ChangesOff

##ChangesOn

meta 'changes'

* -> ChangesOn
* 'on' -> ChangesOn

##CommandsOff

meta 'recording'

* 'off' -> CommandsOff

##CommandsOn

meta 'recording'

* -> CommandsOn
* 'on' -> CommandsOn

##CommandsRead

meta 'replay'

* -> CommandsRead

##Dekliniere

meta 'deklinier' 'dekl'

* noun -> Dekliniere

##DekliniereAll

meta 'deklinier' 'dekl'

* 'komplett' noun -> DekliniereAll

##GlkList (nur unter Glulx; listet ale Glk-Objekte)

meta 'glklist'

* -> Glklist

##GoNear

meta 'gonear'

* noun -> Gonear

##GoTo (die Nummer ist die eines Raumes, herauszukriegen mit *tree*)

meta 'goto'

* number -> Goto

##Predictable

meta 'random'

* -> Predictable

##RoutinesOff

meta 'routines' 'messages'

* 'off' -> RoutinesOff

##RoutinesOn

meta 'routines' 'messages'

* -> RoutinesOn
* 'on' -> RoutinesOn

##Scope

meta 'scop'

* -> Scope
* noun -> Scope

##ShowObj

meta 'showobj'

* -> Showobj
* multi -> Showobj

##ShowVerb

meta 'showverb'

* special -> ShowVerb

##TimersOff

meta 'timers' 'daemons'

* 'off' -> TimersOff

##TimersOn

meta 'timers' 'daemons'

* -> TimersOn
* 'on' -> TimersOn

##TraceLevel

meta 'trac'

* number -> TraceLevel

##TraceOff

meta 'trac'

* 'off' -> TraceOff

##TraceOn

meta 'trac'

* -> TraceOn
* 'on' -> TraceOn

##TraceOff

meta 'trac'

* 'off' -> TraceOff

##XAbstract

meta 'abstract'

* noun 'to' noun -> XAbstract

##XPurloin

meta 'purloin'

* multi -> XPurloin

##XTree

meta 'tre'

* -> XTree
* noun -> XTree

4.2 Der eigentliche Wortschatz

##Answer

'antwort' 'schrei' 'beantwort'

* creature topic -> Answer reverse
* topic 'zu' creature -> Answer

##Ask

'frag' 'befrag'

* creature -> Ask
* creature 'ueber' topic -> Ask
* creature 'nach'/'ueber'/'ob' topic -> Ask

##AskFor

'frag' 'befrag'

* creature 'nach' noun -> AskFor[1]

'bet' 'bitt'

* creature 'um' noun -> AskFor

##Attack

'greif' 'fass' 'ergreif'

* creature 'an' -> Attack
* creature 'mit' held 'an' -> Attack

[1] Diese Zeile kann zu Schwierigkeiten führen, da "frag nach" normalerweise in erster Linie im Sinn von ##Ask verstanden wird.

* 'mit' held creature 'an' -> Attack reverse
* creature 'an' 'mit' held -> Attack

'pack'

* creature -> Attack

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

* 'dich' 'auf' creature -> Attack
* 'dich' 'gegen' noun -> Attack
* creature 'nieder' -> Attack

'mach'

* noun 'kaputt' -> Attack

'tret' 'tritt'

* noun -> Attack

'attackier' 'brich' 'brech' 'zerbrich' 'vernichte' 'zerstoer' 'zerschlag' 'zertruemmer' 'destroy' 'kaempf' 'toet' 'kick' 'ermord' 'mord' 'bekaempf' 'folter' 'quael' 'pruegel'

* noun -> Attack
* noun 'mit' held -> Attack
* noun xdamit -> Attack

'schlag' 'hau'[2]

* creature -> Attack
* xhinein -> Attack
* xdarauf -> Attack
* noun -> Attack
* 'auf' noun -> Attack
* 'mit' held 'gegen' noun -> Attack reverse
* held 'gegen' noun -> Attack reverse
* noun 'um'/'ab'/'entzwei'/'kaputt' -> Attack
* xdamit 'gegen' noun -> Attack reverse

'schneid' 'trenn' 'loese' 'spalte' 'zertrenn' 'durchschneid' 'teil' 'zerteil' 'zerschneid'

* creature -> Attack

##Blow

'blas' 'pust'

* held -> Blow
* 'in'/'auf' held -> Blow
* xhinein -> Blow
* xdarauf -> Blow

##Burn

'steck'

* noun 'an' -> Burn
* noun 'mit' held 'an' -> Burn
* 'mit' held noun 'an' -> Burn

[2] Wegen der von 'hau' nicht geteilten Bedeutung von 'schlag nach' für ##Consult ist diese Verbindung nicht glücklich. In GInfoG.h habe ich sie aufgelöst, aber noch nicht alle 'schlagenden' Probleme beseitigt.

'zuend' 'brenn'

* noun 'an'/'nieder' -> Burn
* noun 'mit' held 'an' -> Burn
* noun xdamit -> Burn
* 'mit' held noun 'an' -> Burn reverse
* xdamit noun -> Burn reverse

'entzuend' 'entflamm' 'verbrenn'

* noun -> Burn
* noun 'mit' held -> Burn
* 'mit' held noun -> Burn
* noun xdamit -> Burn
* xdamit noun -> Burn

##Buy

'biet'

* 'fuer' noun -> Buy

'kauf' 'erwerb' 'erwirb'

* noun -> Buy

##Climb

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* 'auf' noun -> Climb

'kletter' 'klettr' 'steig' 'erklimm' 'erkletter'

* noun -> Climb
* noun 'hoch' -> Climb
* 'auf'/'ueber' noun -> Climb
* 'auf' noun 'hoch' -> Climb
* xdarauf -> Climb

##Close Klasse 2: schließt Türen und Container.

'mach'

* noun 'zu' -> Close

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun 'zu' -> Close

'dreh' 'rotier' 'schraub'

* noun 'zu' -> Close

'schlag' 'hau'

* noun 'zu' -> Close

'schliess' 'sperr'

* noun -> Close

'verschliess' 'verriegel' 'verriegl'

* noun -> Close

##Consult

'forsch' 'lern'

* 'nach'/'ob'/'ueber'/'von' topic 'in'/'aus' noun -> Consult

* 'in'/'aus' noun 'nach'/'ob'/'ueber'/'von' topic -> Consult

'konsultier'

* noun 'ueber' topic -> Consult
* noun 'bezueglich' topic -> Consult

'schau' 'seh' 'l//' 'sieh' 'blick'

* topic 'nach' 'in' noun -> Consult
* 'nach' topic 'in' noun -> Consult
* 'in' noun 'ob'/'ueber' topic 'nach' -> Consult
* 'ueber'/'ob' topic 'in' noun 'nach' -> Consult

'lies' 'les'

* 'ueber'/'von' topic 'in' noun -> Consult
* 'ueber'/'von' topic 'in' noun 'nach' -> Consult
* topic 'in' noun -> Consult
* 'in' noun 'ueber'/'von' topic -> Consult

'such' 'spuere' 'stoeber' 'wuehl'

* 'in' noun 'nach' topic -> Consult
* 'nach' topic 'in' noun -> Consult;

'schlag' 'hau'[3]

* 'in' noun 'ueber' topic 'nach' -> Consult
* 'in' noun topic 'nach' -> Consult
* topic 'in'/'nach' noun 'nach' -> Consult ! fehlendes reverse
* topic 'nach' 'in' noun -> Consult ! fehlendes reverse

##Cut

'schneid' 'trenn' 'loese' 'spalte' 'zertrenn' 'durchschneid' 'teil' 'zerteil' 'zerschneid'

* noun -> Cut
* noun 'durch' -> Cut
* 'mit' noun -> Cut
* noun 'mit' noun -> Cut reverse
* noun 'durch'/'ab' 'mit' noun -> Cut reverse
* noun 'mit' noun 'durch'/'ab' -> Cut reverse
* noun 'auf' -> Cut
* xdamit -> Cut
* noun xdamit -> Cut reverse
* noun 'durch'/'ab' xdamit -> Cut reverse
* noun xdamit 'durch'/'ab' -> Cut reverse

##Dig

'grab' 'hack'

* -> Dig
* noun -> Dig

[3] siehe die Anmerkung bei ##Attack

- 'in' noun -> Dig
- 'in' noun 'mit' held -> Dig
- 'mit' held 'in' noun -> Dig reverse
- noun 'mit' held -> Dig
- 'in' noun xdamit -> Dig
- xdamit 'in' noun -> Dig reverse
- noun xdamit -> Dig

##Disrobe Klasse 2: Zieht getragene Kleidungsstücke aus.

'nimm' 'nehm' 'hol'

- worn 'ab' -> Disrobe

'entfern'

- worn -> Disrobe

'setz' 'leg'

- worn 'ab' -> Disrobe

'zieh' 'reiss' 'zerr' 'rupf'

- held 'aus' -> Disrobe

'streif'

- noun 'ab' -> Disrobe

##Drink

'trink' 'sauf' 'schluerf' 'schluck'

- noun -> Drink

##Drop Klasse 2: Lässt gehaltene Gegenstände los bzw. fallen.

'entfern'

- held -> Drop

'tu' 'platzier'

- multiheld -> Drop
- multiheld 'weg' -> Drop

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

- multiheld -> Drop
- multiheld 'fallen'/'hier'/'ab'/'weg' -> Drop
- multiheld xhinweg -> Drop

'setz' 'leg'

- multiheld -> Drop
- multiheld 'ab'/'hin'/'weg' -> Drop
- xhinweg multiheld -> Drop
- multiheld xhinweg -> Drop

'stell'

- multiheld 'ab' -> Drop
- multiheld xhinweg -> Drop

'schneid' 'trenn' 'loese' 'spalte' 'zertrenn' 'durchschneid' 'teil' 'zerteil' 'zerschneid'

- 'dich' 'von' multiheld -> Drop

##Eat Klasse 2: Läst essbare Gegenstände verschwinden.

'ess' 'iss' 'friss' 'verspeis' 'verzehr'

* held -> Eat

##Empty Klasse 2: Leert Container aus.

'leer' 'schuett' 'giess' 'entleer'

* noun -> Empty
* noun 'aus' -> Empty

##EmptyT Klasse 2: Füllt Container um.

'leer' 'schuett' 'giess' 'entleer'

* noun 'zu'/'in'/'auf' noun -> EmptyT
* noun xhinein -> EmptyT

'fuell'

* noun 'in' noun -> EmptyT
* noun 'mit' noun -> EmptyT reverse

##Enter Klasse 2: Betritt einen Raum, einen Container oder Supporter.

'nimm' 'nehm' 'hol'

* 'platz' 'auf'/'in' noun -> Enter
* 'auf'/'in' noun 'platz' -> Enter

'pack'

* 'dich' 'auf' noun -> Enter

'steh'

* 'auf' noun -> Enter

'tu' 'platzier'

* 'dich'/'mich' 'auf'/'in' noun -> Enter
* 'dich'/'mich' xdarauf -> Enter
* 'dich'/'mich' 'auf'/'in' noun 'nieder' -> Enter
* 'dich'/'mich' xhinein -> Enter

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

* 'dich' 'auf'/'in' noun -> Enter

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* 'in'/'durch' noun -> Enter
* noun -> Enter

'verlass' 'v//'

* 'in'/'durch' noun -> Enter

'durchquer' 'betret' 'betritt'

* noun -> Enter

'tret' 'tritt'

* 'in' noun -> Enter
* 'in' noun 'hinein'/'ein' -> Enter
* xhinein -> Enter

'setz' 'leg'

* 'dich'/'mich' 'auf'/'in' noun -> Enter

* 'dich'/'mich' 'auf'/'in' noun 'nieder' -> Enter
* 'dich'/'mich' xhinein -> Enter
* xhinein -> Enter

'sitz' 'lieg'

* 'auf' noun -> Enter
* 'in' noun -> Enter
* xdarauf -> Enter

'kletter' 'klettr' 'steig' 'erklimm' 'erkletter'

* 'in' noun -> Enter
* 'in' noun hinein -> Enter
* xhinein -> Enter

##Examine Klasse 2: Gibt die description eines Objekts oder eines Raumes an.

'schau' 'seh' 'l//' 'sieh' 'blick'

* noun 'an' -> Examine
* noun -> Examine
* 'auf'/'durch' noun -> Examine

'untersuch' 'x//' 'b//' 'u//' 'betracht' 'beschreib' 'check' 'begutacht'

* noun -> Examine
* multiexcept -> Examine

'lies' 'les'

* noun -> Examine
* 'in' noun -> Examine

##Exit Klasse 2: Verläßt einen Raum, Container oder Supporter.

'heb'

* 'dich' 'auf' -> Exit

'steh'

* -> Exit
* 'auf' -> Exit

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* 'nach' 'draussen' -> Exit
* 'raus'/'hinaus'/'heraus' -> Exit

'verlass' 'v//'

* noun -> Exit

'tret' 'tritt'

* 'aus'/'heraus'/'hinaus' -> Exit

'raus' 'hinaus' 'heraus' 'a//'

* -> Exit
* 'aus' noun -> Exit

##Fill

'fuell'

* noun -> Fill
* noun 'auf' -> Fill

##GetOff Klasse 2: Verläßt einen Supporter.

'heb'

* 'dich' 'auf' 'aus'/'von' noun -> GetOff
* 'dich' 'aus'/'von' noun -> GetOff

'steh'

* 'auf' 'von'/'aus' noun -> GetOff

'kletter' 'klettr' 'steig' 'erklimm' 'erkletter'

* xheraus -> GetOff
* 'von'/'aus' noun -> GetOff
* 'von'/'aus' noun 'ab'/'runter' -> GetOff

##Give Klasse 2: Gibt einen Gegenstand an jemand anderen.

'gib' 'geb' 'offerier' 'reich' 'uebertrag' 'ueberreich' 'uebergib'

* held creature -> Give
* held 'an' creature -> Give
* 'an' creature held -> Give reverse
* creature held -> Give reverse

'fuetter' 'bezahl' 'zahl'

* creature 'mit' held -> Give reverse
* 'mit' held creature -> Give
* held 'an' creature -> Give
* 'an' creature held -> Give reverse
* creature xdamit -> Give reverse
* xdamit creature -> Give

'biet'

* creature held 'an' -> Give reverse
* held creature 'an' -> Give
* 'fuer' noun -> Buy

##Go Klasse 2: Bewegung in alle Richtungen, insbesondere von einem Raum zum anderen.

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* noun=ADirection -> Go
* 'nach' noun=ADirection -> Go

'verlass' 'v//'

* noun=ADirection -> Go

##GoIn Klasse 2: Betritt einen Raum oder ein `enterable`.

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* 'rein'/'hinein'/'herein' -> GoIn

'durchquer' 'betret' 'betritt'

* -> GoIn

'tret' 'tritt'

* 'ein' -> GoIn

'rein' 'hinein' 'herein'

* -> GoIn
* 'in' noun -> GoIn

##Insert Klasse 2: Lege etwas in einen Container.

'pack'

* multiexcept 'in' noun hinein -> Insert
* multiexcept xhinein -> Insert

'tu' 'platzier'

* multiexcept 'in' noun -> Insert
* multiexcept 'in' noun hinein -> Insert

'steck'

* multiexcept 'in' noun hinein -> Insert
* multiexcept 'in' noun -> Insert
* multiexcept xhinein -> Insert

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

* multiexcept 'in'/'unter' noun -> Insert
* multiexcept 'in' noun hinein -> Insert
* multiexcept xhinein -> Insert

'setz' 'leg'

* multiexcept 'in' noun -> Insert
* multiexcept 'in' noun hinein -> Insert
* multiexcept xhinein -> Insert

'stell'

* multiexcept 'in' noun -> Insert
* multiexcept 'in' noun hinein -> Insert
* multiexcept xhinein -> Insert

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* multiexcept 'in' noun -> Insert
* multiexcept xhinein -> Insert

##Inv Klasse 2: Inventar.

'zeig' 'praesentier' 'fuehr'

* 'inventar'/'besitz'/'eigentum' -> Inv

'inventar' 'inv' 'i//' 'besitz' 'eigentum' 'zeug'

* -> Inv

##InvTall Klasse 2: Inventar.

'inventar' 'inv' 'i//' 'besitz' 'eigentum' 'zeug'

* 'gross'/'lang'/'liste' -> InvTall

##InvWide Klasse 2: Inventar.

'inventar' 'inv' 'i//' 'besitz' 'eigentum' 'zeug'

* 'weit'/'breit'/'satz' -> InvWide

##Jump

'spring' 'huepf'

* -> Jump
* 'hoch'/'herum'/'umher' -> Jump

##JumpOver

'spring' 'huepf'

* 'ueber' noun -> JumpOver

##Kiss

'kuess' 'umarm' 'lieb' 'streichel' 'streichl' 'knutsch' 'umgarn' 'liebkos'

* creature -> Kiss

##Lift verweist standardmäßig auf ##LookUnder.

'heb'

* noun 'an'/'hoch' -> Lift
* 'an'/'hoch' noun -> Lift

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun 'hoch'/'hinauf' -> Lift
* noun 'nach' 'oben' -> Lift

##Listen

'hoer' 'horch' 'lausch'

* -> Listen
* noun -> Listen
* noun 'zu' -> Listen
* 'an' noun -> Listen

##Lock Klasse 2: Verschließe ein abschließbares Objekt.

'schliess' 'sperr'

* noun 'mit' held -> Lock
* noun 'mit' held 'ab' -> Lock
* noun 'ab' -> Lock
* noun 'ab' 'mit' held -> Lock
* noun xdamit -> Lock
* noun xdamit 'ab' -> Lock
* noun 'ab' xdamit -> Lock

'verschliess' 'verriegel' 'verriegl'

* lockable -> Lock
* noun 'mit' held -> Lock
* noun xdamit -> Lock

##Look Klasse 2: Raumbeschreibung.

'schau' 'seh' 'l//' 'sieh' 'blick'

* -> Look
* 'dich'/'mich' 'um' -> Look
* 'herum'/'umher' -> Look

'untersuch' 'x//' 'b//' 'u//' 'betracht' 'beschreib' 'check' 'begutacht'

* -> Look

##LookUnder

'schau' 'seh' 'l//' 'sieh' 'blick'

* 'unter'/'hinter' noun -> LookUnder
* 'unter'/'hinter' noun 'nach' -> LookUnder

'such' 'spuere' 'stoeber' 'wuehl'

* 'unter' noun -> LookUnder

##Mild

'tadel' 'verfluch' 'verdamm' 'verdammt' 'mist' 'jammer' 'winsel' 'schimpf' 'wuete' 'zuerne' 'schiet' 'hader' 'scheibenkl' 'fluch'

* -> Mild

##No

'nein' 'nee' 'noe'

* -> No

##Open Klasse 2: Öffne ein zu öffnendes Objekt.

'mach'

* noun 'auf' -> Open

'oeffn'

* noun -> Open

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun 'auf' -> Open

'dreh' 'rotier' 'schraub'

* noun 'auf' -> Open

'schlag' 'hau'

* noun 'auf' -> Open

##Pray

'bet' 'bitt'

* -> Pray

##Pull

'zieh' 'reiss' 'zerr' 'rupf'

* noun -> Pull
* 'an' noun -> Pull
* noun 'weg' -> Pull

##Push

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun -> Push
* xhinein -> Push
* noun 'weg' -> Push
* 'gegen' noun -> Push

##PushDir

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun noun -> PushDir
* noun 'nach'/'richtung' noun -> PushDir

##PutOn Klasse 2: Lege etwas auf einen Supporter.

'pack'

* multiexcept 'auf' noun -> PutOn
* multiexcept xdarauf -> PutOn

'tu' 'platzier'

* multiexcept 'auf' noun -> PutOn
* multiexcept xdarauf -> PutOn

'steck'

* multiexcept 'auf' noun -> PutOn

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

* multiexcept 'auf'/'ueber' noun -> PutOn

'setz' 'leg'

* multiexcept 'auf' noun -> PutOn

'stell'

* multiexcept 'auf' noun -> PutOn
* multiexcept xdarauf -> PutOn

##Remove Klasse 2: Nimm etwas von einem Supporter oder aus einem Container.

'nimm' 'nehm' 'hol'

* multiinside 'von'/'aus' noun -> Remove
* multiinside 'von'/'aus' noun heraus -> Remove
* multiinside xheraus -> Remove

'greif' 'fass' 'ergreif'

* multiinside 'in' noun -> Remove
* multiinside 'auf' noun -> Remove
* multiinside xheraus -> Remove
* 'nach' multiinside 'in' noun -> Remove
* 'nach' multiinside 'auf' noun -> Remove
* multiinside 'von'/'aus' noun -> Remove
* multiinside 'von'/'aus' noun heraus -> Remove

'heb'

* multiinside xheraus -> Remove

'entfern'

* multiinside 'von'/'aus' noun -> Remove

'zieh' 'reiss' 'zerr' 'rupf'

* noun heraus -> Remove

##Rub

'wisch' 'reinig' 'putz' 'reib' 'schrubb' 'saeuber' 'polier' 'glaette' 'schmirgel' 'buerst'

* noun -> Rub
* noun 'mit' held -> Rub
* noun xdamit -> Rub

##Search Klasse 2: Schaue in einen Container hinein.

'schau' 'seh' 'l//' 'sieh' 'blick'

* 'in'/'aus'/'nach'/'neben' noun -> Search
* 'in'/'auf'/'neben' noun 'nach' -> Search
* 'in' noun hinein -> Search
* xhinein -> Search

'durchsuch' 'durchwuehl' 'durchstoeber'

* noun -> Search

'such' 'spuere' 'stoeber' 'wuehl'

* noun -> Search
* 'in'/'auf'/'ab'/'hinter'/'neben' noun -> Search
* noun 'ab' -> Search

##Set

'stell'

* noun -> Set

##SetTo

'stell'

* noun 'auf' special 'ein' -> SetTo
* noun 'auf' special -> SetTo
* noun 'auf' topic 'ein' -> SetTo
* noun 'auf' topic -> SetTo

'dreh' 'rotier' 'schraub'

* noun 'auf' special -> SetTo

##Show Klasse 2: Zeige einem anderen einen Gegenstand (den du hast),

'zeig' 'praesentier' 'fuehr'

* creature held -> Show reverse
* held creature -> Show
* creature held 'vor' -> Show reverse
* held creature 'vor' -> Show

##Sing

'sing' 'traeller' 'pfeif'

* -> Sing

##Sleep

'schlaf' 'nick' 'schlummer' 'does'

* 'ein' -> Sleep
* -> Sleep

##Smell

'riech' 'schnueffl' 'schnueffel' 'schnupper'

* -> Smell
* noun -> Smell
* 'an' noun -> Smell

##Sorry

'sorry' 'entschuldigung' 'entschuldig' 'verzeih' 'pardon'

* -> Sorry
* 'mir' -> Sorry
* 'mir' 'bitte' -> Sorry
* 'bitte' -> Sorry

##Squeeze

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun 'aus'/'zusammen' -> Squeeze

'zerdrueck' 'quetsch' 'zerquetsch'

* noun -> Squeeze

##Strong

'scheiss' 'kack' 'arsch' 'arschloch' 'wichser' 'wixer' 'piss' 'verpiss'

* -> Strong
* topic -> Strong

##Swim

'schwimm' 'tauch'

* -> Swim

##Swing

'schwing' 'schwenk' 'wedel' 'wedl' 'bauml' 'baumel'

* 'auf'/'an' noun -> Swing
* 'dich'/'mich' 'auf'/'in' noun -> Swing
* xdarauf -> Swing
* dich'/'mich' xdarauf -> Swing

##SwitchOff Klasse 2: Ausschalten eines schaltbaren Gegenstands.

'schalt'

* noun 'aus'/'ab' -> Switchoff
* noun 'aus' -> Switchoff

'mach'

* noun 'aus' -> SwitchOff

'stell'

* switchable 'ab' -> SwitchOff

'dreh' 'rotier' 'schraub'

* noun 'aus' -> Switchoff
* noun 'aus' -> Switchoff

##SwitchOn Klasse 2: Einschalten eines schaltbaren Gegenstands.

'schalt'

* noun -> Switchon
* noun 'ein'/'an' -> Switchon
* 'ein'/'an' noun -> Switchon

'mach'

* noun 'an' -> SwitchOn

'stell'

* noun 'an' -> SwitchOn
* switchable 'ab' -> SwitchOff

'dreh' 'rotier' 'schraub'

* noun 'ein'/'an' -> Switchon
* 'ein'/'an' noun -> Switchon

##Take Klasse 2: Nimm einen Gegenstand.

'nimm' 'nehm' 'hol'

* multi -> Take
* multi 'auf'/'mit' -> Take

'greif' 'fass' 'ergreif'

* multi -> Take
* 'nach' multi -> Take

'pack'

* multi -> Take
* multi 'ein' -> Take
* 'ein' multi -> Take

'heb'

* multi 'auf' -> Take

'entfern'

* multi -> Take

'steck'

* multiexcept 'ein' -> Take

'lies' 'les'

* multi 'auf' -> Take

'zieh' 'reiss' 'zerr' 'rupf'

* noun 'ab' -> Take

##Taste

'schmeck' 'leck' 'kost' 'probier'

* noun -> Taste

##Tell

'red' 'sprech' 'sprich' 'schwatz' 'schwaetz' 'babbel'

* creature 'ueber' topic -> Tell
* 'mit'/'zu' creature 'ueber' topic -> Tell

'erzaehl' 'unterricht' 'bericht' 'erklaer'

* creature 'ueber'/'von' topic -> Tell

'sag'

* creature topic -> Tell
* creature 'ueber' topic -> Tell

##Think

'denk'

* -> Think

##ThrowAt

'lass' 'wirf' 'werf' 'schmeiss' 'schleuder'

* held 'nach'/'gegen'/'auf'/'zu' noun -> ThrowAt
* 'nach'/'gegen'/'auf'/'zu' noun held -> ThrowAt
* held noun 'zu'/'nach' -> ThrowAt

##Tie

'fixier' 'bind' 'befestig' 'knot' 'verknot'

* noun -> Tie
* noun 'an' noun -> Tie

##Touch

'greif' 'fass' 'ergreif'

* 'an' noun -> Touch
* noun 'an' -> Touch

'beruehr' 'fuehl' 'ertast' 'tast' 'befuehl' 'betaste'

* noun -> Touch
* noun 'an' -> Touch
* 'an'/'ueber'/'nach' noun -> Touch

##Transfer Klasse 2: Nimm einen Gegenstand und lege ihn in einen Container oder auf einen Supporter.

'drueck' 'press' 'beweg' 'schieb' 'verschieb'

* noun 'zu' noun -> Transfer

##Turn

'dreh' 'rotier' 'schraub'

* noun -> Turn

##Unlock Klasse 2: Schließe einen verschließbaren Gegenstand auf.

'oeffn'

* 'mit' held noun -> Unlock reverse
* noun 'mit' held -> Unlock
* xdamit noun -> Unlock reverse
* noun xdamit -> Unlock

'schliess' 'sperr'

* noun 'mit' held 'auf' -> Unlock
* noun 'auf' 'mit' held -> Unlock
* noun xdamit 'auf' -> Unlock
* noun 'auf' xdamit -> Unlock

##VagueGo

'pack'

* 'dich' -> VagueGo

'heb'

* 'dich' xhinweg -> VagueGo

'geh' 'lauf' 'renn' 'wander' 'fluechte' 'flieh' 'schreite' 'spazier'

* -> VagueGo
* 'umher'/'weg'/'fort' -> VagueGo

'verlass' 'v//'

* -> VagueGo
* creature -> VagueGo
* 'diesen' 'ort'/'raum'/'platz' -> VagueGo
* 'ort'/'raum'/'platz' -> VagueGo

##Wait

'wart' 'z//' 'harr' 'verharr'

* -> Wait

##Wake

'wach' 'erwach' 'weck' 'erweck'

* -> Wake
* 'auf' -> Wake

##WakeOther

'wach' 'erwach' 'weck' 'erweck'

* creature -> WakeOther
* creature 'auf' -> WakeOther

##Wave

'wink'

* 'mit' noun -> Wave
* xdamit -> Wave

'schwing' 'schwenk' 'wedel' 'wedl' 'bauml' 'baumel'

* noun -> Wave
* 'mit' noun -> Wave
* xdamit -> Wave

##WaveHands

'wink'

* -> WaveHands

'schwing' 'schwenk' 'wedel' 'wedl' 'bauml' 'baumel'

* 'mit' 'der' 'hand' -> WaveHands
* 'mit' 'den' 'haenden' -> WaveHands

##Wear Klasse 2: Ziehe ein Kleidungsstück an.

'trag'

* held -> Wear

'steck'

* clothing 'an' -> Wear
* 'an' clothing -> Wear

'kleid' 'bekleid' 'schmueck'

* 'dich'/'mich' 'an' 'mit' noun -> Wear
* 'dich'/'mich' 'mit' noun 'an' -> Wear
* 'dich'/'mich' 'mit'/'in' noun -> Wear
* 'dich'/'mich' 'an' xdamit -> Wear
* 'dich'/'mich' xdamit 'an' -> Wear
* 'dich'/'mich' xdamit -> Wear

'setz' 'leg'

* held 'an'/'um' -> Wear

'zieh' 'reiss' 'zerr' 'rupf'

* held 'an'/'ueber' -> Wear

##Yes

'ja' 'j//' 'jawohl'

* -> Yes

4.3 implizite Aktionen („*fake actions*")

Die folgenden Aktionen können vom Spieler nicht unmittelbar ausgelöst werden, sondern werden von der Library als Folge anderer Aktionen erzeugt. Aus Sicht der Autorin ergibt sich kein Unterschied zu anderen Aktionen.

##LetGo Wird für einen *container* oder *supporter* ausgelöst, wenn ein Gegenstand aus ihm entfernt wird.

##ListMiscellany und

##Miscellany werden nur für die Ausgabe einer ganzen Reihe verschiedenster Mitteilungen gebraucht.

##NotUnderstood wird an die `Orders`-*property* eines `animate`-Objektes geschickt, wenn an dieses Objekt (diesen NPC) etwas gerichtet wurde, das wie ein Befehl aussieht, aber vom Parser nicht verstanden wird („Troll, troll dich").

##Order Mit dieser Aktion kommt ein vom Spieler an einen NPC gerichteter (und vom Parser verstandener) Befehl an die `life`-*property* des NPC zurück, wenn er in `orders` nicht behandelt wird oder wenn `orders` ein `false` zurückgibt. (Wird der Befehl vom Parser nicht verstanden, wird die Aktion ##Answer versucht.)

##PluralFound Eine `parse_name`-Routine muss die Variable `parser_action` auf diesen Wert setzen, um zurückzumelden, dass eine Pluralbezeichnung für dieses Objekt gefunden wurde (DM[4], p. 214).

##Prompt Druckt den Prompt (normalerweise Zeilenwechsel gefolgt von >). Wer den Prompt ändern will, muss nur die Library message dieser Aktion ändern.

##Receive Wird für einen *container* oder *supporter* ausgelöst, wenn ein Gegenstand hineingetan wird.

##TheSame wird vom Parser intern verwendet, wenn entschieden werden muss, ob zwei Pluralobjekte mit gesetzter `parse_name`-*property* tatsächlich gleich sind (DM[4], p. 213).

##ThrownAt Wird Gegenstand x auf das Ziel y geworfen, so „sieht" Objekt x ein ##ThrowAt mit dem `second` y. Das Zielobjekt Y „sieht" ein ##ThrownAt mit dem `second` x.

5 Zusätze

5.1 Am Anfang war das Wort: Das Zusatzpaket GInfo

GInfo erleichtert die Programmierung der Aktionen `##Consult, ##Ask, ##Tell` und `##Answer.` Die Standardlibrary enthält nur Grammatikzeilen, in denen das indirekte Objekt dieser Aktionen mit dem Token `topic` als unstrukturierter Text erfasst ist. Die Autorin müsste also jedes mögliche Thema der Unterhaltung (oder des Nachschlagens in einem Buch etc.) selbst 'von Hand' parsen. „Aber warum sollte er das tun," dachte sich JESSE BURNEKO, der Autor des originalen Info.h, „wenn er doch schon den guten Parser der Library zur Hand hat?"

GInfo.h definiert einfach alle möglichen Themen der Unterhaltung und Lektüre als Objekte (einer Klasse `Topic`) und prüft, ob sie von einem bestimmten Punkt aus im Blickfeld *(scope)* sind. Dies ist dann der Fall, wenn sie Kinder eines besonderen Objekts mit dem Namen `Topics` sind. Je nach Bedarf kann man sie dort dynamisch ein- und ausklinken (mit `move ... to` und `remove`).

Die Benutzung ist einfach: Nach dem Parser wird der erste Teil der Bibliothek eingebunden, der `Topics` und die Definition der Klasse `Topic` sowie die Definition des Attributs `legible` enthält:

```
Include "GInfo";
```

Die einzelnen Themen werden als Objekte definiert:

```
Topic Tantwort Topics
with  short_name "Antwort",
      name 'Antwort' 'Frage' 'Sinn' 'Leben' 'Universum' 'Rest',
      dekl 3,
has   female;
```

Für Konversation oder Konsultation werden sie vom Parser als `second` übergeben (`noun` ist die Person, mit der gesprochen wird oder das `legible`-Objekt, in dem gelesen werden soll). In der `before`-Routine kann also für jede der oben genannten vier Aktionen ein einfaches `switch(second)` die ganze Arbeit erledigen:

```
Object Heft
has    neuter legible,
with   short_name "Heft",
       adj "klein",
       dekl 1,
       description "Es ist ein kleines Heft ...",
       name 'heft' 'klein' 'handbuch',
       before [;
           Consult: switch (second)
                   {
                     TAntwort: "Natürlich Zweiundvierzig ...".
                     ...
```

Nach `GrammarG.h` muss dann nur noch der zweite Teil von *GInfo* eingebunden werden, der die TopicScope-Routine und umfangreiche Neudefinitionen der Verben für die vier Aktionen enthält:

```
Include "GInfoG";
```

Will man selbst noch Verben zu diesen Aktionen umdefinieren, muss man als Platzhalter für das Thema das Grammatiktoken `scope=TopicScope` verwenden. Für ein Objekt, in dem mit `##Consult` gelesen werden soll, sollte das Attribut `legible` als Grammatiktoken verwendet werden, die Definition eines solchen Objekts *muss* die Klausel `has legible` enthalten.

GInfoG funktioniert auch mit Glulx.

5.2 Ob ich auch wanderte im finstern Tal: automagische Invisiclues

Invisiclues sind Hilfestellungen zur Spiellösung (auch Hinweise, Tipps oder neudeutsch „Hints” genannt), die zu einem bestimmten Rätsel des Spiels mehrere Antworten – von ansteigender Deutlichkeit und Ausführlichkeit – vorsehen, die der Spieler nacheinander aufrufen kann. Als Beispiel die Hilfestellung zu einem der ersten Rätsel eines Klassikers[1]:

Wie komme ich in die Höhle?

Der Eingang ist im Haus.
Falltüren können versteckt liegen.
Schau' unter den Teppich.

[1] aus: Infocom, Zork 1

Slag und Inform Es ist möglich, dies mit einer der Bibliotheken `menus.h`, `hints.h`, `branch.h` zu schreiben.[2] So richtig automagisch funktioniert es aber erst mit dem Codegenerator `slag`, einem Perlscript[3] von BRIAN 'BEEJ' HALL, in der Release 8 mit Beispieldateien (Lösungen einiger Infocom-Adventures) zu finden bei `http://www.piratehaven.org/~beej/slag/` oder `http://if-archive/solutions/slag`. Hier erstellen wir in einem ersten Schritt eine Quelldatei mit den Hints, die wir haben wollen; hier mit einem vorgeschalteten, zwei Ebenen umfassenden Menüsystem:

```
.MENU Zork I Invisiclues
.LINK Über der Erde
.LINK Im Keller
.LINK Der Irrgarten
.LINK Der Runde Raum

.MENU Über der Erde
.LINK Wo finde ich ein Machete?
.LINK Wie überquere ich das Gebirge?
.LINK Wie komme ich in die Höhle?
.LINK Was ist ein Graus?

.HINT Wie komme ich in die Höhle?
.CLUE Der Eingang ist im Haus.
.CLUE Falltüren können versteckt liegen.
.CLUE Schau' unter den Teppich.

.HINT Was ist ein Graus?
.CLUE Frage ZORK I.
```

Beachte: Der Text jeder `.LINK`-Zeile kommt eine Ebene tiefer identisch nochmal vor – als `.MENU` oder als `.HINT`. So wird die Menüstruktur aufgebaut.

Anschließend „übersetzen" wir mit dem Perlscript[4] `slag` die Quelldatei in `inform`-Code:

```
slag -s QUELLDATEI ZIELDATEI.inf
```

und binden diese in unser Spiel ein mit der Direktive

```
Include ">ZIELDATEI";
```

[2] alle zu finden in `http://www.ifarchive.org/indexes/if-archiveXinfocomXcompilersXinform6XlibraryXcontributions.html`

[3] Active Perl für Windows bei `http://www.activestate.com/Products/ActivePerl/`

[4] Dies setzt eine normale Installation von perl unter Unix als /usr/bin/perl voraus. Windows-Benutzer können z.B. die Endung .pl für Perl registrieren, slag in slag.pl umbenennen und auch so aufrufen.

Jetzt müssen wir dafür sorgen, dass der Spieler die Hints auch aufrufen kann: Wir müssen ihm dazu nur die Funktion `SLAG_RunMenus();` zugänglich machen. Der einfachste Weg geht über eine eigene Aktion und neue Verben:

```
[HintSub;
    SLAG_RunMenus();
];

verb meta 'hinweis' 'hilf' 'hint'
    * -> Hint;
```

Für eine alternative Art der Einbindung siehe das Heft in „Eden".

Weitere Möglichkeiten Das slag-Format ist nicht frei; jeder Hinweis muss auf eine Zeile passen. Dafür sind die aus Inform bekannten `^` und `~` - Zeichen einsetzbar.

Wir kennen bis jetzt die slag-Direktiven `.MENU, .LINK, .HINT` und `.CLUE`. Statt der abschnittweisen Preisgabe von Information mit `.HINT` und `.CLUE` kann auch ein zusammenhängender, über mehrere Zeilen reichender Text gezeigt werden, der zwischen die Direktiven `.TEXT` und `.ENDTEXT` zu stellen ist. `.TEXT` entspricht dabei der Zeile `.HINT`, d.h. der in dieser Zeile nachfolgende Text muss mit einem `.LINK` übereinstimmen. Zwischen `.TEXT` und `.ENDTEXT` können einzelne Zeilen mit `.CENTER` zentriert werden und ganze Bereiche können zwischen `.FIX` und `.UNFIX` gestellt und so vorformatiert mit fixem Zeichensatz ausgegeben werden.

Ohne den Schalter `-s` baut `slag` eine selbständig kompilierbare Inform-Quelldatei, in der dann nur die Hinweismenüs drin sind. Mit den Direktiven `.INTRO` und `.ENDINTRO` – die sonst wie `.TEXT` und `.ENDTEXT` funktionieren – kann für diesen Zweck ein Vorspann geschrieben werden.

Die Menüzeile ist aber englisch! Kein Problem. Ein paar kleine Änderungen in `slag`, dann ist alles deutsch; zu diesem Zweck wird von mir mit der Datei `Gslag.patch` ein Patch[5] bereitgestellt:

```
patch slag Gslag.patch
```

slag funktioniert – wegen der grundverschiedenen Bildschirmsteuerung – leider nicht mit Glulx.

[5] Benötigt wird das Programm patch, für Windows zu finden bei `http://www.cygwin.com/packages/patch/`

Danksagungen

GRAHAM NELSON für Inform, die originale Library und das „Inform Designers Manual”.

ROGER FIRTH für seine englische Kurzreferenz (und damit der Idee für dieses Machwerk).

TONI ARNOLD, RALF HERRMANN und MAX KALUS für die deutsche Übersetzung der Library.

FLORIAN EDLBAUER und MARTIN OEHM für Korrekturen.

JESSE BURNEKO für Info.h.

BRIAN ’BEEJ’ HALL für slag.

ELLEN SCHRAMM, die alle meine *anderen* Texte tippt.

Erstellt wurde dieser Text mit dem LaTeX-Frontend Lyx und dem KomaSkript-Paket unter GNU/Linux.

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under theterms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include

translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any

> later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Schlussbemerkung:

Bei der Entwicklung und Herstellung dieses Texts wurden keine Tierversuche durchgeführt und auch Produkte von M$ wurden nicht eingesetzt.

Die ursprüngliche Fassung des „Abentheurlichen Informissimus Teutsch” stammt von Frank Borger (2002).

Index