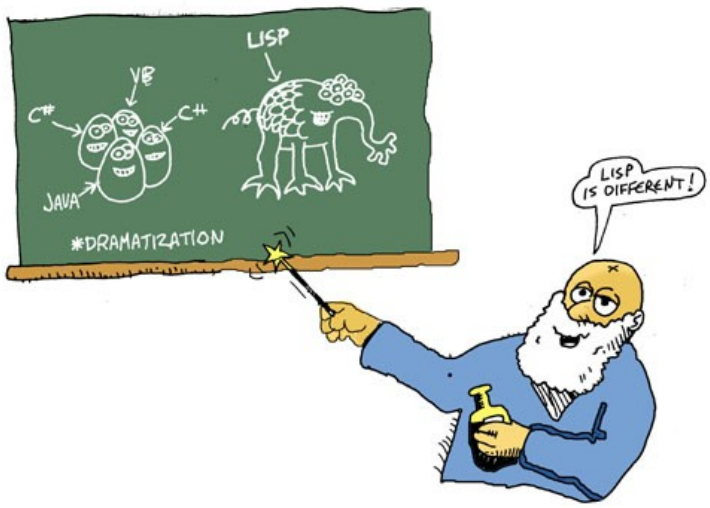
Ein COMIC BUCH
Von Conrad Barski, M.D.
LISPERATI.COM

Dieses **Maxima Tutorial** ist eine recht freie Übersetzung des Text Adventures **Casting SPELs** ins Deutsche. Casting SPELs ist ursprünglich ein Lisp Tutorial. Die Programmiersprache wurde also auch übersetzt. Verantwortlich hierfür ist Volker van Nek. Autor und Übersetzer stellen dieses Tutorial unter den Bedingungen der GNU General Public License jedem zur freien Verfügung.

Die beiden Personen im obigen Bild sind die Helden unserer Geschichte. Da gibt es diesen Zauberer mit der Whiskeyflasche und den Typ mit der Brille. Der bist Du. Und für Dich gibt es, wie in jedem guten Abenteuer, eine Aufgabe zu erledigen. Am Ende winkt ein `magic low_carb donut` als Belohnung! Ok, legen wir los.

Chapter_1: "Ever wonder what makes Maxima so powerful? Now you can find out for yourself" $

Maxima ist in **Lisp** geschrieben. Das sagt eigentlich schon alles.



Maxima steht also, um ein Zitat von Isaac Newton zu gebrauchen, auf der Schulter eines Riesen.

Eigentlich ist es ja ein Computer Algebra System. Es wurde erfunden, um die verschiedensten mathematischen Aufgaben zu erledigen. Integrale bestimmen, Gleichungen lösen, Graphen plotten und all diese Dinge. Arbeite damit und Du wirst sehr schnell feststellen, dass Maxima zusätzlich über eine wirklich wunderbare Programmiersprache verfügt. Die **Syntax**, das heißt, die Art und Weise, wie Du Deine Ideen aufschreiben musst, ist erstaunlich einfach. Gleichzeitig ist diese Sprache enorm flexibel und kraftvoll, und das soll in diesem Tutorial unter Beweis gestellt werden.

Chapter_2: "Syntax and Semantics" $

Die **Semantik** ist der Inhalt Deines Programms, also eher Zufälliges, während die Syntax in jeder Programmiersprache alles andere als zufällig ist, sondern genau festgelegt sein muss. Wie gesagt, die Syntax beschreibt, **wie** Du Dich mit dem Compiler verständigen kannst, damit der dann für Dich Dein Programm in die Maschinensprache des Computers übersetzen kann.

Grundlegend für die Maxima Syntax ist folgendes: Alles, was Du in Maxima bis zum abschließenden Semikolon eingibst, ist ein **Ausdruck**. Und jeder Ausdruck besitzt einen **Wert**, den Du dann als Antwort erhältst. Du musst also keine print-Ausgabe schreiben.

```
'whiskey_bottle;
==> whiskey_bottle
```

Die Ausdrücke `in dieser Schrift und Farbe` sind Maxima-Ausdrücke, die Du einfach in eine Eingabezeile des Maxima-Editors kopieren kannst. Enter drücken und los geht es, Dein Text Adventure.

Die grundlegendsten Ausdrücke sind **Atome**. Atome, das klingt schon sehr elementar, sind **Symbole** (Namen), Zahlen und Text in Anführungszeichen. Die beiden letzteren haben sich selbst als Wert. Bei Symbolen kommt es darauf an, in welchem Modus der Compiler Deinen Ausdruck liest. Es gibt den **Programm-** und den **Daten-Modus**.

Durch das alles entscheidende **Hochkomma** wurde unser Symbol `whiskey_bottle` im Daten-Modus gelesen und hat damit seinen Namen als Wert. Lassen wir das Hochkomma weg, befinden wir uns im Programm-Modus.

```
whiskey_bottle;
==> whiskey_bottle
```

Schon wieder `whiskey_bottle`! Wozu dann die Unterscheidung? `whiskey_bottle` hat immer noch seinen Namen als Wert. Antwort: Im Programm-Modus können wir unserem Symbol einen anderen Wert zuweisen.

```
whiskey_bottle: 'full;
==> full
```

Mit dem **Doppelpunkt** bewirken wir eine Zuweisung an das davor stehende Symbol. Danach hat das Symbol `whiskey_bottle` im Programm-Modus einen geänderten Wert. Den haben wir dann auch als Antwort erhalten. Ab jetzt müssen wir das Hochkomma verwenden, wenn wir die Antwort `whiskey_bottle` erhalten wollen. Oder wir löschen die Zuweisung. Das geht mit `kill`.

```
kill(whiskey_bottle);
==> done
```

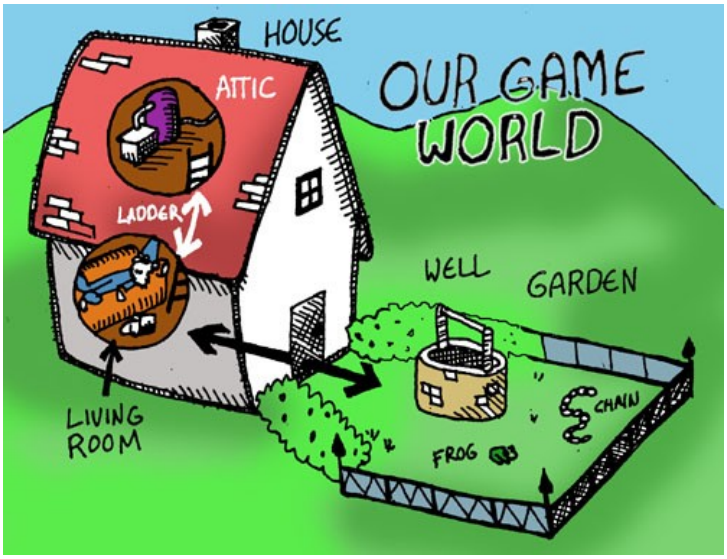
Chapter_3: "Defining the Data for our Game World" $

In unserem Spiel gibt es neben den beiden Personen noch vier bewegliche Objekte, mit denen der Spieler irgend etwas machen kann. Die Whiskeyflasche des Zauberers kennen wir schon. Dann werden wir bald noch einem Frosch, einer Kette und einem Eimer begegnen. Diese vier Objekte packen wir in eine **Liste** und speichern die Liste unter dem Namen `objects`.

```
objects: '[whiskey_bottle, bucket, chain, frog]$
```

Die eckigen Klammern werden in Maxima für Listen verwendet. Eine Liste ist offensichtlich kein Atom, sondern ein aus mehreren Elementen zusammengesetzter Ausdruck. Da Maxima auf Lisp (LISt Processing) aufbaut, ermöglicht die Verwendung von Listen eine sehr flexible und elegante Programmierung. Davon werden wir bald reichlich Gebrauch machen.

Die Antwort wurde hier mit dem **Dollarzeichen** unterdrückt. Die kennst Du doch nun schon. Oder?



Hier ist sie nun, unsere kleine Welt, in der das Spiel stattfindet.

Sie besteht aus drei verschiedenen Orten. Das Wohnzimmer und der Dachboden des Hauses und der Garten.

Wir definieren als erstes das Symbol `map`, das eine Menge an Informationen über unser Spielfeld enthält.

```
map: '[[living_room,
          "you are in the living_room of a wizards house. there is a wizard
snoring loudly on the couch. ",
          [west, door, garden],
          [upstairs, stairway, attic] ],

       [garden,
          "you are in a beautiful garden. there is a well in front of you. ",
          [east, door, living_room] ],

       [attic,
          "you are in the attic of the wizards house. there is a giant welding
torch in the corner. ",
          [downstairs, stairway, living_room] ]]$
```

`map` ist eine Liste aus drei Listen. Der erste Eintrag ist jeweils der Name des Ortes und der zweite beschreibt diesen Ort in zwei Sätzen. Die weiteren Einträge sind wieder Listen, die in Stichworten enthalten, in welche Richtung und wie man in einen anderen Raum kommt.

Die Whiskeyflasche befindet sich, falls er nicht gerade seinen Rausch ausschläft, stets in der Hand des Zauberers.

Die Kette und der Frosch liegen im Garten und den Eimer finden wir im Wohnzimmer. Diese Informationen sind in `object_locations` gespeichert.

```
object_locations: '[[whiskey_bottle, living_room],
                    [bucket, living_room],
                    [chain, garden],
                    [frog, garden]]$
```

Jedes Objekt ist mit einem Ort **assoziiert**. Eine Liste, die wie `object_locations` aus mehreren Listen besteht, nennt man eine assoziierte Liste. Der jeweils erste Listeneintrag ist der Schlüssel, über den wir auf den Rest der Liste zugreifen können. Das werden wir bald sehen. `map` ist übrigens auch eine assoziierte Liste.

Ok, wir haben jetzt das Spielfeld und die Spielfiguren definiert. Fehlt nur noch, dass wir festlegen, wie es los gehen soll.

```
location: 'living_room$
```

Wir befinden uns also im Wohnzimmer des Zauberers.

Chapter_4: "Looking Around in our Game World" $

Wir wollen nun in unserer Spielwelt agieren und definieren dazu ein paar nützliche Kommandos. Das erste, was wir benötigen, sind Informationen über den Ort, an dem wir uns gerade befinden. Das Kommando `describe_location`, das uns die gewünschte Antwort geben soll, muss dazu natürlich einen Einblick in das gesamte Spielfeld haben und soll dann die unserem Ort entsprechende Beschreibung heraussuchen.

```
describe_location(location,map):= second( assoc_(location,map) )$
```

`describe_location` erhält einen Ort und ein Spielfeld als Eingabeparameter und gibt dann die genau passende Beschreibung des Ortes als Antwort. `describe_location` verhält sich also wie eine mathematische **Funktion**. Und solche Funktionen machen nichts weiter, als einfach nur Funktionswerte als Antwort zurückzugeben. Wir haben damit unsere erste Funktion geschrieben. Sehen wir uns mal genauer an, wie sie arbeitet.

Über assoziierte Listen haben wir schon gesprochen. Der Ausdruck `assoc_(location,map)` gibt uns die Liste aus dem Spielplan als Antwort, die den betreffenden Ort als ersten Eintrag besitzt. Wenn wir noch einmal kurz zurückscrollen und uns `map` anschauen, dann sehen wir, dass die Ortsbeschreibung der jeweils zweite Listeneintrag ist. Um den zu erhalten, wenden wir dann auf das Ergebnis von `assoc_` noch die Funktion `second` an. Auf diese Weise erhalten wir einen verschachtelten Ausdruck, der dann von innen nach außen ausgewertet wird. Das ist eine aus der elementaren Mathematik bekannte Vereinbarung.

Bevor wir nun den Ausdruck `assoc_(location,map)` in Maxima eingeben können, müssen wir die Funktion `assoc_` noch schnell eben selbst schreiben. Die in der Bibliothek von Maxima vorhandene Funktion `assoc` ist nur in der Lage, assoziierte Listen wie `object_locations` zu verarbeiten. Mehr als zwei Einträge pro Liste sind da nicht erlaubt. Hier ist nun eine für unser Spiel passende Version.

```
assoc_(key,alist):= block(
   [ result:false ],
   for elem in alist do
      if key=first(elem) then return(result:elem),
   result )$
```

Hier sehen wir nun, wie in Maxima Funktionen mit mehreren aufeinander folgenden Anweisungen geschrieben werden. Mit Hilfe der Funktion `block` definieren wir einen **Programmblock**, dessen Wert sein letzter Ausdruck ist, in diesem Falle also `result`. Das erste Argument in `block` ist eine Liste mit **lokalen Variablen**. Der Name `result` ist auf diese Weise nur innerhalb des Blocks von Bedeutung und es entsteht kein Konflikt mit irgendeinem eventuell außerhalb der Funktion `assoc_` vorhandenem Symbol mit gleichlautendem Namen.

`do` ist in Maxima ein spezieller Operator und bewirkt eine **Programmschleife**. Durch `for elem in alist` wird eine lokale Schleifenvariable `elem` definiert, die dann während der Schleife die Liste `alist` durchläuft. Die Anweisungen, die dem `do` folgen, werden bis zum Ende der Schleife ausgeführt, es sei denn, die Schleife wird durch ein `return` abgebrochen. Das ist der Fall, wenn der erste Eintrag in `elem` mit `key` übereinstimmt. Der Wert einer Schleife ist entweder `'done` oder im Falle des Abbruchs das Argument von `return`. Beides wird hier jedoch nicht verwendet. Entscheidend ist nur, dass vor dem Abbruch der Schleife noch schnell die Zuweisung `result:elem` stattfindet.

Jetzt sollten wir doch aber endlich mal unsere Funktion `describe_location` testen.

```
describe_location(location,map);
==> you are in the living_room of a wizards house. there is a wizard snoring
loudly on the couch.
```

Perfekt! Genau das wollten wir.

Bevor wir jetzt mit unserem Spiel weiter machen, müssen wir an dieser Stelle unbedingt kurz innehalten und uns über den Gültigkeitsbereich der verwendeten Symbole klar werden. Die beiden Symbole `location` und `map`, die die Informationen über unser Spiel enthalten, sind **globale Variablen**, wir haben sie im sogenannten **Toplevel** definiert. Diese Variablen sind überall sichtbar und abrufbar.
Dagegen sind die beiden Symbole `location` und `map`, die wir in der Funktionsdefinition von `describe_location` verwendet haben, *lokal*e Variablen und haben mit den globalen Variablen gleichen Namens eigentlich nichts zu tun. Wir hätten `describe_location` auch zum Beispiel durch `describe_location(l,m):= second( assoc_(l,m) )$` definieren können. Mit dem Vorteil, dass globale und lokale Variablen unterschiedliche Symbole verwenden, aber auch mit dem Nachteil, dass die Symbole `l` und `m` nicht gerade sprechend sind.

Beim anschließenden Aufruf der Funktion durch `describe_location(location,map);` haben wir dann die beiden *global*en Variablen als Argumente verwendet. Das globale `location` hat momentan den Wert `'living_room` und `describe_location` sucht dann in `map` nach der passenden Beschreibung des Wohnzimmers.

Etwas anderes noch. `describe_location():= second( assoc_(location,map) )$` mit anschließender Eingabe von `describe_location();` hätte zum gleichen Ergebnis geführt. In dieser Form hat `describe_location` keine Argumente und wertet dann direkt die beiden globalen Variablen aus. Eine solche Definition würde jedoch nicht der **Funktionale**n **Programmierung** entsprechen, einem Konzept, das zum Ziel hat, Funktionen möglichst in der folgenden Form zu schreiben.

1. Nur die Funktionsargumente und die Variablen, die lokal in der Funktion selbst definiert sind, werden ausgewertet. Globale Variablen werden nicht gelesen.
2. Der Wert einer einmal gesetzten Variablen wird nicht mehr geändert.
3. Die Interaktion mit der restlichen Welt bleibt für eine Funktion darauf beschränkt, Funktionsparameter anzunehmen und einen Wert zurück zu geben.

Da fragt man sich schnell, ob man mit diesen extremen Einschränkungen tatsächlich nützliche Programme schreiben kann. Die Antwort ist Ja, wenn man sich einmal an diesen Stil gewöhnt hat. Der entscheidende Grund für dieses Konzept ist, dass man auf diese Weise eine sogenannte **referenzielle Transparenz** erhält, was bedeutet praktisch, dass man sich absolut darauf verlassen kann, dass eine Funktion bei denselben Eingabeparametern stets dieselben Antworten erzeugt. Viele Programmierfehler lassen sich so vermeiden.

Natürlich können nicht immer alle Funktionen in diesem Sinne funktional sein. Sonst könnte man ja zum Beispiel auch keinen Kontakt mit der restlichen Welt aufnehmen. So werden denn in diesem Tutorial nicht alle Funktionen diesen Regeln folgen.

Spielen wir nun wieder unser Spiel. Wir benötigen jetzt noch eine Funktion, die uns verrät, in welche Richtung und wie wir von unserem aktuellen Standpunkt aus in einen anderen Raum kommen können. Dazu definieren wir erst einmal eine Funktion, die aus den Wegbeschreibungen, die in `map` enthalten sind, lesbare Sätze formt.

```
describe_path(path):=
   sconcat("there is a ", path[2], " going ", path[1], " from here. ")$
```

`path[n]` bezeichnet dabei das n-te Element der Liste `path`. Diese Kurzschreibweise ist oft recht praktisch. `map[1][4][3]` ergibt zum Beispiel `'attic`.
Die Funktion `sconcat` fügt dann alle Argumente zu einem String zusammen und wir erhalten einen ordentlichen Satz in englischer Sprache.

```
describe_path('[west, door, garden]);
==> there is a door going west from here.
```

Die Information, dass dieser Weg in den Garten führt, werden wir erst dann verwenden, wenn wir uns auch tatsächlich für diesen Weg entscheiden. In `describe_path` mussten wir noch selbst eine passende Liste einsetzen, jetzt aber werden wir analog zu `describe_location` eine Funktion definieren, die alle von unserem aktuellen Standpunkt aus möglichen Wege beschreibt. `describe_paths` erhält dann ebenfalls die globalen Variablen `location` und `map` als Argumente.

```
describe_paths(location,map):=
   apply( sconcat, map( describe_path,rest( assoc_(location,map),2 ) ) )$
```

Diese Funktion ist sehr verschachtelt. Wir beginnen ganz innen. `assoc_(location,map)` kennen wir schon und `rest(list,2)` entfernt aus der Liste, die wir von `assoc_` erhalten, die ersten beiden Elemente. Übrig bleibt eine Liste mit Wegbeschreibungen. Damit das nicht trockene Theorie bleibt, solltest Du das zur Übung mal praktisch ausprobieren. Es ist ja gerade einer der Vorteile, mit einer interpretierten Sprache zu arbeiten, dass man einzelne Programmschritte schnell interaktiv testen kann.

Befinden wir uns im Wohnzimmer, enthält die von `rest` zurückgegebene Liste zwei Wegbeschreibungen, in den beiden anderen Fällen nur eine. Wir benötigen jetzt eine Funktion, die die Hilfsfunktion `describe_path` auf jeden Weg in dieser Liste anwendet. Genau das macht für uns die Funktion `map`. Wir begegnen hier zum ersten mal einer **Funktion höherer Ordnung**. `map` ist eine Funktion, die eine andere Funktion als Argument besitzt. Genauso hat `apply` hier die Funktion `sconcat` als Argument. `apply` sorgt hier dafür, dass `sconcat` alle Elemente der von `map` zurückgegebenen Liste als Argumente erhält.

```
describe_paths(location,map);
==> there is a door going west from here. there is a stairway going upstairs
from here.
```

Wunderbar!

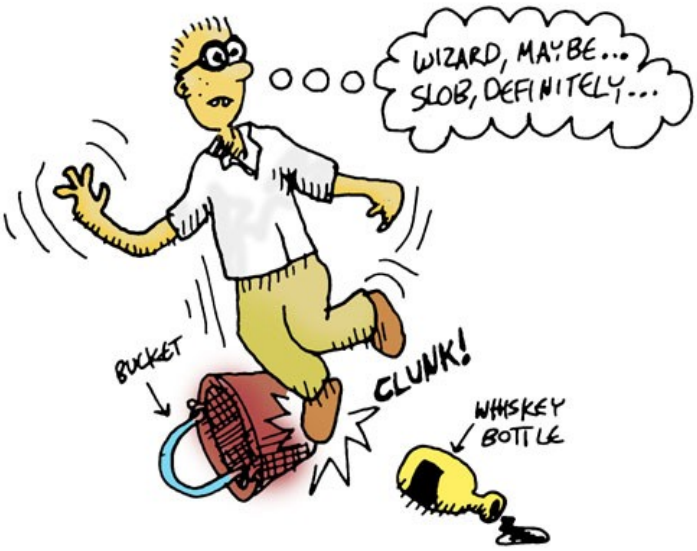
Wir müssen jetzt nur noch beschreiben, ob da, wo wir uns gerade befinden, irgendwelche Objekte auf dem Boden herumliegen. Zuerst schreiben wir die Hilfsfunktion `is_at` die uns sagt, ob sich ein bestimmtes Objekt an einem bestimmten Ort befindet.

```
is_at(obj,loc,obj_loc):= block(
   [ tmp:assoc_(obj,obj_loc) ],
   listp(tmp) and is(second(tmp)=loc) )$
```

Der Wert von `assoc_` ist entweder `false` oder eine Liste, und das heißt, dass wir, bevor wir diesen Wert an `second` weiterreichen, erst einmal testen müssen, ob es sich überhaupt um eine Liste handelt. `listp` übernimmt das für uns. Sollte `listp false` ergeben, blockiert dann das logische `and` den Rest der Zeile. Nur in dem Fall, dass `assoc_` uns eine Liste zurück gibt, wertet `is` dann die Gleichung aus.

Ok, probieren wir es mal.

```
is_at('bucket,'living_room,object_locations);
==> true
```



Offensichtlich! Der Eimer befindet sich im Wohnzimmer.

Wir verwenden nun `is_at`, um alles zu beschreiben, was auf dem Boden herumliegt.

```
describe_floor(loc,objs,obj_loc):=
   apply( sconcat,
          map( lambda([x],sconcat("you see a ",x," on the floor. ")),
               sublist( objs, lambda([x],is_at(x,loc,obj_loc)) ) ))$
```

Wir sehen uns erst einmal das Ergebnis an.

```
describe_floor('living_room,objects,object_locations);
==> you see a whiskey-bottle on the floor. you see a bucket on the floor.
```

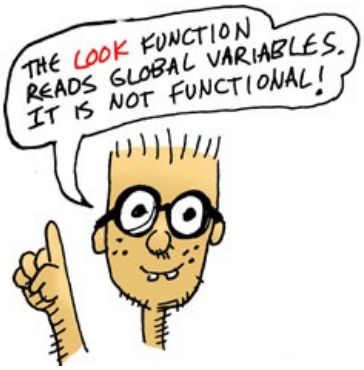
Hier tauchen einige neue Dinge gleichzeitig auf. Wer gerade nicht präsent hat, was in `objects` und `object_locations` gespeichert ist, sollte vielleicht noch mal eben kurz zurückscrollen.

`lambda` ist die sogenannte **anonyme Funktion**. Mit ihr wird an Ort und Stelle eine Funktion definiert und verwendet. Wir hätten auch genau so gut außerhalb von `describe_floor` eine Funktion `blabla(x):= sconcat("you see a ",x," on the floor. ")$` definieren können. Die dritte Zeile hätte dann `map( blabla,` gelautet. `map` wendet `lambda` auf die vierte Zeile an. Das erste Argument von `lambda` ist eine Liste mit den Variablen der anonymen Funktion, in diesem Fall `x`.

Die vierte Zeile ergibt eine Liste mit den Objekten, die in `loc` auf dem Boden liegen. `sublist` macht das für uns und verwendet dabei eine Testfunktion, die auch wieder als `lambda` formuliert wurde. `sublist` ist wie `map` eine Listenfunktion höherer Ordnung.

Wir fügen nun unsere drei Beschreibungsfunktionen zu einem einzigen Kommando zusammen.

```
l_o_o_k():=
   sconcat( describe_location(location,map),
            describe_paths(location,map),
            describe_floor(location,objects,object_locations) )$
```



Dieses ist nun definitiv keine Funktion im Sinne der Funktionalen Programmierung. `l_o_o_k` besitzt keine Inputparameter und liest globale Variablen. Die Antwort wird unterschiedlich ausfallen, je nach dem, wo wir uns gerade befinden. Soll ja auch.

Wir könnten nun dieses Kommando mit `l_o_o_k();` aufrufen. Wir wenden nun aber noch einen kleinen Trick an, so dass wir einfach nur `look;` eingeben müssen.

Maxima bietet uns die Möglichkeit, **Operatoren** zu definieren. Wir definieren die Zeichenkombination `look` als Operator ohne Argument.

```
nofix("look")$
"look"():= l_o_o_k()$
```

Wir sind so weit.

```
look;
==> you are in the living-room of a wizard's house. there is a wizard snoring
loudly on the couch. there is a door going west from here. there is a stairway
going upstairs from here. you see a whiskey-bottle on the floor. you see a
bucket on the floor.
```

Echt cool!

Wir wollen uns nun in unserer Welt bewegen, andere Orte erkunden und auch ein paar Aktionen durchführen. Am Ende winkt ein `magic low_carb donut` als Belohnung!

Fortsetzung in **Casting SPELs** Teil II.

"Aufgaben"$

1. Schreibe ein Kommando `casting SPELs`, das bewirkt, dass der Aufenthaltsort des Spielers und der Objekte Eimer, Kette und Whiskeyflasche zufällig gesetzt werden. Der Frosch soll im Garten bleiben, da wir sonst Ärger mit dem Zauberer bekommen. Die Rückgabe soll eine einfache Textausgabe sein.

```
casting SPELs;
==> have a lot of fun.
```

Tipp: `random(n)` erzeugt eine Zufallszahl zwischen `0` und `n-1` einschließlich.
Mit `set_random_state( make_random_state(true) )` kann ab der Maxima Version 5.9.2 der Zustand des Zufallsgenerators unter Verwendung der Systemzeit zufällig gesetzt werden.

"Danksagung"$

Ein besonderer Dank gilt Conrad Barski für die geniale Idee des Spiels und die wunderbaren Bilder.

Volker van Nek, Aachen, Januar 2006

"Lizenz"$