Chapter_5: "Walking Around In Our World" $

In unserer Welt zu gehen, bedeutet einen anderen Ort aufzusuchen. Die Funktion `w_a_l_k__direction`, die eine Wegrichtung als Argument annimmt, verändert dann entsprechend den Ort, an dem wir uns befinden, das heißt, die globale Variable `location`. Auch dies entspricht demnach nicht dem funktionalen Stil.

```
w_a_l_k__direction(direction):= block(
  [ next_: assoc_( direction,rest( assoc_(location,map), 2 ) ) ],
  if listp(next_) then (
     location: third (next_),
     l_o_o_k() )
  else "you cant go that way. ")$
```

`assoc_` hilft uns hier wieder, aus dem gesamten Spielfeld die passende Wegbeschreibung herauszusuchen. Gibt es einen zu der gewählten Richtung passenden Weg, finden wir den neuen Ort als dritten Eintrag in der Liste `next_`, andernfalls geben wir eine entsprechende Rückmeldung aus. Das Symbol `next` ist in Maxima reserviert, deshalb wurde hier bei der Übersetzung in Maxima der Unterstrich angefügt. Mit `next` kann man beschreiben, wie eine durch `for` definierte Schleifenvariable verändert werden soll.

Sind wir an einem neuen Ort erfolgreich angekommen, sehen wir uns mit `l_o_o_k()` erst einmal um. Mit `w_a_l_k__direction(direction)` gehen wir dann weiter. Zur einfacheren Eingabe definieren wir in diesem Fall einen Präfix-Operator.

```
prefix("walk")$
"walk"(direction):= w_a_l_k__direction(direction)$
```

Wir verwenden hierbei die konkrete Wegrichtung als Argument. Das sieht dann so aus.

```
walk west;
==> you are in a beautiful garden. there is a well in front of you. there is a
door going east from here. you see a chain on the floor. you see a frog on the
floor.
```

Eigentlich hätten wir ja `'west` verwenden müssen. Da aber dem Symbol `west` in unserem Spiel nichts zugewiesen wird, können wir das Hochkomma ruhig weg lassen.

Wir erzeugen nun ein Kommando, das uns erlaubt, Gegenstände vom Boden aufzuheben.

```
prefix("pickup")$
"pickup"(object):=
   if is_at( object,location,object_locations ) then (
      object_locations: cons([object,'body],object_locations),
      sconcat("you are now carrying the ",object,".") )
   else "you cannot get that. "$
```

Wir lernen hier eigentlich nichts Neues. Anders als bei `walk` haben wir hier jedoch die Funktionsdefinition gleich in die Definition des Operators gesteckt. Mit `'body` wurde hier der Liste `object_locations` ein vierter Ort hinzugefügt.

Nehmen wir doch mal die Kette.

```
pickup chain;
==> you are now carrying the chain.
```

Damit wir uns das nicht merken müssen, benötigen wir ein Kommando, dass uns jederzeit Auskunft darüber gibt, welche Gegenstände wir mit uns herumtragen. Dazu definieren wir eine Funktion, durch die wir eine Liste dieser Gegenstände erhalten.

```
at_body():=
   sublist( objects,lambda([x],is_at(x,'body,object_locations)) )$
```

Das Kommando `inventory;` schreibt diese Liste als Text ohne Listenklammer.

```
nofix("inventory")$
"inventory"():=
   apply( sconcat,
      map( lambda([x],sconcat(x," ")),
           at_body() ))$
```

Für das nächste Kapitel stellen wir noch eine Funktion bereit, die sagt, ob wir einen bestimmten Gegenstand mit uns herumtragen. Wir verwenden hierbei die Listenfunktion `member`, die ermittelt, ob ein bestimmtes Objekt Element einer Liste ist.

```
have(object):=
   member( object,at_body() )$
```

Chapter_6: "Creating Special Actions in Our Game" $

Wir fügen dem Spiel nun einige spezielle Aktionen hinzu, die der Spieler, das bist Du, ausführen muss, um das Spiel zu gewinnen. Mit dem ersten Kommando wird die Kette an den Eimer geschmiedet. Diese Aktion findet auf dem Dachboden des Hauses statt. Sinn macht das natürlich nur, wenn das nicht schon bereits erledigt ist. Deswegen wird als erstes eine neue globale Variable definiert, die festhält, ob diese Aktion schon stattgefunden hat oder nicht. Auch sollte der Spieler die Kette und den Eimer im Gepäck haben. Die Antwort des Kommandos `weld` ist also an viele Bedingungen gebunden. Sind alle Voraussetzungen erfüllt, wird geschmiedet, das heißt, die globale Variable `chain_welded` wird auf `true` gesetzt.

```
chain_welded: false$

infix("weld")$
"weld"(subject,object):=
  if location='attic
     and subject='chain
     and object='bucket
     and have('chain) and have('bucket)
     and not chain_welded then (
        chain_welded: true,
        "the chain is now securely welded to the bucket. ")
  else  "you cannot weld like that. "$
```

Da in diesem Fall zwei Argumente verwendet werden, wurde das Kommando als **Infix-Operator** definiert. An dieser Stelle soll jetzt keine Diskussion darüber losgetreten werden, ob die Begriffe `subject` und `object` grammatikalisch richtig gewählt wurden. Es bleibt jedenfalls zu hoffen, dass der geneigte Leser hier über die sprachlich nicht wirklich elegante Formulierung des Kommandos hinwegsehen kann.

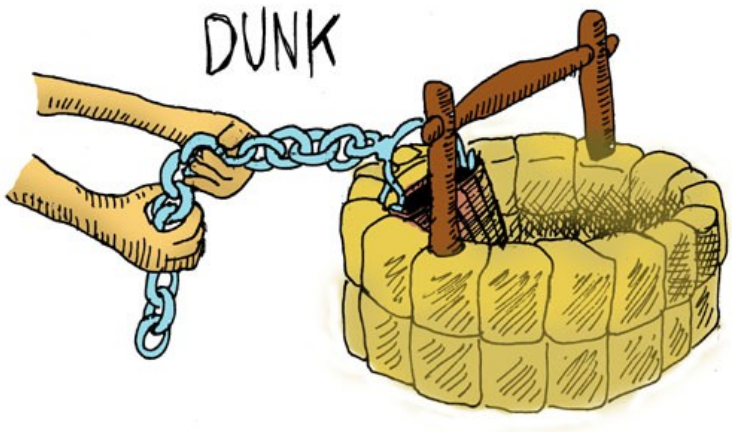
```
chain weld bucket;
==> you cannot weld like that.
```

Offensichtlich waren noch nicht alle Bedingungen erfüllt.

Ist die Kette einmal sicher an den Eimer geschmiedet, können wir damit dann Wasser aus dem Brunnen holen.

```
bucket_filled: false$

infix("dunk")$
"dunk"(subject,object):=
  if location='garden
     and subject='bucket
     and object='well
     and have('bucket) then (
        if chain_welded then (
           bucket_filled: true,
           "the bucket is now full of water. ")
        else "the water level is too low to reach.")
  else  "you cannot dunk like that. "$
```



Wir sehen, dass sich die Befehle `weld` und `dunk` sehr ähneln. Beide überprüfen erst einmal, ob Ort, Subjekt und Objekt stimmen. Man findet dann aber wiederum auch genug Unterschiede, so dass man nicht alles in eine einzige Funktion stecken kann. Aber, wie gesagt, Maxima baut auf Lisp auf, und das heißt, das es auch für dieses Problem eine Lösung gibt: SPELs.

Chapter_7: "Casting SPELs" $

Wir lernen in diesem Kapitel ein unglaublich machtvolles Sprachelement kennen. Conrad Barski nennt es **SPEL** ("Semantic Program Enhancement Logic"). Bekannter ist es unter dem Namen **Makro**, hat aber mit einem Makro in C++ nur wenig gemeinsam. Eigentlich geht es hier, wie der Name SPEL schon andeutet, um Zauberei. SPELs ermöglichen uns, auf der elementarsten Ebene die Sprache von Maxima selbst für unsere Zwecke zu gestalten. Wir werden zum Designer einer Sprache.

Die Funktion, die in Maxima Makros ermöglicht, heißt `buildq`, **build quoted**. Dieser Name beschreibt, wie wir gleich sehen werden, eigentlich sehr schön, worum es bei einem Makro geht.

Ein quote ist ein Hochkomma und wir wissen schon, dass gequotete Ausdrücke nicht ausgewertet werden. Ausdrücke schreiben zu können, die erst zu einer späteren Zeit ausgewertet werden, eröffnet ungeahnte Möglichkeiten, die, wie gesagt, schon manchmal an Zauberei grenzen. Eine Makrodefinition wird mit einem `::=` statt einem `:=` geschrieben. Der zusätzliche `:` deutet die nachträgliche Auswertung an.

Conrad Barski zu Ehren soll `buildq` nun in `SPEL` umbenannt werden.

```
SPEL([rest])::= buildq([rest],buildq(splice(rest)))$
```

Diese Umbenennung zeigt auf raffinierte, aber auch sehr abstrakte Weise, wie `buildq` funktioniert. `buildq` besitzt grundsätzlich zwei Argumente, eine Liste mit Substitutionsparametern und einen Ausdruck. Würde hier wie gewohnt von innen nach außen ausgewertet werden, bekämen wir sofort eine Fehlermeldung. Der Parser würde feststellen, dass das innere `buildq` nur ein einziges Argument besitzt. Statt dessen werden zuerst einmal nur die Werte der Substitutionsparameter gequotet in den Ausdruck eingesetzt. Die Auswertung findet dann anschließend statt.

Von links nach rechts: Der Ausdruck `[rest]` in einer Funktionsdefinition bedeutet, dass diese Funktion **optionale Argumente** besitzt. Diese werden dann in einer Liste namens `rest` gesammelt und weitergereicht. `buildq` setzt nun diese Liste in den Ausdruck `buildq(splice(rest))` ein. Eine besondere Rolle spielt bei dieser Ersetzung die Funktion `splice` . `splice` macht aus der Liste `rest` wieder einzelne Argumente.
Wenn `SPEL` also nun vorschriftsmäßig mit zwei Parametern aufgerufen wird, wird letztendlich daraus wieder `buildq` mit exakt denselben zwei Argumenten. Auf diese Weise können wir nun das Symbol `SPEL` an Stelle von `buildq` verwenden.

Erst ersetzen, später auswerten. Dieser Trick erlaubt es, Funktionen zu schreiben, die zur Laufzeit eines Programms Funktionen oder jeden anderen beliebigen Code schreiben.

In der nun folgenden Definition des Makros `game_action` werden wir uns sehr konkret ansehen können, wie `SPEL` funktioniert.

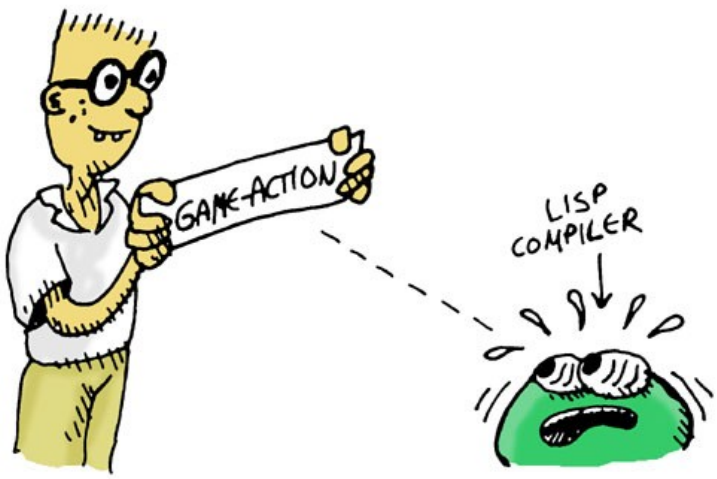
```
game_action(command,subj,obj,place,[rest])::= SPEL(
   [command,subj,obj,place,rest],
   block(
      infix(command),
      command(subject,object):= block(
         if location = place
            and subject = subj
            and object = obj
            and have(subj) then apply(sconcat,rest)
         else sconcat("you cannot ",command," like that. ") )))$
```

Bei näherer Betrachtung sehen wir, dass **SPEL** tatsächlich nur zwei Argumente besitzt. Die Liste der Ersetzungsparameter und einen Block. Und der Block enthält genau das, was in den Definitionen von **weld** und **dunk** analog ist. Es ist die Struktur. **game_action** ist im Prinzip eine Art Schablone. An passender Stelle werden dann in diese die Namen des Kommandos, des Subjekts, Objekts und des Ortes, an dem die Aktion stattfinden soll, eingesetzt. Die unterschiedlichen Textausgaben der einzelnen Kommandos werden dann als optionale Parameter eingetragen.

Ein Beispiel sagt hier mehr als tausend Worte. Wir definieren mit Hilfe von **game_action** das Kommando **weld** noch einmal neu. Wenn wir dabei die Eingabe mit einem Semikolon abschließen, erhalten wir ja bekanntlich als Antwort den Wert dieses Makros. In diesem Fall ist das die Definition des Operators **weld**.

```
game_action("weld",chain,bucket,attic,
   if have(bucket)
      and not chain_welded then (
         chain_welded: true,
         "the chain is now securely welded to the bucket. " )
   else "you do not have a bucket. ");
==> subject weld object := block(
if location = attic
   and subject = chain
   and object = bucket
   and have(chain) then
      apply(sconcat,
            [if have(bucket)
                and not chain_welded then (
                   chain_welded : true,
                   "the chain is now securely welded to the bucket. ")
             else "you do not have a bucket. "])
else sconcat("you cannot ", "weld"," like that. "))
```

SPEL!



Ob der Lisp Compiler, der in unserem Spiel der Frosch des Zauberers zu sein scheint, dabei tatsächlich ins Schwitzen kommt, bleibt zu bezweifeln.

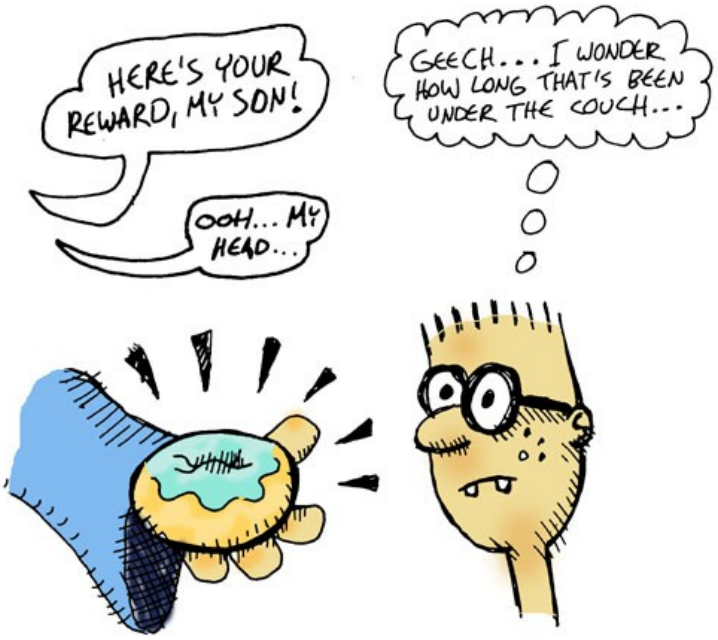
Das Makro `game_action` ermöglicht es, dass wir uns bei der Definition der Kommandos auf die speziellen Einzelheiten des jeweiligen Kommandos konzentrieren können. Die ganze Sache sieht dann wesentlich schöner und übersichtlicher aus. Im Falle von `dunk` dann so.

```
game_action("dunk",bucket,well,garden,
   if chain_welded then (
      bucket_filled: true,
      "the bucket is now full of water. " )
   else "the water level is too low to reach. ")$
```

Und nun unser letztes Kommando. `splash`.

Befindet sich Wasser im Eimer, wecken wir damit den Zauberer.

```
game_action("splash",bucket,wizard,living_room,
   if not bucket_filled then "the bucket has nothing in it. "
   else if have(frog) then sconcat(
      "the wizard awakens and sees that you stole his frog. ",
      "he is so upset he banishes you to the netherworlds ",
      "- you lose! the end. ")
   else sconcat(
      "the wizard awakens from his slumber and greets you warmly. ",
      "he hands you the magic low_carb donut - you win! the end. ") )$
```



Du hast damit ein vollständiges Text Adventure Game geschrieben!

Jetzt solltest Du das Spiel natürlich auch spielen. Und mit ein wenig Glück gibt der Zauberer Dir auch Deine Belohnung.

Ein `magic low_carb donut`!

"Aufgaben"$

2. Erweitere das Kommando `casting SPELs` aus Aufgabe 1 so, dass damit zu jeder beliebigen Zeit das gesamte Spiel in einem zufälligen Anfangszustand gebracht wird.

3. Werde kreativ und erweitere das Spiel um SPELs, locations, game_actions oder sonstige nette Kleinigkeiten und schicke davon eine Kopie an den Übersetzer Volker van Nek.

"Danksagung"$

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

Volker van Nek, Aachen, Januar 2006

"Lizenz"$