Proseminar Programmiersprachen: Lisp

Peter Kiechle
Betreuer: Steven Obua

11. Dezember 2006

TU München, Wintersemester 2006/07
Kursleiter: Prof. Tobias Nipkow, Dr. Stefan Berghofer

Inhaltsverzeichnis

1 Geschichte

Laut eigenem Bericht [McCarthy, 1978] kam John McCarthy, ein Pionier der Künstlichen Intelligenz, auf der berühmten Dartmouth Conference 1956 das erste Mal mit IPL 2 in Berührung. Dabei handelt es sich um eine auf Listen basierende Programmiersprache für das JOHNNIAC-System. McCarthy, der am M.I.T. mit einem dazu inkompatiblen IBM 704 System arbeitete, erkannte das Potential und begann mit dem Design einer neuen Sprache im Rahmen eines KI Projektes. Da es zu aufwendig erschien einen komplett neuen Compiler zu schreiben, begnügte man sich mit einem FORTRAN Unterprogramm. Bei der Entwicklung von FLPL (FORTRAN List Processing Language) stellte man fest, dass wichtige Konzepte wie Rekursion und Kontrollstrukturen nicht (effizient) implementierbar waren und so wurde 1958 damit begonnen einen Lisp-Compiler zu entwickeln. Steve Russell, ein Student von McCarthy, erkannte, dass Lisp-Code auch von einem Interpreter verarbeitet werden kann und schrieb die ersten beiden Implementierungen. Nach und nach wurde der Funktionsumfang erweitert und neue Konzepte (z.B. "garbage collection", "if-then-else", "lexical closures") eingeführt, die zur damaligen Zeit noch in keiner anderen Programmiersprache vorhanden waren. Da Lisp vor allem wegen seiner Makrofunktion beliebig erweiterbar ist, entstanden bald eine Vielzahl verschiedener Dialekte, darunter auch Scheme. Als dann in den 70er Jahren der Bereich der Künstlichen Intelligenz immer wichtiger wurde, stieß Lisp aufgrund der beschränkten Hardware dieser Zeit an seine Grenzen. Um die Performanz von Lisp Programmen zu verbessern, wurden eigene Computer entwickelt, die für den Lisp-Befehlssatz optimiert waren und mit ihrem besonders großen Adressraum eine relativ hohe Rekursionstiefe erlaubten. Heute ist spezielle Hardware aufgrund des technischen Fortschritts nicht mehr notwendig. Als Ende der 70er Jahre Smalltalk entstand, fand das Paradigma der Objektorientierung sogleich Einzug in die Erweiterung LOOPS (Lisp Object Oriented Programming System). Um die vielen unterschiedlichen Dialekte zu vereinen wurde schließlich 1994 Common Lisp, als die erste objektorientierte Programmiersprache überhaupt, standardisiert (ANSI X3.266-1994). Heute wird Common Lisp neben Scheme vor allem an Amerikanischen Universitäten in den Einführungskursen gelehrt, was die guten Top-Seller Platzierungen von Lisp-Literatur bei Amazon, sowie Rang 14 auf einer Liste der beliebtesten Programmiersprachen (Tiobe Software) erklärt.
Wenn im Folgenden von Lisp die Rede ist, ist meist Common Lisp gemeint, wobei im speziellen die Implementierung "Allegro Common Lisp" von Franz Inc. verwendet wurde.

2 Was unterscheidet Lisp von anderen Programmiersprachen?

> "Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot." - Eric Raymond, "How to Become a Hacker"

Worin unterscheiden sich Programmiersprachen? Die gängigsten sind alle turing-vollständig, was bedeutet, dass man damit alle Funktionen berechnen kann, die auch von einer Turingmaschine gelößt werden können. Sie sind also im Prinzip alle "gleich mächtig". Für Paul Graham, einen einflussreichen Lisp-Programmierer und Autor, liegt der Unterschied nur darin, wieviel Aufwand man betreiben muss, um Zugang zu einem bestimmten Problem zu bekommen [Graham, 1996].
Wenn jemand bisher nur mit Assemblersprache gearbeitet hat und dann auf eine höhere Programmiersprache wie C umsteigt, dann wird er nach einer gewissen Eingewöhnungszeit nicht mehr zurück wollen. Genauso verhält es sich beim Umstieg von C auf Lisp. Wenn man die Konzepte erst einmal verinnerlicht hat, dann erweitern sie den Horizont eines jeden Programmierers. Umgekehrt fühlt man sich bei der Rückkehr zu C möglicherweise eingeschränkt.

- Lisp ist eine funktionale Programmiersprache

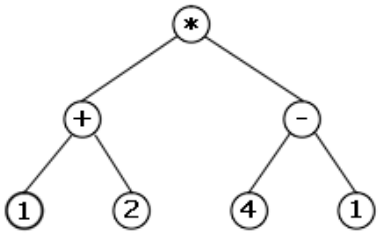
Das bedeutet, dass für Berechnungen immer Funktionen ausgewertet werden. Dabei werden im Fall von Lisp nicht Zahlen oder Zeichenketten sondern Symbole verarbeitet. Grundlage dieses Programmierparadigmas ist das λ-Kalkül (siehe Vortrag vom 22.11.2006). Im Gegensatz dazu besteht ein Programm, das im imperativen Programmierstil geschrieben wurde, aus einer Aneinanderreihung von Befehlen. Lisp ist nicht rein funktional, Seiteneffekte sind durchaus möglich.

- **Lis**t **P**rocessing vs. (**L**ots of ((**I**rritating **S**uperfluous) (**P**arentheses)))

Wenn man Lisp Quelltext betrachtet fallen sofort die vielen Klammern auf. Dabei handelt es sich um sogenannte S-Expressions, eine Notation um Programmcode und Daten gleichermaßen darzustellen. Lisp

verdankt seine Flexibilität der Tatsache, dass sich Code wie eine Datenstruktur behandeln lässt - man kann ihn also zur Laufzeit des Programms verändern. S-Expressions verwenden die Präfix-Notation um eine Baumstruktur mit Hilfe von Listen darstellen zu können. Diese Klammern wirken am Anfang etwas verwirrend, wenn man den Code aber richtig formatiert und einen Editor mit Syntaxhighlighting verwendet, fallen sie nach einer Weile nicht mehr auf.

Beispiel S-Expression: (* (+ 1 2) (- 4 1))



- Lisp ist typfrei

Variablen, Parameter und Funktionsergebnisse haben in einer typischen imperativen Sprache einen bestimmten Typ. Beispielsweise sind ganze Zahlen meist vom Typ Integer. Das vereinfacht und beschleunigt die Ausführung eines Programms, da der Compiler im Voraus weiß, von welchem Typ ein bestimmter Wert ist und entsprechend effizienter Maschinencode erzeugt werden kann. Eine Typüberprüfung zur Laufzeit wird überflüssig. Ausserdem stellt die Typisierung die Korrektheit des Programms sicher. Auf der anderen Seite beschränkt eine strikte Typisierung den Programmierer und läuft der mathematischen Vorstellung von Variablen zuwider. Dort kann eine Variable x einmal eine natürliche Zahl sein und kurz darauf eine rationale. Lisp verbindet die Freiheiten untypisierter Variablen mit den Vorteilen strikter Typisierung, indem es zusätzlich zum Standarddatentyp Symbol weitere Datentypen wie Zahlen, Strings, Booleans usw. bereitstellt, die zwar nicht unbedingt notwendig sind, aber die Effizienz erhöhen. Zur Verdeutlichung soll folgendes Beispiel dienen, dass einige Konzepte vorwegnimmt:

```
> (defun plus-untyped(x y)
    (+ x y))

> (defun plus-typed(x y)
    (declare (fixnum x y))
    (the fixnum(+ x y)))
```

Die erste untypisierte Funktion plus-untyped nimmt alle Argumente, die von der Funktion '+' akzepiert werden. Dazu zählen alle Untertypen von number wie z.B. integer, float und complex. Die Funktion plus-typed hingegen erlaubt nur Argumente vom Typ fixnum, einem Untertyp von integer. Auch der Rückgabewert wird hier auf fixnum festgelegt. Dadurch hat der Compiler genügend Informationen um performanten Maschinencode zu erzeugen und ggf. Typfehler anzuzeigen.

- Lisp ist eine Interpretersprache

Ein Programm, das in einer Compilersprache geschrieben wurde, wird erst nach der Fertigstellung in ausführbaren Maschinencode übersetzt. Dabei wird der gesamte Quelltext ausführlich untersucht und optimiert, um eine möglichst hohe Ausführungsgeschwindigkeit zu erreichen. Im Gegensatz dazu wird in Lisp jeder einzelne Ausdruck, sobald man ihn in den sogenannten Top-Level eingegeben hat, sofort ausgewertet. Die interaktive "read-eval-print-loop" (REPL) wartet bis ein Ausdruck eingegeben wird, liest ihn ein, wandelt ihn in eine interne Darstellung (verkettete Liste) um, evaluiert ihn und gibt das Ergebnis zurück. In allen folgenden Codebeispielen wird der Top-Level mit einem vorangestellten > gekennzeichnet. Es ist nicht ganz richtig, die REPL als einen Interpreter zu bezeichen, da in Common Lisp dynamisch kompiliert wird, das heißt es existiert ein compile-Aufruf, der den eingegebenen Quelltext, Funktion für Funktion, zur Laufzeit in Maschinencode übersetzt. Ein reiner Interpreter würde den übersetzten Programmcode sofort wieder "vergessen", wodurch keine Optimierungsmöglichkeit mehr bestünde. Neben dem Interpretermodus bieten die meisten Implementierungen zusätzlich die Möglichkeit,

den gesamten Quelltext auf einmal zu kompilieren. Damit wird eine Ausführungsgeschwindigkeit erreicht, die sich je nach Problem durchaus mit anderen Sprachen messen lassen kann. Die Vorteile dieser interaktiven Programmierumgebung liegen auf der Hand. Man kann damit jede einzelne Funktion, sobald sie entworfen wurde auf ihre Korrektheit hin testen. Damit erspart man sich eine langwierige Fehlersuche, wie sie bei reinen Compilersprachen immer wieder vorkommen. Ein weiterer Vorteil davon ist, dass man relativ schnell ein funktionierendes Programm zur Hand hat, das man einem Auftraggeber vorzeigen kann. Somit gewinnt man kostengünstig wichtige Aussagen über die Machbarkeit eines Projektes - ein Grund dafür, warum sich auch die NASA für Lisp interessiert.[1] Lisp ist wie geschaffen für dieses auch als Rapid Prototyping bezeichnete Konzept.

- "Lisp is a programmable programming language" [2]

Mit Hilfe von Makros sind Lisp-Programme in der Lage, neuen Lisp-Code zu erzeugen. Im Gegensatz zu gewöhnlichen Lisp-Funktionen, die als Eingabe Werte erhalten und auch Werte wieder zurück geben, erwarten Lisp-Makros Lisp-Code als Eingabe und geben ein verändertes Programm zurück. Diese Transformation geschieht nicht erst zur Laufzeit des Programms, sondern schon bei der Übersetzung. Auf der einen Seite eröffnet dieses Merkmal ungeahnte Möglichkeiten, nicht nur im Bereich der Künstlichen Intelligenz, wird aber auf der anderen Seite von vielen Programmierern abgelehnt, weil es besonderer Vorsicht bei der Anwendung bedarf.
Lisp stellt zwar bereits viele Funktionalitäten standardmäßig zur Verfügung, lässt dem Programmierer jedoch immer die Freiheit, neue Sprachelemente dem Kanon hinzuzufügen. Dabei werden neue Funktionen mit Hilfe grundlegenderer Funktionen definiert. Um die Argumente untereinander auszutauschen werden Listen und Makros verwendet. In anderen Programmiersprachen wie C ist dies nur durch die umständliche Definition abstrakter Datentypen möglich. Damit ist es beispielsweise möglich neue Kontrollstrukturen wie `while` auf Basis des primitiveren `do` auf effiziente Weise zu erzeugen. Nur 25 Lisp-Befehle sind (oft aus Effizienzgründen) spezielle Operatoren, der Rest besteht aus in Lisp geschriebenen Funktionen und Makros. Bei dieser Art des bottum-up-designs verändert sich die Programmiersprache zunehmend in Richtung des zu entwickelnden Programms. Nicht zuletzt wegen dieser Erweiterbarkeit hat Lisp fast 50 Jahre auf dem Buckel und ist damit neben FORTRAN die älteste noch lebendige Programmiersprache.

3 Syntax: Forms

Wie bereits erwähnt, wird die Lisp-Syntax mit S-Expressions beschrieben, die eine Art Syntaxbaum darstellen (ähnlich dem Code einer anderen Programmiersprache *nach* dem Parsen). S-Expressions bestehen entweder aus Atomen oder aus Listen. Eine Liste kann wiederum aus einzelnen Atomen oder weiteren verschachtelten Listen bestehen. Aber S-Expresions sind nicht gleich ausführbarer Code, da auf die selbe Art und Weise auch Daten dargestellt werden.
Lisp-Forms sind S-Expressions, die zusätzlich auch evaluiert werden können. Beispielsweise genügt die Liste `(1 2 3)` den Anforderungen einer S-Expression, kann aber im Gegensatz zur Form `(+ 1 2 3)` nicht evaluiert werden, da eine Funktion fehlt. Die Auswertung von Forms erfolgt immer rekursiv. Dabei werden die Argumente der Funktion von links nach rechts ausgewertet und an die Funktion übergeben. Bei der Evaluation der Form `(* (+ 1 2) (- 4 1))` wird also zunächst `(+ 1 2)` dann `(- 4 1)` ausgewertet und die Ergebnisse 3 und 3 an die Funktion * weitergegeben.
Um zu verhindern, dass ein Lisp-Ausdruck evaluiert wird, verwendet man die Special Form `quote`. `(quote (+ 1 2))` evaluiert zu (+ 1 2). Da `quote` relativ häufig benötigt wird, gibt es dafür die Abkürzung '. Dank Syntactic Sugar kann man also stattdessen auch `'(+ 1 2)` schreiben.

3.1 Atome

Man kann Atome in zwei unterschiedliche Typen aufteilen:

- Selbstevaluierende Atome

Atome wie Zahlen und Strings werden immer zu sich selbst evaluiert.
Common Lisp unterscheidet verschiedene Arten von Zahlen, die optional mit beliebiger Genauigkeit (limitiert nur durch die Hardware) angegeben werden können.

[1] Franz Inc. über die Pathfinder Mission der NASA
[2] John Foderaro, Communications of the ACM, September 1991

Dazu gehören:
Ganze Zahlen (z.B. `1`), Fließkommazahlen (z.B. `1.5`), Brüche (z.B. `1/2`) und komplexe Zahlen (z.B. `#C(2.0 1.0)`)

Strings werden mit Anführungszeichen begrenzt. (z.B. `"Hallo Welt!"`) Da die Lisp-Spezifikation älter als ASCII ist, ist der Zeichenvorrat für Strings nicht genau definiert. Die meisten Implementierungen verwenden jedoch Unicode.

- Symbole

Symbole können in Lisp dazu verwendet werden andere Objekte zu bezeichnen. Ähnlich den Variablen anderer Programmiersprachen haben Symbole einen Namen und einen Wert. Dabei dient die Funktion `setf` der Zuweisung von Variablen. Sie erwartet einen Speicherort und eine Form als Argument. In diesem Fall den Symbolnamen und den Wert.

```
> (setf x 5)
5
```

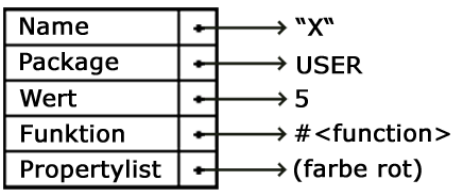
Symbole leisten aber mehr als nur eine bestimme Adresse im Speicher zu referenzieren. Beim Erstellen eines neuen Symbols wird eine Struktur im Speicher angelegt, die verschiedene andere Objekte referenziert. So verweist jedes Symbol nicht nur auf seinen Wert, sondern unter anderem auch auf ein Package. Packages sind Symboltabellen, die Lisp verwendet, um seperate Namensräume zu schaffen. Dabei wird jedem Symbol ein bestimmtes Package zugeordnet und ist somit standardmäßig nur von diesem aktiven Package aus erreichbar. Um auf Variablen oder Funktionen aus einem anderen Package zugreifen zu können, muss man entweder das aktive Package wechseln oder das benötigte Symbol explizit exportieren. Damit ist sichergestellt, dass auch bei größeren Projekten kein Variablenname doppelt vorkommt.
Falls ein Symbol der Name einer Funktion ist, dann wird auf diese in der sogenannten function cell des Symbols verwiesen. Über die Funktion `symbol-function`, die ein Symbol erwartet, in Kombination mit `setf` kann eine neue Funktion definiert werden.

```
> (setf(symbol-function 'addiere-zwei) #'(lambda (x) (+ x 2)))
> (addiere-zwei 5)
7
```

Die anonyme Funktion `(+ x 2)` wird mit der function cell des Symbols addiere-zwei verknüpft. (siehe λ-Expression auf Seite 12) `#'` ist syntactic sugar für die special form `function`, die ein Funktionsobjekt zurückgibt.
Tatsächlich definiert man Funktionen im Allgemeinen mit dem einfacheren Schlüsselwort `defun`, siehe dazu Punkt 6 auf Seite 11.
Jedes Symbol referenziert ausserdem eine Eigenschaftsliste (propertylist) in der Daten gespeichert werden können. Dabei kann mit Hilfe von `(get 'symbol 'key)` auf einen bestimmten Wert zugegriffen und mit `setf` verändert werden.

```
> (get 'x 'farbe)
NIL
> (setf (get 'x 'farbe) 'rot)
> (get 'x 'farbe)
ROT
```

Struktur eines Symbols



3.2 Listen

Neben den Atomen besteht die Lisp-Syntax aus Listen. Es gibt drei unterschiedliche Arten von Ausdrücken, die als Listen dargestellt werden und beliebig verschachtelt sein können.

- Function-Call-Form
  Es handelt sich dabei um ganz gewöhnliche Funktionsaufrufe, wie sie bereits verwendet wurden.
- Special-Form
  Es gibt eine Reihe von speziellen Funktionen die vom Interpreter/Compiler extra behandelt werden. Die Evaluationsreihenfolge der Argumente kann im Unterschied zu normalen Funktionsaufrufen variieren. Zu den Special Forms zählen beispielsweise Ausdrücke, die Variablen und Funktionen definieren. Aber auch `quote`, das die Evaluation eines nachfolgenden Ausdrucks verhindert und Kontrollstrukturen wie `if` gehören dazu.
- Macro-Form
  Auch die bereits erwähnten Makros weichen von der Evaluationsregel gewöhnlicher Funktionen ab. Sie werden in Punkt 8 auf Seite 16 besprochen.

4 Datenstrukturen

Wie der Name **LIS**t **P**rocessing schon andeutet, sind Listen eine grundlegende Datenstruktur in Lisp. Moderne Implementierungen bieten heute aber eine Vielzahl von Datenstrukturen wie Arrays oder Hashtabellen an.

4.1 Listen

Seit den Anfangszeiten greifen Lisp-Programmierer gern auf Listen zurück, da einige recht komfortable Funktionen zur Listenverarbeitung bereitgestellt werden. Obwohl Listen im Vergleich mit anderen Datenstrukturen oft langsam sind, eignen sie sich besonders gut dafür, einfache Prototypen zu erstellen. Listen können mit der Funktion `cons` erzeugt werden. `cons` erwartet zwei Argumente, wobei das zweite bereits eine Liste sein muss, zu dem das erste Argument hinzugefügt wird.

```
> (cons 'a '(b c))
(A B C)
```

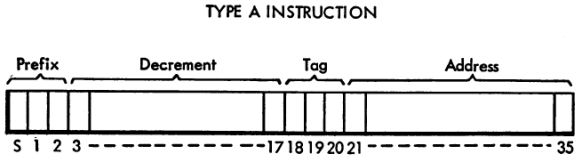
Wenn man Listen von Grund auf neu erzeugen will, fügt man ein Element an eine leere Liste an.

```
> (cons 'a (cons 'b (cons 'c nil)))
(A B C)
```

In diesem Fall bietet sich auch die Funktion `list` an.

```
> (list 'a 'b 'c)
(A B C)
```

Der Zugriff auf die einzelnen Elemente einer Liste erfolgt durch die Funktionen `car` und `cdr`. Manche Implementierungen bieten die equivalenten Funktionen `first` und `rest` an. Die Namen stammen noch aus Zeiten des IBM 704, dessen Instruktionsformat aus einem 36 Bit Wort bestand, das in zwei 15 Bit lange Bereiche, den address part und den decrement part aufgeteilt wurde. `car` bedeutete demnach **c**ontents of **a**ddr ess of **r**egister und verhielt sich wie ein Zeiger auf das (erste) Element einer Liste. `cdr` stand für **c**ontents of **d**ecrement of **r**egister und referenzierte ein 36 Bit Wort im Speicher, das wiederum in `car` und `cdr` unterteilt werden konnte.



Quelle: IBM 704 Manual

`car` gibt das erste Element einer Liste zurück.

```
> (car '(a b c))
A
```

`cdr` gibt den Rest hinter dem ersten Element zurück.

```
> (cdr '(a b c))
(B C)
```

Wenn man beide Funktionen entsprechend kombiniert, kann man jedes Element einer Liste erreichen.

```
> (car (cdr (cdr '(a b c))))
C
```

Mit den Makros `first`, `second`, `third`, ..., `tenth` kann man die ersten 10 Positionen direkt ansprechen.

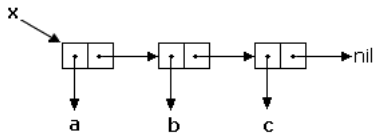
```
> (third '(a b c))
C
```

Die Funktion `dolist` wird dazu genutzt über eine Liste zu iterieren. Dabei wird bei jedem Schritt der Iteration das aktuelle Element der Liste an eine Variable gebunden, auf die innerhalb des Schleifenkörpers zugegriffen werden kann.

```
> (dolist (elem '(a b c)) (print elem))
A
B
C
NIL
```

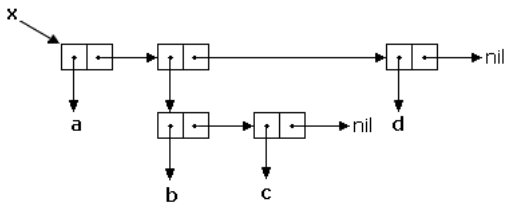
Lisp kommt ohne explizite Zeiger aus. Eine einfach verkettete Liste wird mit Hilfe von sogenannten Cons-Zellen, einem Paar von Zeigern realisiert. Eine eigentliche Liste ist entweder `nil` oder eine Cons-Zelle, deren `cdr` eine eigentliche Liste ist. Der erste Teil einer Cons-Zelle verweist also auf ein Element der Liste und der zweite Teil referenziert die nächste Cons-Zelle, deren erster Teil wiederum auf ein Element zeigt.

```
> (setf x '(a b c))
(A B C)
```



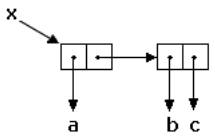
Eine Liste kann Elemente beliebigen Typs enthalten. Wenn es sich bei den Elementen wiederum um Listen handelt, entsteht eine Baumstruktur, wobei `car` den linken Unterbaum und `cdr` den rechten Unterbaum darstellt.

```
> (setf x (list 'a (list 'b 'c) 'd))
(A (B C) D)
```



Der zweite Teil einer Cons-Zelle muss nicht `nil` sein. Lisp unterscheidet zwischen "eigentlichen Listen" und "Listen mit Punkt" (dotted lists). Letztere kommt zum Einsatz, wenn man eine Struktur mit einer festgelegten Anzahl von Elementen benötigt.

```
> (setf x (cons 'a (cons 'b 'c)))
(A B . C)
```



4.2 Arrays

Um ein Beispiel dafür zu geben, dass Lisp mehr als nur Listen beherrscht, seien kurz Arrays vorgestellt. Im Gegensatz zu den Listen erfolgt der Zugriff auf die Elemente eines Arrays nicht sequentiell sondern indexbasiert, was Geschwindigkeitsvorteile bringen kann. Um ein Array zu erzeugen, wird die Funktion `make-array` mit der Dimension als Argument aufgerufen. Arrays können bis zu sieben Dimensionen mit je 1023 Elementen haben. Mit dem Argument `initial-element` wird bestimmt, mit welchem Wert das Array initialisiert wird. Ein 2x3 Array kann folgendermaßen erstellt werden.

> (setf mein-array (make-array '(2 3) :initial-element nil))
#2a((nil nil nil) (nil nil nil))

Um auf ein bestimmtes Element aus dem Array zuzugreifen, benutzt man die Funktion `aref`. Die als Argument erwartete Position des gesuchten Elements wird von 0 an gezählt.

```
> (aref mein-array 0 0)
NIL
```

Geschrieben wird wieder in Kombination mit `setf`.

```
> (setf (aref mein-array 0 0) 'x)
X
> mein-array
#2a((X NIL NIL) (NIL NIL NIL))
```

Man kann ein Array auch direkt beschreiben, indem man die Syntax $\#n$a wie in der Ausgabe der obigen Zeile benutzt. n steht dabei für die Dimension des Arrays.

```
> (setf mein-array2 #2a((y nil nil) (nil nil nil)))
#2a((Y NIL NIL) (NIL NIL NIL))
```

5 Variablen

Lisp unterscheidet zwischen lexikalischen und dynamischen Variablen.

5.1 Lexikalische Variablen

Lexikalische Variablen verhalten sich ähnlich wie lokale Variablen in anderen Programmiersprachen. Sie sind nur innerhalb der Umgebung sichtbar, in der sie auch deklariert wurden. Dabei kann es sich um einen Funktionsblock oder eine `let`-Anweisung handeln.
`let` gliedert sich in zwei Teile. Das Argument ist eine Liste, die weitere Listen der Form (variable wert) enthält. Hier können mehrere Variablen deklariert werden. Der zweite Teil besteht aus einem Rumpf in dem beliebige Funktionen ausgeführt werden können. Nur innerhalb dieses Rumpfes bleiben die Variablen gültig.

```
> (let ((x 5))
       (print x))
5
> x
Error: Attempt to take the value of unbound variable 'X'.
```

5.2 Dynamischen Variablen

Dynamische Variablen sind grob betrachtet globale Variablen, die, einmal deklariert, von überall her eingesehen und verändert werden können.

Die mit Hilfe der Funktionen `defvar` und `defparameter` erzeugten dynamischen Variablen werden "special" deklariert und sollten konventionsgemäß mit zwei * versehen werden. Der einzige Unterschied zwischen den beiden Funktionen besteht darin, dass `defvar` eine bereits bestehende Variable nur dann überschreiben kann, wenn sie noch nicht initialisiert ist.

```
> (defparameter *globalevariable* 5)
*GLOBALEVARIABLE*
> *globalevariable*
5
```

Einer globalen Variable kann lokal ein anderer Wert zugewiesen werden, indem sie entweder in einer `let`-Umgebung neu deklariert, oder einer Funktion als Parameter übergeben wird. Die alte Variablenbindung wird durch die neue Bindungsumgebung überdeckt. Man spricht von Verschattung. Werden Funktionen oder `let`-Anweisungen verschachtelt, so wird jedesmal eine neue Bindungsumgebung erzeugt, die erst wieder gelöscht wird,wenn der Block abgearbeitet ist. Man kann sich dieses Verhalten wie einen Stack vorstellen, auf den bei jeder weiteren Verschachtelung die alten Bindungsumgebungen gepusht werden. `let` erkennt ob es sich um eine lexikalische oder dynamische Variable handelt und erzeugt entsprechend eine lexikalische oder eine dynamische Bindung.

```
> (defparameter *x* 10)
  (defun get-x () (print *x*))
  (get-x)
10
> (let ((*x* 20))
       (get-x))
20
```

Alle Variablen, die innerhalb einer `let`-Form deklariert sind, werden gleichzeitig gebunden. Das bedeutet, dass diese sich nicht gegenseitig referenzieren können, da es keinen zeitlichen Ablauf gibt. Für diesen Fall gibt es die Variante `let*`, die sich genauso verhält wie `let`, aber alle Argumente hintereinander abarbeitet.

```
> (let ((x 3)
        (y x))
       ;Die Ausgabefunktion format funktioniert ähnlich wie das Pendant aus C
       (format t "Variable x = ~A ~%Variable y = ~A" x y))
Error: Attempt to take the value of the unbound variable 'X'.

> (let* ((x 3)
         (y x))
       (format t "Variable x = ~A ~%Variable y = ~A" x y))
Variable x = 3
Variable y = 3
```

Das universelle `setf` kann sowohl lexikalische als auch dynamische Variablen zuweisen. Es ist jedoch kein guter Stil auf `defvar` und `defparameter` zu verzichten, weil der Code damit schlechter lesbar ist.

```
> (let ((y 5)) ;lexikalische Variable
       (setf y 10)
       (print y))
10

> (setf *foo* t) ;dynamische Vaiable
T
```

Wenn möglich sollten lexikalische Variablen benutzt werden, da sie wieder verschwinden, sobald die Funktion abgearbeitet ist. Dynamische Variablen bleiben im Gegensatz dazu die ganze Zeit über erhalten und verbrauchen somit auch dann noch Speicherplatz, wenn die Variable gar nicht mehr gebraucht wird. Hier sieht man auch den in Abschnitt 3.1 auf Seite 6 erläuterten Zusammenhang zwischen Symbolen und dynamischen Variablen. Während dynamische Variablen immer von einem Symbol referenziert werden und umgekehrt, haben lexikalische Variablen nach dem kompilieren keine Verbindung mehr zu einem Symbol. Stattdessen existiert nur noch ein Zeiger auf die Adresse im Speicher, an der sich der Wert der Variablen befindet.

5.3 Closures

Nehmen wir an, innerhalb einer let-Anweisung sei sowohl eine lexikalische Variable, als auch eine Funktion, die die Variable verändert, deklariert. Die Variable ist also im Gegensatz zur Funktion von "aussen" nicht zu sehen. Was passiert nun, wenn die Funktion ggf. mehrmals aufgerufen wird? Eine solche Funktion wird Closure genannt, da sie sich alle Variablenbindungen in ihrem lexikalischen Kontext merkt und ihn somit "abschließt".

```
> (let ((counter 0))
       (defun inc () (setf counter (+ counter 1)))
       (defun reset () (setf counter 0)))
> (inc)
1
> (inc)
2
> (reset)
0
> (inc)
1
```

6 Funktionen

Das Rückgrat einer funktionalen Sprache sind natürlich Funktionen. Lisp besteht zum großen Teil aus einer Sammlung von Funktionen. Jede weitere Funktion, die neu definiert wird, erweitert den Umfang der Sprache, da es keine Unterscheidung zwischen eingebauten und selbstdefinierten Funktionen gibt.
Jede Funktion besitzt einen Rückgabewert, der an andere Funktionen weitergereicht werden kann. Mit Hilfe von Seiteneffekten ist Lisp aber in der Lage, diesen strengen Zusammenhang zu durchbrechen. `setf` ist ein Beispiel für eine Funktion, die mehr macht als nur einen Wert zurückzugeben. Da Seiteneffekte aber potenzielle Fehlerquellen sind, sollte ihre Verwendung auf ein Minimum begrenzt werden.
Im Abschnitt über Symbole haben wir gesehen, dass ein Symbol über seine function cell eine Funktion referenzieren kann und dieser als Namensträger dient. Funktionen werden mit dem Schlüsselwort `defun` (define function) definiert. `defun` erwartet mehrere Argumente. Das erste stellt den Namen der Funktion dar, das zweite ist eine Liste von Parametern und das dritte Argument ist ein Rumpf von Anweisungen. Das Beispiel der Funktion, die zu einem beliebigen Wert die Zahl 2 addiert, kann also auch folgendermaßen angegeben werden:

```
> (defun addiere-zwei (x)
       (+ x 2))
> (addiere-zwei 5)
7
```

6.1 Lokale Funktionen

Alle bisher aufgetretenen Funktionen sind globale Funktionen. Ähnlich dynamischen Variablen sind sie von überall her aufrufbar. In Analogie zu den Variablen kann man auch lokale Funktionen definieren, die nur innerhalb eines bestimmten Kontextes gültig sind und eventuell existierende gleichnamige Funktionen verschatten. Das `let` - Pendant für Funktionen heißt `labels`.

```
> (defun inc (x)
       (+ x 1))
> (defun add3 (x)
       (labels ((inc (x) (+ x 3)))
            (inc x)))
> (inc 3)
4
> (add 3)
6
```

6.2 Rekursion

Da Lisp den Anspruch hat das λ- Kalkül umzusetzen, darf natürlich die Rekursion nicht fehlen, denn diese lässt sich direkt aus (Y F) = (F (Y F)) ablesen. (Stichwort: Fixpunktkombinator Y) . Sie bietet sich vor allem dann an, wenn eine Liste als Datenstruktur zugrunde liegt, da bei einem rekursiven Aufruf jeweils der Rest der Liste als Parameter beim erneuten Aufruf der Funktion mit übergeben werden kann. Wenn es sich dabei um eine Endrekusion handelt, das heißt, wenn der rekursive Aufruf das letzte ist was die Funktion bearbeitet, dann lässt sich der Ausdruck in eine effinzientere Iteration umwandeln. Da dies beim kompilieren automatisch geschieht, kann man sich darauf konzentrieren, auf den ersten Blick nicht-endrekursive Funktionen in endrekursive Varianten umzuwandeln. Dies kann z.B. mit Hilfe von Akkumulatoren, die als Parameter mitgeschleift werden, bewerkstelligt werden. Eine Funktion, die jedes Element einer Liste der Reihe nach ausgibt, kann rekursiv auf einfache Art und Weise definiert werden:

```
> (defun print-elements (list)
    (if (null list)                        ;Abbruchbedingung wenn die Liste leer ist
        (return-from print-elements nil)) ;beendet die Auswertung und gibt nil zurück
     (print (car list))
     (print-elements (cdr list)))
> (print-elements '(a b c))
A
B
C
NIL
```

6.3 λ- Expression

Um möglichst nah an A. Churchs Notation des Lambda-Kalküls zu sein, entwickelte McCarthy eine Representation um Funktionen nicht benennen zu müssen. Eine Funktion, die ihr Argument um zwei erhöht wird im Lambda-Kalkül als λx. x + 2 beschrieben. In Lisp sieht die gleiche Funktion folgendermaßen aus: (lambda (x) (+ x 2))
Anonyme Funktionen werden meist zur Konstruktion komplexerer Funktionen verwendet, die aus mehreren einfachen Funktionen aufgebaut sind. Es wäre störend und ausserdem eine Verschwendung von Ressourcen, wenn jede dieser simplen Funktionen einen Namen tragen müsste.

6.4 Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Parameter erwarten oder als Ergebnis liefern, also Funktionen wie Werte behandeln, werden als Funktionen höherer Ordnung bezeichnet. Da in Lisp auch Funktionen Objekte sind und es zudem keine syntaktischen Unterscheidungen zwischen Funktionen und Daten gibt, muss es eine Möglichkeit geben "leblosen" Code in eine ausführbare Funktion umzuwandeln. Diesen Zweck erfüllt `apply` bzw. `funcall`. Sie erwarten eine Funktion und eine Reihe von Argumenten, die der Funktion übergeben werden sollen. Die Anzahl der Argumente muss dabei nicht im Voraus bekannt sein. Dies

ermöglich die Generierung einer Liste mit Argumenten zur Laufzeit. Der Unterschied zwischen `apply` und `funcall` ist rein syntaktischer Natur.

```
> (apply #'+ '(1 2 3))
6
> (funcall #'+ 1 2 3)
6
```

Ein weiterer Vertreter von Funktionen höherer Ordnung ist `mapcar`, das eine Funktion auf jedes Element einer oder mehrerer angegebener Listen anwendet.

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

6.5 Parameterlisten optionale, rest parameter keywords

Beim Aufruf einer Funktion werden meist Argumente mit übergeben. Wenn der Funktion aber eine falsche Anzahl an Parametern übergeben wird, meldet der Interpreter einen Fehler. Parameterlisten dienen dazu diese Limitierung aufzuheben. Zusätzlich zu den normalen und bereits bekannten required parameters gibt es in Lisp Optionale-, Rest- und Keyword-Parameter. Optionale Parameter kommen dann zum Einsatz, wenn eine Funktion nur in Speziallfällen alle Argumente braucht und sonst mit einem Standardwert zufrieden ist. Alle Argumente, die hinter dem Symbol &optional stehen, gelten als zusätzliche Parameter.

```
> (defun person (vorname nachname &optional (alter 'unbekannt))
      (format t "Die Person heißt ~s ~s. Das Alter der Person ist: ~s" vorname nachname alter))
> (person 'erika 'mustermann)
"Die Person heißt ERIKA MUSTERMANN. Das Alter der Person ist: UNBEKANNT"
> (person 'erika 'mustermann '45)
"Die Person heißt ERIKA MUSTERMANN. Das Alter der Person ist: 45"
```

Mit &rest eingeleite Argumente werden als Liste an eine einzige Variable gebunden. Diese Liste kann dann entweder der Reihe nach abgearbeitet oder einer anderen Funktion übergeben werden.

```
> (defun count-arguments (&rest list) (length list))
> (count-arguments 'a 'b 2 3 'c)
5
```

Keyword-Parameter dienen der sicheren Identifizierung eines optionalen Parameters. Da bei &optional die Argumente nur aufgrund ihrer Position innerhalb der Parameterliste zugeordnet werden können, muss die Reihenfolge immer sichergestellt sein. Im Gegensatz dazu werden Argumente, die nach &key einer Funktion übergeben werden, mit einem Schlüsselwort identifiziert.

```
> (defun list-in-order (&key first second third)
      (list first second third))
> (list-in-order :third 'kiesinger :first 'adenauer :second 'erhard)
(ADENAUER  ERHARD KIESINGER)
```

7 Kontrollstrukturen

7.1 Booleans, Prädikate und Gleichheit

Boolesche Werte werden in Lisp durch die Symbole T für true und NIL für false dargestellt. Beide Symbole evaluieren zu sich selbst. Funktionen, die boolesche Werte zurückgeben, werden Prädikate genannt.
Die Funktion listp beispielsweise gibt T zurück, wenn das Argument eine Liste ist, anderenfalls NIL.

```
> (listp '(a b c))
T
> (listp '5)
NIL
```

Es ist Konvention, dass Prädikate immer auf p oder ? enden und selbsterklärend sein sollten. Weitere gebräuchliche Prädikate sind:

atomp	Überprüft ob das Argument ein Atom ist.
numberp	Überprüft ob das Argument eine Zahl ist
symbolp	Überprüft ob das Argument ein Symbol ist
stringp	Überprüft ob das Argument ein String ist
consp	Überprüft ob das Argument eine aus cons-Zellen bestehende Liste ist
zerop	Überprüft ob das Argument 0 ist
plusp	Überprüft ob das Argument eine positive Zahl ist
minusp	Überprüft ob das Argument eine negative Zahl ist
evenp	Überprüft ob das Argument eine gerade Zahl ist
oddp	Überprüft ob das Argument eine ungerade Zahl ist
null	Überprüft ob das Argument eine leere Liste ist
endp	Überprüft ob das Argument (das zwingend eine Liste sein muss) eine leere Liste ist

Es gibt ausserdem eine Reihe von Funktionen, die die Gleichheit von Objekten bestimmen:

`eq` prüft ob zwei Lisp-Objekte identisch sind, also an der selben Stelle im Speicher stehen.

```
> (eq 'a 'a)
T
```

`eql` testet die Gleichheit zweier Zahlen vom selben Typ

```
> (eql 1 1.0)
NIL
```

= überprüft die Gleichheit von Zahlen unabhängig von deren Typ.

```
> (= 1 1.0)
T
```

`equal` testet Listen auf die Gleichheit ihrere Elemente.

```
> (equal '(a b c) '(a b c))
T
```

Die booleschen Operatoren `AND` und `OR` berechnen ihre Argumente jeweils nur solange wie unbedingt notwendig. Sobald ein Argument von `AND nil` ergibt, bricht die Berechnung mit `nil` ab, da der gesamte Ausdruck nicht mehr T werden kann. Umgekehrt geschieht genau das gleiche bei `OR`.

```
> (AND (= 1 1) (= 1 2))
NIL
> (OR (listp '(a b c)) (= 1 2))
T
```

`NOT` erwartet nur ein Argument und gibt dessen Wahrheitswert invertiert zurück.

```
> (NOT (= 1 1))
NIL
```

7.2 Conditional Statements

Lisp war die erste Sprache, die die heute nicht mehr wegzudenkenden conditional statements einführte. Die einfachste Kontrollstruktur dürfte wohl das bekannte if-then-else sein. Als erstes Argument muss ein Testausdruck übergeben werden. Ist dieser wahr, wird der folgende then-Ausdruck ausgewertet. Anderenfalls wird mit dem dritten else-Ausdruck fortgefahren.

```
> (if (> 4 3) (print 'richtig) (print 'falsch))
RICHTIG
```

Die Verallgemeinerung von `case` aus anderen Sprachen heißt in Lisp `cond`. Die aufgelisteten Testausdrücke werden der Reihe nach ausgewertet bis ein Ergebnis T ist. In diesem Fall werden alle folgenden Anweisungen ausgeführt, wobei der letzte Ausdruck gleichzeitig den Rückgabewert der Funktion darstellt. Wenn man nicht möchte, dass `cond nil` zurückgibt, muss immer mindestens ein Testausdruck wahr sein. Dazu hängt man an das Ende, anstelle eines Testausdrucks, ein T, verbunden mit den entsprechenden Anweisungen für den else-Fall.

```
> (cond (<test-1> <S-expression> <S-expression> ... <S-expression> <Rückgabewert-1>)
        (<test-2> <S-expression> <S-expression> ... <S-expression> <Rückgabewert-2>)
        ...
        (<T> <S-expression> <S-expression> ... <S-expression> <Rückgabewert-else>))
```

7.3 Iteration

Da sich mit Hilfe von Rekusion bei weitem nicht jedes Problem effizient lösen lässt, gibt es in Lisp natürlich auch die Möglichkeit einen Anweisungsblock wiederholt zu durchlaufen. Interessanterweise gibt es unter den 25 Spezialoperatoren in Lisp keine Funktion, die sich für eine Iteration eignet. Der grundlegende Schleifenoperator `do` ist ein Makro und setzt sich aus einer geschickten Nutzung lokaler Variablen und interner GOTOs zusammen. `do` erwartet als erstes Argument eine Liste mit Variablenspzifikationen. Jede Variablenspezifikation hat dabei die Form (variable startwert schrittweite). Das zweite Argument beinhaltet einen Testausdruck um die Iteration zu beenden und einen entprechenden optionalen Rückgabewert. Das dritte Argument ist der Schleifenkörper, der bei jedem Iterationsschritt durchlaufen wird.

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

Es gibt auch etwas weniger komplizierte, dafür aber unflexiblere Konstrukte um Schleifen zu erzeugen. Um über Listen zu iterieren, benutzt man gewöhnlich das bereits bekannte `dolist`. Wenn eine einzige Schleifenvariable ausreicht, kann statt `do` auch auf das einfache `dotimes` zurück gegriffen werden. Die Variable wird dabei immer mit 0 initialisiert. Das zweite Argument stellt die Anzahl der Durchläufe dar und mit dem dritten optionalen Argument kann der letzte Rückgabewert bestimmt werden.

```
> (dotimes (i 3 'ende)
      (format t "~A~%" i))
0
1
2
ENDE
```

Lisp stellt eine Vielzahl weiterer Iterationskonstrukte wie `while, for` und `loop` zur Verfügung, die, wenn sie nicht bereits implementiert wären, auch selbst hätten erstellt werden könnten. Siehe Punkt 8 über Makros auf der nächsten Seite.

7.4 Blöcke

Die if-Kontrollstruktur ist sehr unflexibel, da immer nur eine Anweisung ausgewertet wird. Wenn beispielsweise für den Fall dass die Bedingung zutrifft, zwei unterschiedliche Aktionen ausgelößt werden sollen, so interpretiert Lisp dies nicht als zwei Befehle der then-form sondern führt nur den ersten aus, da der zweite zum else-Ausdruck gehört. Da es keine geschweiften Klammern wie in C gibt, um einen Anweisungsblock zu begrenzen, greift man in diesem Fall auf `progn` zurück. Alle Ausdrücke innerhalb einer `progn` Funktion werden der Reihe nach ausgewertet.

```
> (if (> 4 3)
      (progn
         (print 'richtig)
         (setf var 5))
      (print 'falsch))
RICHTIG
5
```

Neben `progn` gibt es noch zwei andere Schlüsselwörter um Codeblöcke zu erstellen. `block` und `tagbody` haben mit `progn` gemein, dass sie Seiteneffekte erzeugen können. Blöcke die mit `block` eingeleitet werden können zusätzlich an jedem beliebigen Punkt wieder verlassen werden. Falls der Block einen Namen hat geschieht dies mit `(return-from <name> <rückgabewert>).` Ein Block, der den Namen nil trägt, kann mit `(return <rückgabewert>)` verlassen werden. Da viele Funktionen automatisch einen Block mit Namen nil erzeugen, können auch sie unterbrochen werden. Ein Beispiel dafür ist `dolist`.

```
> (dolist (x '(a b c d e f))
       (format t "~A " x)
       (if (eql x 'c)
          (return 'done)))
A B C
DONE
```

Ein mit `tagbody` begrenzter Codeblock hat die Besonderheit, dass darin die berühmt berüchtigten GOTOs vewendet werden können. Aufgrund der Neigung zu Spaghetticode wird diese Blockart fast ausschließlich nur zur Definition anderer (Schleifen-)Operatoren benutzt.

8 Makros

Nachdem andere Programmiersprachen viele Konzepte von Lisp kopiert haben und umgekehrt neue Methoden und Paradigmen in Lisp umgesetzt wurden, sind es vor allem die Makros, die Lisp einzigartig machen. Dabei kann es leicht zu begrifflichen Mißverständnissen kommen. Lisp-Makros haben nicht viel mit C-Makros und fast nichts mit Visual Basic for Applications zu tun. Es handelt sich dabei nicht nur um kurze, eventuell wiederkehrende Codefragmente, die eine bestimmte Stelle des eigentlichen Programms automatisiert ersetzen, sondern um mächtige Funktionen, die direkt den Programmcode verändern und dessen Auswertung beeinflussen können. Damit lässt sich die Sprache erweitern, ohne dass Änderungen am Compiler vorgenommen werden müssen.

Makros werden mit `defmacro` definiert. Genau wie Funktionen haben sie einen Namen, Parameterlisten und einen Funktionskörper. Aber im Gegensatz zu diesen bewirken sie nichts direkt, sondern werden noch vor dem Kompilieren in Programmcode transformiert, der später etwas bewirkt. Ein simples Makro, das sein Argument auf `nil` setzt, sieht folgendermaßen aus:

```
> (defmacro set-nil (var)
       (list 'setf var nil))
> (set-nil bla)
NIL
```

Sobald nun der Parser beim Durchlaufen des Programmcodes auf das Makro `(set-nil bla)` stößt, wird anhand der Makrodefinition der Ersetzungstext konstruiert. In diesem Fall wird `(list 'setf var nil)` zu `(setf bla nil)` transformiert, man spricht von Makroexpansion. Im zweiten Schritt wird der neu eingefügte Code anstelle des ursprünglichen Makros evaluiert. Dabei ist es möglich, dass ein expandiertes Makro, ähnlich wie eine russische Matroschkapuppe, wieder einen Makroaufruf enthält. Obwohl es sich bei der Expansion in diesem Beispiel um eine gewöhnliche Evaluation von `list` handelt, macht es Sinn, zwischen Evaluation und Expansion zu unterscheiden, da zuerst die Codetransformation im Vordergrund steht, während es im zweiten Schritt um die eigentliche Auswertung geht. Zusammenfassend lässt sich sagen, dass Makros im Gegensatz zu Funktionen keine Ergebnisse zurückgeben, sondern S-Expressions liefern, die später evaluiert werden.

Es macht wenig Sinn ein Makro wie `set-nil` zu definieren, da eine gewöhnliche Funktion dafür ausreicht. Um komplexere Sprachelemente zu konstruieren, bedarf es allerdings noch weiterer Hilfsmittel. Während der Makroexpansion wird eine Liste mit Programmsyntax erzeugt, deren Elemente vom Makro mit Hilfe von `list` konkateniert wurden. Um die Übersichtlichkeit zu wahren, greift man üblicherweise auf ‘ (Backquote) zurück, das sich genauso verhält wie ' (Quote) und die Evaluation verhindert. Statt `(list 'a 'b 'c)` reicht ‘`(a b c)` was das gleiche ist wie `'(a b c).` Man kann die Evaluaton allerdings wieder anschalten indem man ein , (Komma) vor die entsprechenden Elemente der Liste setzt. Man erhält Schablonen, mit denen sich bequem Makrodefinitionen erstellen lassen, die fast so aussehen wie ihre Expansion.

```
> (setf b 5)
5
> `(a ,b c)
(A 5 C)
> (defmacro set-nil (var)
       `(setf ,var nil))
```

,@ (Komma-at) verhält sich ähnlich wie , (Komma) mit dem Unterschied, dass es nur auf Listen angewendet werden kann und deren Elemente zur Auswertung freigibt.

```
> (setf lst '(a b c))
(A B C)
> `(lst ist ,lst)
(LST IST (A B C))
> `(Die Elemente von lst sind ,@lst)
(DIE ELEMENTE VON LST SIND A B C)
```

Damit kann man sich jetzt auch an komplexere Makros wagen. Die Kontrollstruktur `for`, die eine angegebene Anzahl von Iterationen über einem Schleifenkörper ausführt, könnte folgendermaßen definiert werden:

```
> (defmacro for ((var start stop ) &rest body)
       `(do ((,var ,start (incf ,var))
          (limit ,stop))
          ((> ,var limit))
          ,@body))
> (for (x 1 5) (format t "~A " x))
1 2 3 4 5
NIL
```

Auf den ersten Blick scheint diese Definition von `for` zu funktionieren.
Aber warum führt `(for (limit 1 5) (format t "~A " limit))` zu einer Endlosschleife? Die Antwort liegt im resultierenden Code versteckt, zu dem das Makro expandiert. Mit `macroexpand-1` kann man die Makroexpansion zum Debuggen betrachten.

```
> (macroexpand-1 '(for (limit 1 5) (format t "~A " limit)))
(DO ((LIMIT 1 (INCF LIMIT))
       (LIMIT 5))
       ((> LIMIT LIMIT))
       (FORMAT T "~A " LIMIT))
```

Jetzt wird deutlich, dass ein Namenskonflikt besteht, da `limit` zwei Mal innerhalb der do-Schleife vorkommt. Dieses Problem kann behoben werden, indem innerhalb des Makros ein neues Symbol erstellt wird, das in der Umgebung der Expansion garantiert nicht existiert. Diese Aufgabe erledigt die Funktion `gensym`. Da das neu erzeugte Symbol zu keinem Package gehört ist sichergestellt, dass es kein reguläres Symbol gibt, das zu einem `gensym` identisch ist. Eine fehlerfreie Implementierung von `for` sieht folgendermaßen aus.

```
> (defmacro for ((var start stop ) &rest body)
       (let ((uniquelimit (gensym)))
       `(do ((,var ,start (incf ,var))
          (uniquelimit ,stop))
          ((> ,var uniquelimit))
          ,@body)))
```

Es gibt beim Makrodesign noch einige weitere Stolpersteine zu beachten. Die Möglichkeit Lisp mit Hilfe von Makros zu erweitern ist vergleichbar mit dem Verändern des Compilers selbst. Es erfordert eine andere Denkweise als beim Definieren von Funktionen, da man die Transformation im Voraus mitbedenken muss. An diesem Punkt scheiden sich die Geister. Auf der einen Seite bieten Makros dem Programmierer ungeahnte Freiheiten, halten aber andererseits auch eine Menge unerwarteter Fehlerquellen bereit.

9 Objektorientiertes Programmieren mit CLOS

Ende der 70er Jahre begann sich das Paradigma der Objektorientierung langsam aber sicher durchzusetzen, woraufhin verschiedene Projekte mit dem Ziel entstanden, Lisp um diesen Ansatz zu erweitern. 1994 wurde mit dem Common Lisp Object System (CLOS) ein ANSI-Standard geschaffen, der die verschiedenen Implementierungen miteinander verschmolz. Da CLOS ein integraler Bestandteil von Common Lisp ist, unterscheidet es sich grundlegend von anderen objektorientierten Programmiersprachen wie C++ und Java. Beispielsweise kann die Definition einer Klasse dynamisch, d.h. zur Laufzeit, geändert werden.

9.1 Klassen und Instanzen

Klassen werden mit der Funktion `defclass` erzeugt. Da CLOS multiple Vererbung unterstützt, kann eine Liste mit Klassen von denen sie erbt angegeben werden. Im Hauptteil der Funktion werden die Slots, in anderen Sprachen Instanzvariablen oderAttribute genannt, definiert.

```
>(defclass kreis (oberklasse)
  (radius))
```

Nun kann mit `make-instance` eine neue Instanz der Klasse erstellt werden. Mit `slot-value` kann jederzeit auf die Slots zugegriffen werden.

```
>(setf x (make-instance 'kreis)

>(setf (slot-value x 'radius) 3)
3
```

Es gibt die Möglichkeit für jeden Slot einen `:accessor` zu definieren, so dass man `slot-value` nicht mehr benötigt. Will man auf einen Slot nur lesend bzw. schreibend zugreifen, so kann `:accessor` durch `:reader` bzw. `:writer` ersetzt werden.

```
>(defclass kreis ()
  (radius :accessor get-radius))

>(setf x (make-instance 'kreis)

>(setf (get-radius x) 3)
3
```

Um einen Slot beim Erzeugen einer Instanz initialisieren zu können, muss das Argument `:initarg` definiert sein. Wurde kein Wert für den Slot angegeben, kann mit `:initform` ein Default-Wert festgelegt werden.

```
>(defclass kreis ()
  (radius :accessor get-radius
          :initarg :radius
          :initform 1))

>(setf x (make-instance 'kreis)
>(get-radius x)
1

>(setf x (make-instance 'kreis :radius 3)
>(get-radius x)
3
```

9.2 Generische Funktionen

Im Gegensatz zum Nachrichtenmodell gehören Methoden in CLOS nicht zu bestimmten Klassen und können auch nicht vererbt werden. Methoden werden mit Hilfe von generischer Funktionen, die extra für das jeweilige Objekt spezialisiert werden, realisiert. In Java oder C++ wird immer genau eine Nachricht an ein Objekt "geschickt". In CLOS kann man dagegen beliebig viele Objekte erreichen, d.h. generische Funktionen bilden eine Obermenge zum Nachrichtenmodell. Wenn man möchte, könnte man sich also sein eigenes Message-Passing nachbilden, indem man eine neue Funktion send einführt. `x.getRadius();` würde dann folgendermaßen aussehen: `(send x 'get-radius)`während er in CLOS so aussieht: `(get-radius x)`
In LOOPS, einem Vorläufer von CLOS, wurde genau das gemacht. Dadurch dass sich der Aufruf einer Methode aber syntaktisch von dem einer Funktion unterscheidet, kommt es zu einer Inkonsistenz zur funktionalen Programmierung. Um den Radius aller Kreise in einer Liste zu bestimmen müsste man umständlich schreiben:
`(mapcar #'(lambda (kreis) (send kreis 'get-radius)) kreis-liste)`
In CLOS kann man Funktionen höherer Ordnung wie gewohnt benutzen.
`(mapcar #'get-radius kreis-liste)`

Methoden werden mit dem Makro `defmethod` erzeugt. Dabei kann Namensgleichheit durchaus erwünscht sein. Indem Objekte als Argument übergeben werden, wird die Methode weiter spezialisiert. Beim Aufruf einer Methode mit einem bestimmten Argument wird immer diejenige Methode gewählt, die "am besten" spezialisiert ist. Mit Hilfe der Funktion `call-next-method` kann aber auch zur nächst speziellsten Funktion gesprungen werden.

```
>(defmethod area ((x circle))
  ;Methode für Kreise
  (* pi (expt (get-radius x) 2)) ;r^2*pi
  )

(defmethod area ((y rectangle))
  ;Methode für Rechtecke
  (* (get-height y)(get-width y)) ;height*width
  )

(defmethod area ((x circle)(y rectangle))
  ;Summiert die Flächen eines Kreises und eines Rechtecks
  (+ (* pi (expt (get-radius x) 2)) (* (get-width y) (get-height y)))
  )
```

9.3 Hilfsmethoden

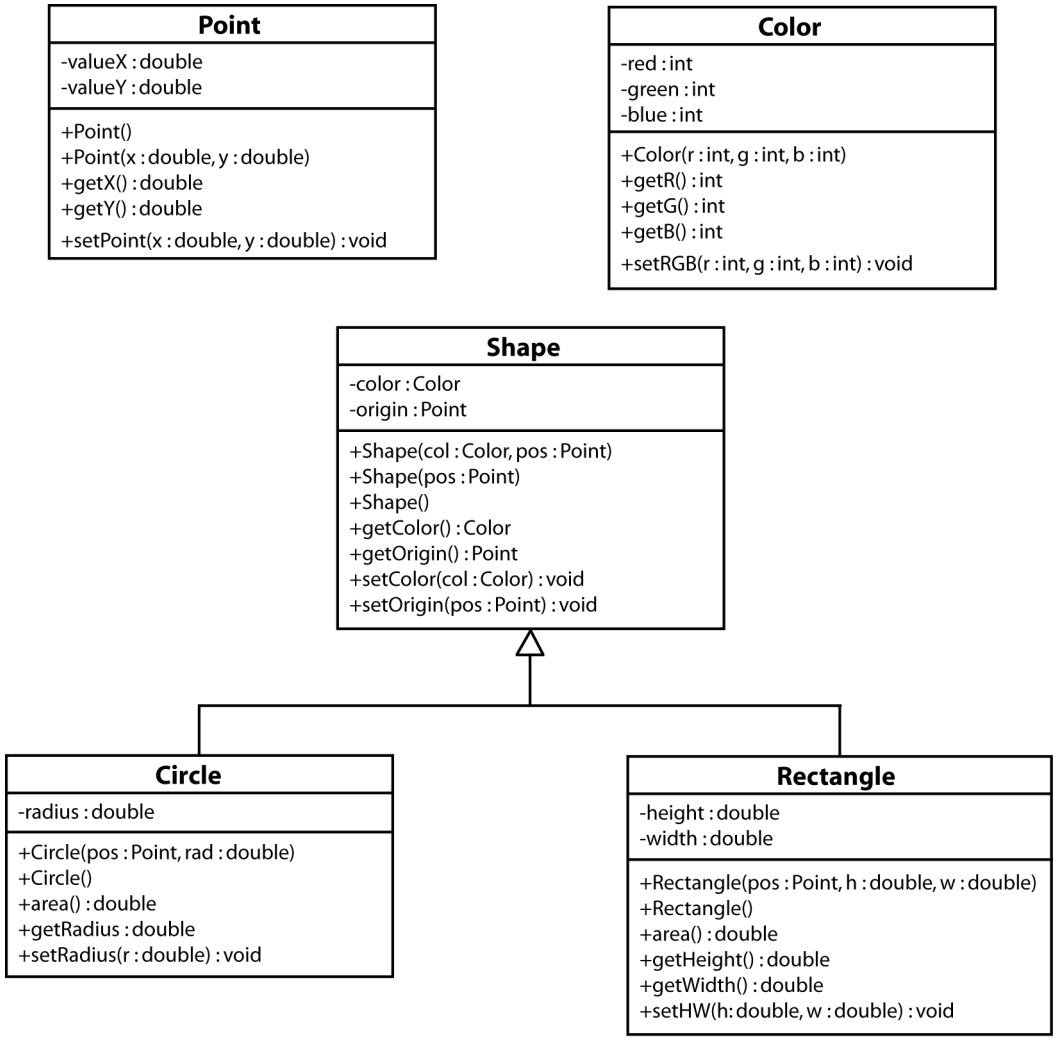
Bei den bisher benutzen Methoden handelt es sich um primäre Methoden. Zusätzlich können die Hilfsmethoden :before, :after und :around definiert werden. Ist eine :before-Methode vorhanden, so wird sie immer vor der eigentlichen Methode ausgeführt. Das ist nützlich, wenn vor der eigentlichen Berechnung noch vorbereitende Maßnahmen anstehen. Um nach dem Aufruf der primären Methode ergänzende Schritte zu unternehmen kann die :after Methode definiert werden. Die :around-Methode wird anstelle der normalen Primärmethode aufgerufen und kann z.B. dazu genutzt werden, lexikalische Variabeln zu binden. Anschließend ruft sie selbst die primäre Methode auf.

```
>(defmethod area :before ((x circle))
  ((format t "Gleich wird die Fläche des Kreises berechnet werden")
  )

>(defmethod area :after ((x circle))
  ((format t "Das war also die Kreisfläche")
  )
```

9.4 Beispiel

Das folgende Beispiel soll einen Überblick über den objektorientierten Ansatz in Lisp liefern. Da sich die Philosophie von Lisp, dem Programmierer so viele Freiheiten wie möglich zu lassen, sich nicht besonders mit den üblichen Restriktionen verträgt, werden bestimmte Funktionalitäten nicht umgesetzt. Besonders deutlich wird dies bei der Sichtbarkeit von Attributen und Methoden. Es gibt keine direkte Umsetzung der z.B. aus Java bekannten Deklarationen wie public, private und protected. Das bedeutet nicht, dass es überhaupt keine Kapselung gibt. Um den Zugriff auf einen Slot einzuschränken, kann er z.B. in ein seperates Package gesteckt werden. Die gewünschte Kapselung wird erreicht, indem der Name der Zugriffsfunktion exportiert wird, der Name des Slots aber nicht. Diese Lösung ist allerdings nicht konsequent, da mit `slot-value` immer noch auf den Slot zugegriffen werden kann.



Mögliche Realisierung in CLOS:

```
(defclass point ()
  ((valueX :type double-float
           :accessor get-x
           :initarg  :x
           :initform 0.0)
   (valueY :type double-float
           :accessor get-y
           :initarg  :y
           :initform 0.0))
  )
```

```
;;Diese Funktion ist überflüssig, da mit :accessor bereits
;;eine Zugriffsfunktion definiert wurde.
(defmethod set-point ((pos point)(x double-float)(y double-float))
  (setf (get-x pos) x)
  (setf (get-y pos) y)
  )

(defclass color ()
  ((red :type integer
        :accessor get-r
        :initarg  :r
        )
   (green :type integer
          :accessor get-g
          :initarg  :g
          )
   (blue :type integer
         :accessor get-b
         :initarg  :b
         ))
  )

;;Auch diese Funktion ist nicht unbedingt notwendig
(defmethod set-rgb ((col color2)(r integer)(g integer)(b integer))
  (setf (get-r col) r)
  (setf (get-g col) g)
  (setf (get-b col) b)
  )

(defclass shape ()
  ((color :type integer
          :accessor get-color
          :initarg  :color
          :initform (make-instance 'color :r 0 :g 0 :b 0)
          )
   (origin :type point
           :accessor get-origin
           :initarg  :origin
           :initform (make-instance 'point)
           ))
  )

;;nicht notwendig
(defmethod set-color ((x shape)(col color2))
  (setf (get-color x) col)
  )
;;nicht notwendig
(defmethod set-origin ((x shape)(pos Point))
  (setf (get-origin x) pos)
  )

(defclass circle (shape)
  ((radius :type double-float
           :accessor get-radius
           :initarg  :radius
           :initform 0.0))
  )
```

```
(defmethod area ((x circle))
  (* pi (expt (get-radius x) 2))
  )

(defclass rectangle (shape)
  ((height :type double-float
           :accessor get-height
           :initarg  :h
           :initform 0.0)
   (width :type double-float
          :accessor get-width
          :initarg  :w
          :initform 0.0))
  )

(defmethod area ((x rectangle))
  (* (get-width x) (get-height x))
  )

;;Instanziierung
>(setf mycircle (make-instance 'circle
                 :origin (make-instance 'point :x 2 :y 2)
                 :color (make-instance 'color :r 128 :g 64 :b 255)
                 :radius 5))

>(get-r (get-color mycircle))
128
>(area mycircle)
78.53981634
```

Es macht eigentlich keinen Sinn die Klasse Shape aus dem Beispiel zu instanziieren, da es ein solches Objekt in der Realität nicht gibt. In Java würde man diese Klasse deshalb als abstract definieren. CLOS stellt dieses Feature standardmäßig jedoch nicht zur Verfügung. Da CLOS aber auf einem Metaobjekt-Protokoll (MOP) aufbaut, das Verhalten eines CLOS-Programms also auf internen CLOS Klassen und Methoden basiert, ist es möglich dies (und noch viel mehr) durch eine nachträgliche Änderung dieser sogenannten Metaobjekte in CLOS zu realisieren. Während das Hinzufügen zusätzlicher Metaobjekte, wie beispielsweise einer abstrakten Klasse, im Allgemeinen keine Probleme bereitet, kann die direkte Manipulation bestehender Metaobjekte zu Insellösungen führen.

10 Fazit

Lisp hat in den vergangenen 50 Jahren viele andere Programmiersprachen kommen und gehen sehen und ist doch selbst nie in die Jahre gekommen. Der Grund dafür ist die enorme Anpassungsfähigkeit, die es Lisp erlaubt sich immer wieder neu zu erfinden. Diese Fähigkeit prädestiniert Lisp für unkonventionelle, domänespezifische Lösungen, ist aber gleichzeitig dafür verantwortlich, dass es nie zum Mainstream wurde. Erschwerend kommt hinzu, dass die Softwaretechnik eine andere Philosophie vertritt und versucht der zunehmenden Komplexität von Software mit Reglemetierungen und dem Benutzen von ausgetretenen Pfaden entgegenzuwirken. Das bedeutet nicht, dass Lisp sich dieser Entwicklung komplett verschließt. CLOS wurde eingeführt, weil sich damit strukturierter programmieren lässt und mit der integrierten Typhierarchie wurde eine Reihe von möglichen Fehlerquellen eliminiert. Aber das Grundproblem bleibt. Mächtige Konzepte wie Makros und das Metaobjekt Protokoll werden von der Mehrheit der Programmierer geschmäht, obwohl sie nach Meinung von Befürwortern den Horizont erweitern und die Kreativität fördern. Dass das nicht immer so bleiben muss, zeigt ein Blick in die Vergangenheit. Viele Konzepte, die sich heute in anderen Programmiersprachen finden, haben ihren Ursprung in Lisp. Das liegt natürlich auch daran, dass Lisp einfach zuerst da war und man das Rad nicht so ohne weiteres neu erfinden kann. Es lässt sich aber auch ein allgemeiner Trend erkennen, dass konzeptionell eher statische Sprachen nachträglich auf dynamische Errungenschaften wie automatische Speicherbereinigung und Generische Programmierung setzen.

Mit dem Erfolg von Python, Pearl und Ruby hat Lisp zwar ernstzunehmende Konkurrenz in diesem Bereich bekommen, doch gibt es immer noch einen bedeutenden Unterschied - die Lisp-Syntax. Ohne die Listenstruktur des Programmcodes wären Makros nicht vorstellbar. Diese sind zwar ein zweischneidiges Schwert, aber komplett darauf zu verzichten kann bedeuten, mit einer stumpfen Waffe zu kämpfen.

"Lisp has jokingly been called "the most intelligent way to misuse a computer". I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts."
-Edsger Dijkstra, CACM

Literatur

[Graham, 1994] Graham, P. (1994). *On Lisp: Advanced Techniques for Common Lisp.* Prentice-Hall, Englewood Cliffs, NJ.

[Graham, 1996] Graham, P. (1996). *ANSI Common Lisp.* Prentice Hall, Upper Saddle River, New Jersey.

[IBM, 1955] IBM (1955). *704 electronic data-processing machine - manual of operation.* International Business Machines Corporation, New York, U.S.A.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.

[McCarthy, 1978] McCarthy, J. (1978). History of LISP. In Wexelblat, R. L., editor, *History of Programming Languages: Proceedings of the ACM SIGPLAN Conference*, pages 173–197. Academic Press. Published in 1981; 1978 is date of conference.

[Seibel, 2005] Seibel, P. (2005). *Practical Common Lisp.* Apress, pub-APRESS:adr.