Seminararbeit

LISP

Matej Stanic
m.stanic@uibk.ac.at

16. Februar 2013

Betreuer: Martin Avanzini

Zusammenfassung

Lisp (List processing) is a functional programming language based on the lambda calculus which was designed by John McCarthy in the late 1950s. Being one of the oldest, it has influenced a lot of well-known programming languages by introducing new features and paradigms. This seminar report gives a brief history of Lisp and is primarily concerned with important features of Common Lisp, a dialect introduced in the 1980s. The syntax and the functionality of Common Lisp are described, with a focus on special types of functions, so-called macros. In conclusion, the main example of this seminar, 99 Bottles of Beer, is presented.

Inhaltsverzeichnis

1 Einleitung 1

2 Geschichte 1

3 Grundlegende Eigenschaften 2

4 Syntax 2

5 Wichtige Datenstrukturen 4
- 5.1 Listen 4
- 5.2 Andere Datenstrukturen 5

6 Variablen 5

7 Funktionen 6
- 7.1 Lambda-Ausdrücke und Funktionen höherer Ordnung 6
- 7.2 Optionale Parameter 7

8 Kontrollstrukturen 7

9 Makros 9

10 Beispiel: 99 Bottles of Beer 11

11 Zusammenfassung 11

1 Einleitung

Lisp (Abk. für List Processing) ist eine auf dem Lambda-Kalkül basierende, funktionale Programmiersprache, die in den späten 1950er Jahren von John McCarthy entworfen wurde. Aufgrund der vielen neuartigen Eigenschaften in der damaligen Zeit, beeinflusste sie sehr viele bekannte Programmiersprachen wie z.B. *Haskell* und *Smalltalk*. Trotz der einfach gehaltenen, listenbasierten Syntax ist Lisp sehr mächtig, vor allem wegen der Makrofunktion, die es erlaubt Quellcode während der Laufzeit zu manipulieren. In dieser Seminararbeit wird Common Lisp vorgestellt, einer der bekanntesten und am häufigsten benutzten Dialekte von Lisp. Der Einfachheit halber ist im Folgenden die Rede von Lisp, wobei im speziellen die Implementierung *Steel Bank Common Lisp*[1]verwendet wurde.

2 Geschichte

Die Ursprünge von Lisp gehen zurück auf John McCarthy, einem Pionier der künstlichen Intelligenz. Während der *Dartmouth Conference* 1956, der ersten Konferenz für Künstliche Intelligenz, kam er zum ersten Mal in Kontakt mit der Programmiersprache *IPL 2*, einer listen-basierten Sprache für das *JOHNNIAC*-System. McCarthy, der zu dieser Zeit am MIT (Massachusetts Institute of Technology) arbeitete, hatte nur IBM 704-Systeme zur Verfügung, welche nicht kompatibel mit IPL 2 waren. Daraufhin entschloss sich McCarthy eine neue Programmiersprache, basierend auf IPL 2, zu entwerfen. Zunächst war diese neue Programmiersprache nur auf FORTRAN-Unterprogramme beschränkt. Schließlich entdeckte Steve Russell, ein Student McCarthys, dass der Code auch von einem Interpreter verarbeitet werden könnte und implementierte den ersten LISP-Interpreter im Jahre 1958. Detaillierte Informationen über die Entwicklung von Lisp können in McCarthys Bericht [McC79] nachgelesen werden.
LISP 1.5 wurde 1960 zur ersten Implementierung, die über das MIT hinweg verbreitet wurde. Nach und nach wurde der Funktionsumfang von Lisp erweitert und zahlreiche Lisp-Dialekte entstanden, wie zum Beispiel *Scheme*. Aufgrund großer Probleme mit der Standardhardware der 70er Jahre wurden spezielle, optimierte Lisp-Maschinen entworfen, die das Ausführen von Lisp-Programmen erleichterten.
Ein anderes Problem war die Vielfalt von Lisp-Dialekten, die jeweils unterschiedliche Funktionalitäten besaßen. Um die steigende Vielfalt und Inkompatibilität zu dämmen, wurde 1984 *Common Lisp* implementiert, eine Zusammenlegung von mehreren Lisp-Dialekten. 1994 wurde Common Lisp ANSI-standardisiert [Ste90] (ANSI/X3.226-1994).
Obwohl Lisp historisch gesehen eine der Sprachen der künstlichen Intelligenz war, wird sie heute nur selten verwendet. Wegen ihrer einfachen Syntax und ihren positiven Eigenschaften wird sie in den USA oft als erste Programmiersprache gelehrt, wobei Common Lisp und Scheme am häufigsten eingesetzt werden.

[1]siehe `http://www.sbcl.org/`

3 Grundlegende Eigenschaften

- **Lisp ist multiparadigmatisch.** Ein Lisp-Programm besteht ausschließlich aus Funktionen, die evaluiert werden und gegebenenfalls Werte zurückgeben. Da diese Funktionen auch Seiteneffekte besitzen können, ist Lisp nicht nur funktional, sondern auch imperativ. Mit CLOS (Common Lisp Object System)[2]wird eine objektorientierte Erweiterung für Lisp zu Verfügung gestellt. Näheres zu Funktionen in Kapitel 7.

- **Lisp ist eine listenbasierte Programmiersprache.** Die sehr einfach gestaltete Lisp-Syntax besteht aus sogenannten S-Expressions. Zwischen Quellcode und Daten wird nicht unterschieden. Da verschachtelte Anweisungen viele Klammern benötigen, wird Lisp manchmal ironischerweise auch als „Lots of Irritating Superfluous Parentheses" (eine Menge lästiger, überflüssiger Klammern) interpretiert. Näheres zur Syntax in Kapitel 4.

- **Lisp ist dynamisch typisiert.** Variablen müssen keine Typen zugewiesen werden. Aus Effizienzgründen werden optionale Typen angeboten, welche dynamisch, also zur Laufzeit typisiert werden.

- **Lisp-Code muss nicht vorkompiliert werden.** Sobald Lisp-Code eingegeben wird, wird dieser sofort vom Interpreter in einer sogenannten *Read-Eval-Print-Schleife* ausgewertet. Die Funktion wird eingelesen, ausgewertet und ausgegeben. Der Lisp-Interpreter ist kein reiner Interpreter, da der Code beim Einlesen dynamisch kompiliert wird.

- **Lisp ist eine "programmierbare Programmiersprache".** Spezielle Funktionen, sogenannte *Makros*, ermöglichen es dem Programmierer, ausführbaren Code in Lisp selbst zu implementieren. Dadurch ist Lisp beliebig erweiterbar: Neue Kontrollstrukturen, Entwurfsmuster usw. können direkt implementiert werden ohne die Sprachdefinition zu ändern. Mehr zu Makros in Kapitel 9.

- **Automatische Speicherbereinigung.** Speicher muss nicht explizit reserviert und freigegeben werden.

4 Syntax

Die Lisp-Syntax besteht aus sogenannten S-Expressions, die induktiv definiert sind: Eine S-Expression ist entweder ein Atom oder eine Liste von S-Expressions. **Atome** sind entweder selbstevaluierende Objekte wie Zahlen und Strings, oder Symbole, die für andere Werte oder Funktionen stehen:

- **Zahlen** (`number`):
  `1   2   3.4   -1.5633335   6.02E+23`

[2]Näheres siehe `http://www.dreamsongs.com/CLOS.html`

- **Symbole** (`symbol`): bis auf ein paar Ausnahmen (Klammern, Anführungszeichen, Leerzeichen...) sind alle Zeichen erlaubt.
  `counter` `x` `foo` `TR74g` `%arg%` `false`

- **Strings**: Zeichenfolgen in Anführungszeichen.
  `"Freitag"` `"44"` `"blabla"`

Listen bestehen aus einer öffnenden Klammer, beliebig vielen Listenelementen und einer schließenden Klammer (siehe Beispiel 4.1). Da zwischen Funktionen und Daten nicht unterschieden wird, werden Listen auch für Funktionsaufrufe benutzt; Dabei wird die Präfixnotation verwendet: Das erste Element der Liste ist die Funktion, alle weiteren Elemente sind Funktionsargumente.
Listing 1 zeigt zwei Beispiele von Funktionsaufrufen: Die Liste `(* 3 4)` wendet die Funktion `*` auf die Argumente `3` und `4` an. Als Ergebnis wird `12` zurückgegeben. Im zweiten Beispiel wird die Funktion `+` auf das Ergebnis der zwei Funktionsaufrufe `(* 2 1)` und `(- 4 2)` angewendet, die Ausgabe ist `4`. (Anmerkung: Zeilen mit > bezeichnen Benutzereingaben im Interpreter, Zeilen ohne > und ohne Leerzeichen/Tabulatoren bezeichnen Ausgaben. Diese Notation wird in den weiteren Beispielen beibehalten)
Die Listenstruktur ist nichts anderes als eine Baumstruktur, d.h. ganze Lisp-Programme können als Baum dargestellt werden (siehe Abb. 1).

Beispiel 4.1. `(1 2 (3 4))` ist eine Liste bestehend aus den Zahlen 1, 2 und einer Liste, die wiederum aus den Zahlen 3 und 4 besteht.

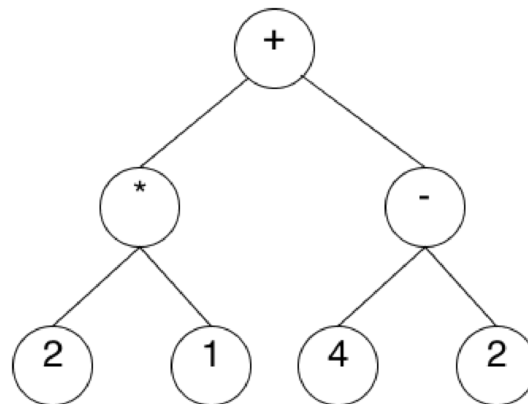

Abbildung 1: Baumstruktur der Liste `(+ (* 2 1) (- 4 2))`

Da man Funktionen häufig nicht evaluieren will, gibt es die Funktion `quote`. Mittels `quote` wird die Evaluierung der gekennzeichneten Argumente verhindert. Da diese Funktion sehr oft benutzt wird, wurde `'` als Abkürzung für `quote` eingeführt (siehe Listing 2).

```
> (* 3 4)
12
> (+ (* 2 1) (- 4 2))
4
```

Listing 1: Funktionsaufrufe im Interpreter mit Ausgabe

```
> (quote (+ 2 3))
(+ 2 3)
> '(+ 2 3)
(+ 2 3)
```

Listing 2: Verwendung von quote und ' im Interpreter

5 Wichtige Datenstrukturen

5.1 Listen

Die grundlegende Datenstruktur von Lisp ist die *einfach verkettete Liste*. Jedes Element einer Liste besitzt einen Zeiger auf ein Datum (`car`[3]-Zeiger) und einen Zeiger auf eine andere Liste (`cdr`[4]-Zeiger). Eine Lisp-Liste ist entweder die leere Liste (`nil`) oder eine Zusammensetzung aus Listenelementen, bei der der `car`-Zeiger auf das aktuelle Element, und der `cdr`-Zeiger auf den Rest der Liste zeigt. Abb. 2 zeigt die bildhafte Darstellung einer Lisp-Liste anhand eines Beispiels.

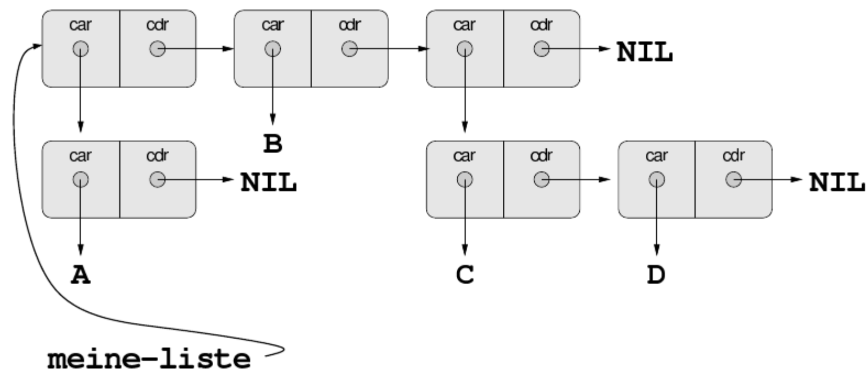

Abbildung 2: Interne Darstellung der Liste `((A) B (C D))`

Zahlreiche Funktionen zur Erstellung, Traversierung und Bearbeitung von Listen stehen zur Verfügung. Die Funktion `list` erstellt eine Liste aus beliebig vielen Argumenten, `cons` fügt ein Element an eine Liste hinzu. Mittels `car` und `cdr` kann auf einzelne Elemente von Listen zugegriffen werden. `Car` gibt das erste

[3] Abk. für *Contents of Address Register* (siehe `http://en.wikipedia.org/wiki/CAR_and_CDR`)

[4] Abk. für *Contents of Decrement Register* (siehe `http://en.wikipedia.org/wiki/CAR_and_CDR`)

```
> (let ((x 10))
   (print x))
10
> x
Error: Attempt to take the value of unbound
variable 'X'.
```

Listing 3: Definition einer lexikalen Variable

Element zurück, während `cdr` den Rest der Liste zurückgibt. Durch Kombination von `car` und `cdr` kann auf jedes einzelne Element einer Liste zugegriffen werden.
Weitere wichtige Funktionen, die in einer funktionalen Programmiersprache nicht fehlen dürfen, sind die Funktionen höherer Ordnung `mapcar` (map) und `reduce` (fold). `Mapcar` wendet eine Funktion auf jedes Element einer oder mehrerer angegebener Listen an, während `fold` eine Funktion zwischen den Elementen anwendet (auch als *Faltung* bekannt). Mehr zu `mapcar` und zu Funktionen höherer Ordnung im Allgemeinen in Kapitel 7.1.
Weitere Funktionen zu Listen können unter [Ste90] nachgesehen werden.

5.2 Andere Datenstrukturen

Zahlreiche andere Datenstrukturen werden von Lisp unterstützt, Wie z.B. Arrays oder Hashtables. Näheres zu diesen Datenstrukturen kann in [Ste90] nachgelesen werden.

6 Variablen

Mit der *Special-form* `setf` kann einem Symbol ein Wert zugewiesen werden. Eine Special-Form verhält sich wie eine Funktion, weicht aber von den Evaluationsregeln normaler Funktionen ab. In diesem Fall ist die Wertzuweisung nur ein Seiteneffekt, der eigentliche Rückgabewert ist der übergebene Wert.

Lisp unterscheidet zwischen zwei Variablentypen:

- **Lexikale Variablen** verhalten sich wie lokale Variablen in anderen Programmiersprachen. Sie sind nur in der Umgebung sichtbar, in der sie definiert wurden. Lexikale Varibalen werden mit `let` definiert.
  Listing 3 zeigt eine beispielhafte Benutzung von `let`: Die Variable `x` ist nur innerhalb der Klammer sichtbar, in der ihr der Wert zugewiesen wurde. Bei einem Aufruf außerhalb, kommt es zu einem Fehler, da `x` nicht bekannt ist.

- **Dynamische Variablen** können mit globalen Variablen verglichen werden. Einmal deklariert, können sie überall eingesehen und verändert werden.

```
> (defparameter *globalvar* 10)
*GLOBALVAR*
> *globalvar*
10
```

Listing 4: Definition einer dynamischen Variable

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Listing 5: Definition einer rekursiven Funktion

Wie Listing 4 zeigt, werden dynamische Variablen mit `defparameter` definiert. Zur leichteren Unterscheidung von anderen Variablen sind Sternchen (*) am Anfang und Ende des Variablennamens Konvention.

7 Funktionen

Neben den schon implementierten Standardfunktionen ist es natürlich möglich eigene Funktionen zu definieren. Dies geschieht mit der Special-form `defun <name> <paramlist> <functionbody>`, wobei `name` den Namen der Funktion, `paramlist` eine Liste von Parametern und `functionbody` den Funktionskörper definiert. Listing 5 zeigt die Definition der rekursiven Funktion `factorial`, die die Faktorielle einer Zahl berechnet. Man beachte die Kontrollstruktur `if`, die nach dem Prinzip *if-then-else* funktioniert.

7.1 Lambda-Ausdrücke und Funktionen höherer Ordnung

Da Lisp auf dem Lambda-Kalkül basiert, gibt es eine Methode Funktionen ohne Namen darzustellen. Bei dem Design der Funktion `lambda` orientierte sich McCarthy stark nach der Church-Notation (siehe Beispiel 7.1). Solche Funktionen werden auch anonyme Funktionen genannt, da sie keinen Namen besitzen und nur über Referenzen oder Zeiger angesprochen werden können.

Beispiel 7.1. Eine Funktion, die ein Argument `x` nimmt und um 2 erhöht, sieht in Lambda-Notation folgendermaßen aus: $\lambda x.x + 2$
Das Lisp-Äquivalent dazu ist `(lambda (x) (+ x 2))`.

Anonyme Funktionen werden z.B. als Parameter für Funktionen höherer Ordnung verwendet. Funktionen höherer Ordnung nehmen entweder eine Funktion als Input oder geben eine Funktion als Output zurück. Beispiele für solche Funktionen sind die schon erwähnten *map* und *fold*-Funktionen für Listen, die in Lisp als `mapcar` und `reduce` realisiert sind.
Listing 6 zeigt die Verwendung von `mapcar` anhand eines Beispiels. Zur Erinnerung: `mapcar` wendet eine Funktion auf jedes Element einer oder mehrerer

```
> (mapcar #'list '(a b c) '(1 2 3))
((A 1) (B 2) (C 3))
```

Listing 6: Beispiel zu `mapcar`

```
> (defun person (vorname nachname &optional (alter 'unbekannt))
    (progn
    (format t "Die␣Person␣heisst␣~s␣~s." vorname nachname )
    (format t "Das␣Alter␣der␣Person␣ist:␣~s" alter)))
> (person 'max 'mustermann)
Die Person heisst MAX MUSTERMANN.
Das Alter der Person ist: UNBEKANNT
> (person 'max 'mustermann '45)
Die Person heisst MAX MUSTERMANN.
Das Alter der Person ist: 45
```

Listing 7: Definition einer Funktion mit optional

angegebener Listen an; In diesem Fall ist es die Funktion `list`, die auf die zwei Listen `(a b c)` und `(1 2 3)` angewandt wird. Aus den jeweils ersten, zweiten und dritten Elementen der zwei Listen werden neue Listen erstellt, `(A 1)`, `(B 2)` und `(C 3)`.
Da zwischen Funktionen und Daten nicht unterschieden wird, ist es kein Problem eine Funktion als Argument zu übergeben. Man bemerke das Leser-Makro `#`, welches bewirkt, dass der Code auch ausgeführt wird. Mehr zu Makros in Kapitel 9.

7.2 Optionale Parameter

Funktionen müssen nicht immer eine fixe Anzahl von Parametern besitzen. Mit dem Schlüsselwort `&optional` kann man Funktionen definieren, die optionale Parameter nehmen können, aber nicht müssen.
7 zeigt ein personenbezogenes Beispiel, bei dem Vor- und Nachname angegeben werden müssen, das Alter jedoch ein optionaler Parameter ist. Falls das Alter nicht angegeben ist, wird einfach `UNBEKANNT` verwendet.

Eine andere Möglichkeit, einer Funktion eine variable Anzahl von Parametern zu übergeben ist das Schlüsselwort `&rest`. Einem mit `&rest` markierten Parameter wird eine Liste zugewiesen, die beliebig lang sein kann.
Die Funktion `count-arg` in 8 zählt die Anzahl der Argumente der übergebenen Liste `list`, die mit `&rest` markiert wurde.

8 Kontrollstrukturen

Kontrollstrukturen werden in dieser Seminararbeit nicht im Detail behandelt, detaillierte Beschreibungen und Funktionsweisen können in [Ste90] nachgelesen werden. Hier ein paar Grundlagen bezüglich Kontrollstrukturen:

```
> (defun count-arg (&rest list) (length list))
> (count-arg 'x 'y 2 6 'z)
5
```

Listing 8: Definition einer Funktion mit rest

```
> (do ((i 1 (+ i 1)))
  ((> i 3) 'ende)
  (format t "~A~%" i))
1
2
3
ENDE
```

Listing 9: For-Schleifendefinition mit `do`

- Lisp war die erste Programmiersprache, die Conditional Statements einführte, welche heute nicht mehr wegzudenken sind. Conditional Statements werden mit `cond` gekennzeichnet und funktionieren äquivalent zu dem bekannten `switch` aus der Programmiersprache C. Eine andere Möglichkeit, Conditional Statements zu benutzen, ist das auf dem if-then-else basierende `if`. (Anmerkung zu Booleans: `nil` entspricht dem booleschen "falsch", alle anderen Werte entsprechen dem booleschen "wahr".)

- Schleifen können in Lisp mit dem Makro `do` definiert werden. Listing 9 zeigt die Benutzung von `do` anhand eines Beispiels. `Do` erwartet drei Argumente: Das erste Argument ist eine Liste mit Variablenspezifikationen. Jede solche Liste besitzt die Form `(<variable> <startwert> <schrittweite>)`; In Listing 9 ist dies die Liste `(i 1 (+ i 1))`. `i` hat den Startwert `1` und wird nach jeder Schleifeniteration um `1` erhöht. Das zweite Argument ist ein Testausdruck mit einem optionalen Rückgabewert, welcher die Schleife beendet. In Listing 9 ist dies die Liste `((> i 3) 'ende)`, d.h. wenn `i` größer als `3` ist, wird die Schleife beendet und `ende` wird zurückgegeben. Das dritte Argument beinhaltet den Schleifenkörper, welcher in jeder Schleifeniteration ausgeführt wird. `(format t "~A~%" i)` gibt den aktuellen Wert von `i` aus.
  Weitere von Lisp bereitgestellte Iterationskonstrukte sind `dotimes`, `while` `for` und `loop`.

- `progn` führt alle Anweisungen, die sich innerhalb des gleichen Blocks befinden, nach der Reihe aus. Dies ist wichtig bei `if`-Anweisungen, wie Listing 10 zeigt: Ohne `progn` würde nur `(print 'true)` ausgeführt werden, da `(setf var 5)` als else-Zweig interpretiert werden würde. Durch die Erstellung eines Codeblocks mittels `progn` kann sichergestellt werden, dass auch `(setf var 5)` ausgeführt wird.

```
> (if (> 4 3)
  (progn
  (print 'true)
  (setf var 5))
  (print 'false))
TRUE
5
```

Listing 10: `progn` in einem `if`-Konstrukt

```
> (defmacro my-if (test then else)
  (list 'cond (list test then) (list 't else)))
> (macroexpand '(my-if (< 3 2) 12 42))
(COND ((< 3 2) 12) (T 42))
```

Listing 11: Definition und Ausführung eines Makros

9 Makros

Eine der größten Besonderheiten von Lisp ist das Makrosystem, welches dem Programmierer erlaubt, die Programmiersprache zu erweitern ohne den Compiler zu verändern. Dabei werden bestimmte Anweisungen vor dem Evaluieren zuerst in andere Anweisungen umgewandelt. Makros werden mit `defmacro` definiert.
Listing 11 verdeutlicht die Funktionsweise von Makros anhand eines eigenen if-Konstrukts `my-if`. Beim Aufruf von `(my-if (< 3 2) 12 42)` wird die Anweisung `(list 'cond (list (< 3 2) 12) (list 't 42))` in `(cond ((< 3 2) 12) (T 42))` umgewandelt. Diesen Vorgang bezeichnet man als *Makroexpansion*. Der neu generierte Code wird zurückgegeben und anstatt des Makros evaluiert. In Listing 11 wird daher nach dem Makroaufruf die Anweisung `(cond ((< 3 2) 12) (T 42))` evaluiert. Mittels `macroexpand` können Makros angewandt werden, ohne den umgewandelten Ausdruck zu evaluieren.

Dieses Beispiel ist nicht sehr überzeugend, da Makros in dieser Form nicht benötigt werden. `My-if` hat den gleichen Effekt wie das schon vorhandene `if`. Um sinnvollere und komplexere Makros zu konstruieren, werden zusätzliche Hilfsmittel benötigt. Ein solches Hilfsmittel ist das *Backquote* ‘, welches genauso wie `quote` die Evaluierung von Funktionen verhindert. Dazu kommt aber, dass man die Evaluierung wieder einschalten kann, indem man ein Komma (,) vor die entsprechenden Elemente setzt. Listing 12 zeigt die Auswirkungen von Backquote und Komma in einem einfachen Beispiel. Eine Variation des Kommas für Listen ist das Komma-at (,@), welches das Komma auf jedes Element einer Liste anwendet.

Backquote und Komma erlauben nun das Erstellen komplexer Makros. Typischerweise sind dies neue, selbst definierte Kontrollstrukturen. In Listing 13 wird die Kontrollstruktur until definiert, die den Funktionskörper `body` solange ausführt, bis die Bedingung `test` zu NIL evaluiert. `&body` spezifiziert `body` als

```
> (setf b 5)
5
> `(a ,b c)
(A 5 C)
```

Listing 12: Effekte von Backquote und Komma

```
(defmacro until (test &body body)
  (let ((start-tag (gensym "START"))
        (end-tag   (gensym "END")))
    `(tagbody ,start-tag
              (when ,test (go ,end-tag))
              (progn ,@body)
              (go ,start-tag)
              ,end-tag)))
```

Listing 13: Definition von `until` mittels eines Makros

eine Liste variabler Länge. `&body` hat also den gleichen Effekt wie `&rest`, wird aber aufgrund der einfacheren Verständlichkeit verwendet, da `&body` auf ein implizites `progn` hinweist. Die `let`-Anweisung definiert zwei Variablen `start-tag` und `end-tag`, die im weiteren Verlauf als Tags dienen. Die Variablen werden mittels `tagbody` als Tags spezifiziert, zu denen mit einer `go`-Anweisung direkt gesprungen werden kann. Weiters wird die eigentliche Kontrollstruktur definiert: Wenn die Bedingung `test` zu `t` evaluiert, wird sofort zu `end-tag` gesprungen, welches das Ende der Funktion kennzeichnet. Andererseits wird der Funktionskörper `body` ausgeführt und es wird zurück zum `start-tag` gesprungen, wo sich das ganze Procedere wiederholt.
In Listing 14 wird das Makro `until` mit `makroexpand-1` expandiert, welches die Funktionsweise von `until` nocheinmal verdeutlicht. (Anmerkung: `makroexpand-1` funktioniert wie `makroexpand`, jedoch wird nur einmal expandiert, und nicht wie bei `makroexpand` so lange wie möglich.)

Common Lisp-Makros sind nicht hygienisch, d.h. es können Namenskonflik-

```
(TAGBODY
 #:START1136
 (WHEN (= (RANDOM 10) 0)
   (GO #:END1137))
 (PROGN (PRINT "hello"))
 (GO #:START1136)
 #:END1137)
```

Listing 14: Aufruf von `macroexpand-1 (until (= (random 10) 0) (print Ḧello)̈)`

te entstehen (*Variable capture*). Aus diesem Grund wird bei der Definition der Tags das Schlüsselwort `gensym` verwendet; Es stellt sicher, dass es kein reguläres Symbol gibt, dass zu einem `gensym`-Symbol identisch ist. Im Gegensatz zu Common Lisp verwendet *Scheme* hygienische Makros, welche Namenskonflikte automatisch vermeiden.

10 Beispiel: 99 Bottles of Beer

Beispiel 10.1 implementiert das Programm *99 Bottles of Beer*, welche als Funktion `bottles` definiert ist.
Die Funktion nimmt als Parameter die Anzahl der Bierflaschen und wird solange rekursiv aufgerufen, bis keine Flasche mehr übrig ist. Die Funktion `format` gibt den ihr übergebenen Text formatiert aus, die Funktion ist das Äquivalent zu dem bekannten `printf` aus der Programmiersprache C. Wenn `(< n 1)` nicht zu T evaluiert, wird `progn` ausgeführt, welches einfach alle übergebenen Anweisungen der Reihe nach ausführt. In diesem Fall wird formatierter Text ausgegeben und `bottles` wird mit einer Flasche weniger ausgeführt.

Beispiel 10.1. Implementierung von *99 Bottles of Beer.*

```
;; Bottles by Rebecca Walpole (walpolr@cs.orst.edu)
;; tested in Austin Kyoto Common Lisp version 1.615
;; Note: the p takes care of plurals.
;;

(defun bottles (n) "Prints the lyrics to '99 Bottles of Beer'"
  (if (< n 1)
    (format t -%Time to go to the store. %")
    (progn (format t -% a bottle :p of beer on the wall."n)
      (format t -% a bottle :p of beer."n)
      (format t -% Take one down, pass it around.")
      (format t -% a bottle :p of beer on the wall. %"(- n 1))
      (bottles (- n 1))
    )
  )
)

(bottles 99)
```

11 Zusammenfassung

Lisp schafft es das mächtige Konzept der Flexibilität mit einer sehr einfachen und durchschaubaren Syntax zu kombinieren. Obwohl Lisp funktional ist und auf dem Lambda-Kalkül basiert, erlauben Seiteneffekte von Funktionen auch imperative Befehle. Die Gleichbehandlung von Code und Daten erlaubt es, Code während der Laufzeit zu manipulieren und Lisp beliebig zu erweitern, ohne die

Sprachdefinition zu verändern. Obwohl viele Konzepte aus anderen Programmiersprachen ihren Ursprung in Lisp haben, ist Lisp heute bei weitem nicht mehr relevant wie in den 1970er Jahren. Nichtsdestotrotz erweitert Lisp den Horizont eines jeden Programmierers, da viele Konzepte in den heute relevanten Programmiersprachen nicht vorhanden sind.

Literatur

[McC79] John McCarthy. History of Lisp. Technical report, Artificial Intelligence Laboratory, Stanford University, 1979.

[Ste90] Guy L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.