Lisp in Perl

Ungefähr 1976 las ich in der Zeitschrift "Spektrum der Wissenschaft", Rubrik "Mathematische Knobeleien", einen Artikel von Douglas R. Hofstadter, der dem geneigten Leser auf spielerische Art die Programmiersprache Lisp beibrachte. Ich war geneigt. Mit dem Bleistift und unbeholfen löste ich die Übungsaufgaben. Und unbeleckt von jedem Vorwissen erkundigte ich mich sofort bei einem Thekennachbarn, Studiengang Elektrotechnik, nach einem programmierbaren Taschenrechner mit Lisp ...

In der Ära der Homecomputer erschien eine Lisp-Implementierung für das Wohnzimmerwunder C64. 1987 kaufte ich meinen ersten XT für sensationelle 2 Mille, Monitor grün, und selbstverständlich kam Xlisp drauf, preisgünstiger geht's nimmer. 1990 leistete ich mir eine legale Lizenz für PC-Scheme von Texas Instruments und stellte fest: Scheme ist mein Lieblingslisp. Denn es ist fix und vor allem einfach!

So einen sollte jeder haben!

Heute gibt es Perl5 als beinahe plattformunabhängige Skriptsprache. Läuft überall, kennt kaum Speichergrenzen, kann Daten referenzieren, ist umsonst, und vor allem: es macht ungefragt einen Garbage Collect, wenn Not im Speicher ist. – Ein Perl-Programm ist – nach Aussage des Autors Larry Wall – in der Laufzeit langsamer, aber in der Entwicklung schneller als C. – Setzen wir einen drauf und sagen: Ein Lisp-Programm ist in der Laufzeit langsamer, aber in der Entwicklung schneller als Perl, und begeben uns an die Feierabendaufgabe, Lisp in Perl zu implementieren. (Nichts gegen Perl. Es ist durch und durch prima. Aber warum nicht beides?)

Dieser Text dient als Begleitlektüre für den, der den Quelltext liest oder eigene Erweiterungen vornehmen will. Er ist auch für den interessierte Laien geeignet, der reinschnuppern will. Zum Verständnis brauchen Sie, lieber Leser, etwas Vorkenntnis über die Speicherverwaltung von Lisp – und Sie sollten ein bißchen Perl können.

Wir gehen nach der Strategie Bottom-Top vor. Wir beschränken uns auf die dokumentierten Fähigkeiten von Perl – wir vermeiden unverständlichen Hacks. Einen Knoten realisieren wir z. B. mit einer Referenz auf ein Feld mit

zwei Elementen. Im Dokument `perlref.html` erfahren Sie, wie Referenzen funktionieren.

Zum zitierten Quelltext in diesem Artikel: Sie finden hier eine verständlich aufbereitete Fassung der einzelnen Funktionen vor, die dem Verständnis zugute kommt. Im Quelltext steht meist eine Kurzfassung, die zugunsten des Durchsatzes auf private Variablen verzichtet, sich aber nicht so flüssig liest.

Wir orientieren uns am *Revised*^{4} *Report on Scheme*.

22. November 1999

Wolf Busch
Talstraße 10
35428 Oberkleen

Inhaltsverzeichnis

Kapitel 1

Implementierung

1.1 Die Shell

Beim Entwickeln verwenden wir *Projekt-Shell.pl*, so daß wir die Zwischenresultate interaktiv kontrollieren können:

```
print $prompt;
do { my $commandline = <STDIN>;
     my @result = eval ($commandline);
     print "=> " . (join (", ", @result)) . ";" ;
     print $@; # Fehlerstatus
     print "n$prompt";
} until eof;
```

Die Sorte von Shell kennen wir bereits von Lisp. Sie liest eine Eingabezeile, schickt sie durch die Routine `eval()` und beantwortet unsere Anfrage ohne Umschweife in der nächsten Zeile.

1.2 Anatomie eines Knotens

Lisp arbeitet mit Knoten. Perl arbeitet auf Wunsch mit Referenzen (symbolischen Zeigern) auf allerlei Daten. Einen Lisp-Knoten realisieren wir in Perl mit einer Referenz auf ein Feld (Array), das zwei Elemente hat. Element 1 ist das CAR, Element 2 das CDR. Beide Elemente müssen Skalare sein ("einfache" Werte, also eine Zahl, ein Wort, eine Referenz. Felder und Assoziativlisten scheiden aus.)

So bilden wir einen Primitivknoten:

```
sub ConsLowLevel {
     my $car = shift();
     my $cdr = shift();
     my @pair = ($car, $cdr);
```

```
    my $reference = \@pair;
    return($reference) };
```

Die privaten Skalare `$car` und `$cdr` nehmen die Funktionsargumente auf. Das private Feld `@pair` erhält die Skalare `$car` und `$cdr` als Elemente. Der private Skalar `$reference` nimmt die Referenz auf `@pair` auf. Die Funktion übergibt als Rückgabewert diese Referenz. – Den gleichen Effekt hat die kürzere Version `sub ConsLowLevel  @_` , die ohne private Variablen auskommt. Welche Version wir verwenden, ist Geschmackssache.

Und so sieht die Funktion aus, die den Primitiv-CAR ausliest:

```
sub CarLowLevel {
    my $reference = shift();
    my @pair = @$reference;
    my ($car, $cdr) = @pair;
    return $car };
```

Im Quelltext *node.pl* finden Sie die didaktisch nicht ganz so gehaltvolle Kurzfassung `sub CarLowLevel {$_[0]->[0]}`, deren Analyse wir uns hier schenken. Wer sie kryptisch findet, sei auf die Perl-Dokumentation verwiesen, insbesondere `perllol.html`.

1.3 Daten und Typen

In Pascal z. B. sind die Datentypen mit den Variablen verbunden, in Lisp nicht. Deswegen ist Lisp aber nicht typfrei, sondern verbindet die Datentypen direkt mit den Werten. Wir realisieren ein Datum als Primitivknoten, dessen CAR den Typ und dessen CDR den Wert des Datums enthält. Damit kann eine Rechenoperation zur Laufzeit überprüfen, ob sie auch die richtige Futtersorte hat.

Die Datentypen Symbol, Pair, ... werden durchnumeriert mit 1, 2, ..., und damit bauen wir die Funktionen, welche die Daten mitsamt Typ erzeugen:

```
sub MakeUninternedSymbol { &ConsLowLevel (1, @_) } ;
sub MakePair             { &ConsLowLevel (2, @_) } ;
# (...)
```

Die Typ-Kontrolle erfolgt dementsprechend mit Abfrage des Primitiv-CAR:

```
sub IsSymbol { &CarLowLevel (@_) == 1 } ;
sub IsPair { &CarLowLevel (@_) == 2 } ;
# (...;)
```

Die Wertabfrage des Datums geschieht entsprechend:

```
sub GetValue { &CdrLowLevel (@_) }
```

Was immer der Wert des Datums ist, er steht im CDR des Primitivknotens – sei es ein Skalar, dann darf er körperlich drin sein, oder sei es eine komplexe Datenstruktur, dann muß es eine Referenz darauf sein. Beim Datentyp Vektor z. B. ist der Wert des Datums eine Referenz auf ein Perl-Datenfeld.

Wir brauchen hier noch nicht zu bestimmen, wie die Datentypen im einzelnen zu handhaben sind – sie sind jederzeit feststellbar, und können wir die einzelnen Fälle später behandeln.

```
$apfel = &MakeSymbol("APPLE");
$F = &MakeBoolean("");
&IsBoolean($F)
```

⇒ 1

An dieser Stelle ein Einschnitt: nach dem 3. revidierten Bericht galt die Leere Liste `()` als falsch, nach dem 4. revidierten Bericht nicht mehr als Wahrheitswert und demzufolge – wie alle Werte außer `#f` – als wahr. Er erlaubt aber ausdrücklich, daß er je nach Implementierung dennoch wie `#f` (also wie falsch) behandelt werden darf. Dies Lisp nutzt die Freiheit und läßt ihn als falsch gelten. Trotzdem zählt er nicht zu den Wahrheitswerten, sondern erhält einen eigenen Datentyp namens `EMPTY-LIST`, wird konstruiert mit `&MakeNull` und geprüft mit `&IsNull`.

Zum Thema Symbole: wir wissen im Voraus, daß es in Lisp Symbole regnet. Um Engpässen vorzubeugen, legen wir das Feld `@SymTable` an, in dem die Symbole indiziert werden. Die Funktion `&MakeSymbol` greift auf diese Tabelle zu. – Ähnlich verfahren wir mit den Buchstaben (chars), aber aus anderen Gründen: so ist die Identitätsprüfung mit `eq?` leichter zu verwirklichen.

1.4 Listen

Der fundamentale Typ von zusammengesetzten Daten ist die Liste. Sie besteht aus einer Verkettung von Knoten. Das letzte Element ist die Leere Liste `nil`. Zusammengesetzt wird ein Knoten mit der Funktion `&CONS()`, die ein Lisp-Datum des Typs `PAIR` erzeugt. Im Quelltext *pair.pl* finden Sie die Kurzfassung des Perl-Codes, hier die etwas verständlichere Langfassung:

```
sub CONS {
     my $car = shift();
     my $cdr = shift();
```

```
     my $node = &ConsLowLevel($car,$cdr);
     my $pair = &MakePair($node);
     return $pair };
```

Dazu die Funktion:

```
sub CAR {
     my $pair = shift();
     my $node = &GetValue($pair);
     return &CarLowLevel($node) };
```

und dementsprechend die Funktion `&CDR()`, und ein Grundstock für die Listenarithmetik ist gelegt – na ja, noch nicht ganz. Was soll passieren, wenn wir aus Versehen die Funktion `&CAR()` auf ein Symbol anwenden, z. B. `$apfel` (⇒ `APPLE`)? Jawohl, genau richtig, eine Fehlermeldung!

```
sub CAR {
     my $pair = shift();
     &IsPair($pair)
          || die (&TypeMismatchMessage("CAR",
                                       "PAIR",
                                       $pair));
     my $node = &GetValue($pair);
     return &CarLowLevel($node) };
```

Die Perl-Funktion `die()` ist hier die Ultima Ratio: Datentyp falsch – will tot umfallen. Und weil das zu den Hauptfehlerquellen zählt, lohnt die Funktion `&TypeMismatchMessage()`, die den Absturz in simpler englischer Grammatik begründet, z. B. `CAR on non-PAIR 'APPLE' (is a SYMBOL)`. Sie ruft ihrerseits die Funktion `&PrintForm()` auf, die das Lisp-Datum ins Menschliche übersetzt.

Übrigens sind Listen teilbar (genau das tun die Funktionen `CAR` und `CDR`). Alles andere gilt als unteilbar. Darum gilt in Lisp alles, was nicht Liste ist, als Atom.

1.5 Einlesen der Benutzereingabe

Die klassische Benutzerschnittstelle von Lisp ist die Programmschleife, welche die Eingabe vom Benutzer einliest, berechnet und ausgibt (read-eval-print-loop, abgekürzt REP, hat nichts zu tun mit irgendwelchen politisch kackfarbenen Umtrieben.)

Als Grundlage brauchen wir einen dazu Parser, der aus einem übergebenen Text ein Lisp-Datum macht. Das geschieht mit Mustervergleich. Perl fährt dabei zu Höchstform auf.

Trotzdem müssen wir ein kleines bißchen "um die Ecke" denken. Falls der Parser sich rekursiv selbst aufrufen muß, muß er ständig auf dem Laufenden sein über das aktuelle Zwischenresultat (der abgearbeiteten Zeichenkette) und den Rest. Also übergibt der Parser als Resultat ein Feld, dessen erstes Element das teilberechnete Lisp-Datum ist; das zweite Element enthält den Rest der Kommandozeile, der noch abzuarbeiten bleibt. (Perl pur, diesmal keine Knoten!)

Der Parser erkennt aus der Benutzereingabe die Datentypen Buchstabe (*char*), Wort (*string*), Zahl (*number*), Wahrheitswert (*boolean*), Symbol (*symbol*), Vektor (*vector*) und Liste (*pair*). Die übrigen Datentypen sind nicht unmittelbar einzugeben, sondern entstehen bei Berechnungen in Lisp.

```
sub ParseAtom
{
     my $str = shift;
     my $port = shift;
     my @result;
     ($str, $port) = &input($str, $port);
     $str =~ s/^\s*//; # führende Leerzeichen entfernen
     if ($str eq '' && $port eq 'eof') # eof
     {
         @result = ($EOF, $str)
     }
     elsif ( $str =~ /^(#e|#i)?$NumberPattern/io ) # Number
     {
         my @arr = &ParseNumArr($&);
         @result = (&MakeNumber(\@arr), $')
     }
     # ... Buchstabe, Wort, ...
     elsif ($str =~ s/^'//)  # quote
     {
         my ($qval, $rest) = &ParseAtom($str);
         my $first = &MakeList(&MakeSymbol("QUOTE"), $qval);
         @result = ($first, $rest)
     }
     # ... Quasiquote, Unquote, Unquote-Splicing ...
     elsif ($str =~ s/^\(//) # Listen
     {
         @result = &ParseList($str)
     }
     else { die "Don't understand input '$str'" };
         # sonst stimmt was nicht.
     @result
}
```

Der Parser ist übrigens die Instanz, die das ’ Hochkomma als Abkürzung für `(quote ...)` behandelt, ebenso das umgekehrte ‘Hochkomma für `(quasiquote ...)`, das ,Komma für `(unquote ...)` und das ,@Komma mit Klammeraffen für `(unquote-splicing ...)`.

Die Funktion `&ParseAtom()` erhält als Argument die Benutzereingabe und den Port. Falls die Eingabe sich über mehrere Zeilen erstreckt (bei Listen und Vektoren), füllt sie mit `&input()` die Zeile wieder auf.

Die Funktion `&ParseList()` macht aus Klammerausdrücken Listen. Dabei ruft sie für die atomaren Elemente wiederum `&ParseAtom()` auf. Sie unterscheidet zwischen “ordentlichen” und “unordentlichen” Listen (solchen, deren innerstes CDR nicht `nil` ist), bietet aber ansonsten nicht viel neues und ist deshalb hier ausgelassen.

1.6 Interne Zahlendarstellung

Oben habe ich gesagt, daß wir uns erst bei Bedarf um die Behandlung der einzelnen Datentypen zu kümmern brauchen. Sie dürfen mich beim Wort nehmen, jetzt sind die Zahlen dran. – Das Zahlen-Vergleichsmuster fällt etwas länglich aus und wird deshalb aus mehreren Teilmustern zusammengesetzt, die letztlich die Standard-Zahleneingaben erkennen und für Lisp aufbereiten.

Wir stellen eine Zahl als Referenz auf ein Feld dar, dessen erstes Element den Zähler und dessen zweites den Nenner der Zahl enthält. – Der Scheme-Standard sieht komplexe Zahlen vor. Das Feld enthält also weitere zwei entsprechende Einträge für den imaginären Anteil.

Der Scheme-Standard sieht vor, daß die Zahlen *genau* und *ungenau* sein dürfen. Wenn wir eine Zahl in der Form `3/4+2/5i` eingeben, ruft der Parser unterm Strich die Funktion `&MakeNumber([3, 4, 2, 5])` auf. Die Zahl in diesem Beispiel ist genau. – *Ungenaue* Zahlen kennzeichnen wir durch den Nenner `0`, z. B. `&MakeNumber([0.75, 0, 0.4, 0])`. Das ist verwechslungssicher, denn bekanntlich darf nicht durch 0 dividiert werden.

Um die Genauigkeit durch möglichst viele Rechenoperationen hindurch zu retten, muß unser Lisp den Bruch bei Bedarf kürzen. Um z. B. die Zahl 48/56 darzustellen, ermittelt die Perl-Funktion `&ggT(48, 56)` deren größten gemeinsamen Teiler, also 8, und unser Lisp kürzt damit zu 6/7.

1.6.1 Großer Zahlenkompromiß

Die interne Zahlengenauigkeit in meinem verwendeten Perl ist mäßig. Als Kriterium für Genauigkeit verwende ich den Mustervergleich auf den Buchstaben e im Zahlwort (in Perl ist es Wurscht, ob eine Zahl als “richtige” Zahl oder als Wort eingesetzt ist.) Falls also der Buchstabe `e` auftaucht, gilt die Zahl als ungenau. Falls z. B. die Funktion `&ggT(1e45, 5)` darauf

stößt, warnt sie: `can't compute lcm(1e45, 5) - returning 1` – und tut es auch. Was soll's.

1.6.2 Kleiner Zahlenkompromiß

Intern rechnet dies Lisp mit der vollen verfügbaren Zahlengenauigkeit, aber bei der Bildschirmdarstellung der Zahlen wird die 14. Nachkommastelle kaufmännisch gerundet, so daß die Funktion `(cos (- pi))` auf dem Bildschirm `0.0` produziert, obwohl es bei meinem Perl `-1.22514845490862e-16` sind. Wenn die Rechenungenauigkeit genau wissen wollen, verwenden Sie die Funktion `(number->string)`. Wenn Sie die volle Wahrheit auf dem Bildschirm haben wollen, entfernen Sie in der Funktion `&NumberPrintForm()` den Funktionsaufruf `&ungefaehr()` (siehe Quelltext *pair.pl*). – Ach so, Kosinus und PI sind in diesem Lisp nicht drin, aber Sie können es definieren, das Werkzeug ist da.

Kapitel 2

Ausdrücke evaluieren

Wir haben die fundamentalen Lisp-Funktionen `&CONS()`, `&CAR()`, `&CDR()` ganz normal in Perl programmiert. Das läßt vermuten, daß Perl von sich aus imstande ist, Lisp-Programme auszuführen. Die Vermutung stimmt. Die Unterschiede sind rein syntaktischer Natur, und sie lassen sich an einer Hand aufzählen.

Die Funktion `&GeneratePerlSource($expr)` macht aus einem Lisp-Ausdruck einen Perl-Quelltext, den der Evaluierer `&EvalExpr()` der Perl-Funktion `eval()` zuführt. Die Details:

2.1 Symbolnamen

Lisp-Symbole schöpfen aus einem größeren Zeichenvorrat als Perl-Variablen. Perl kann zwar auch Variablennamen aus Sonderzeichen zusammenstellen, z. B. `$'---'`, aber nicht innerhalb von privaten Umgebungen via `my` – dort geht es strikt alphanumerisch zu. Also ersetzen wir jedes Sonderzeichen durch seinen ASCII-Wert, dreistellig, mit führendem Unterstrich. Zur Vermeidung von Kollisionen erhält jeder Variablenname zusätzlich einen führenden Unterstrich. Mit der Funktion `&Lisp2PerlName("+")` erzeugt der Quelltextgenerator z. B. aus dem Symbol `+` den Variablennamen `\${__043}`.

2.2 Konstanten

Wie kriegen wir ein normales Lisp-Datum in den Quelltext? Struktur aufdröseln bis zurück zum Quelltext? Es geht auch billiger. Die Funktion `&EvalExpr` erzeugt ein Datenfeld `@ConstTable`, und zwar mit dem Schlüsselwort `local`. Das Feld ist für alle von hier aufgerufenen Funktionen sichtbar, auch für `&GenerateQuotedSource`. Es fügt das Lisp-Datum am Ende ein, merkt sich die Position, z. B. 3, und übergibt z. B. den Perl-Ausdruck `$ConstT[3]`.

`&EvalExpr` wiederum umschließt seinen zurückerhaltenen Quelltext mit `{my @ConstT=@ConstTable; (Ausdruck)}`. Beim Kompilieren lernt der Code also die Tabelle "auswendig" und kann später darauf zurückgreifen.

2.3 Listen

Eine Liste ist normalerweise ein Funktionsaufruf, bei dem der *Wert* des ersten Listenelements den Funktionskörper darstellt. Der Quelltext `(list apfel birne)` muß im Prinzip zum Quelltext

```
do { my $function = ${_LIST|;
     my $code = $function->[1];
     @\_=(${_APFEL, ${_BIRNE});}
     &$code }}
```

umgeformt werden. (Das Perl-Feld `@_` nimmt die Parameter für Funktionen auf.) Die Perl-Variablen für die Funktion `list` und die Argumente `apfel`, `birne` werden gewonnen wie in Symbolnamen beschrieben.

Eine Feinheit kommt noch dazu. Falls der Ausdruck an letzter Stelle in der Spezialform `lambda` vorkommt, erfolgt ein unbedingter Sprung: `goto &$code`. Der Quelltextgenerator `&GeneratePerlSource()` erhält dazu einen optionalen zweiten Parameter `$tail`, der gegebenenfalls den sprechenden Wert `goto` erhält, und den er ungeprüft in den Quelltext reinkleben darf.

Auf die Art werden abschließende Selbstaufrufe von Funktionen ganz zwanglos zu Wiederholschleifen. – Im Quelltext steht `$f` und `$c` anstelle `$function` und `$code`, außerdem finden Sie einen Test auf den Datentyp "Funktion", den Sie übrigens auskommentieren dürfen – Perl beschwert von alleine, falls es kein echter Code ist. Aber Perl sagt Ihnen nicht, welches Datum an der Pleite schuld war.

2.4 Spezialformen

Ohne die Spezialform ist kein Lisp lebensfähig. Zu den fundamentalen zählt die wenig beachtete QUOTE, die sich hinter dem `'`Hochkomma versteckt.

Für die Spezialformen legen wir eine Assoziativliste `%SpecialSource` an, deren Schlüssel die Symbolnamen und deren Werte Quelltextgeneratoren sind.

Unser Quelltextgenerator `&GeneratePerlSource` muß so erweitert werden, daß er bei einer Liste feststellt, ob sie eine Spezialform bildet, und zwar mit `$SpecialSource{&GetSymbolName(&CAR($expr))}`.

2.5 Quote

Die `QUOTE` behandelt ihre Argumente sämtlich mit `&GenerateQuotedSource`, insbesondere auch Symbole und Listen. – Die Spezialform `set!` überschreibt eine Variable dann, wenn diese existiert – falls nein, Fehlermeldung. –

Theoretisch könnten wir auch `define` als Spezialform anlegen. Tun wir aber nicht, das gibt'n Makro. Zum Definieren verwenden wir `**define**`. Steht in keinem Standard, ist aber erforderlich.

2.6 Lazy Evaluation

Zu den Spezialformen zählen u. a. IF, OR, AND. Diese evaluieren ihre Ausdrücke nachträglich bei Bedarf (lazy evaluation). Beispiel:

```
Lisp> (or "so wahr ich hier stehe" was für'n Quatsch)
```

⇒ `"so wahr ich hier stehe"`

`OR` hört auf zu rechnen, sobald es auf einen wahren Ausdruck stößt – also einen, der nicht zu `#f` evaluiert. Dahinter darf jeder Quatsch der Welt stehen, z. B. auch gerne mal ein Symbol, das gar nicht definiert ist. Pfusch? Nein, sondern seriöse Kurzschlußlogik.

2.7 LAMBDA

`LAMBDA` ist in Scheme nichts weiter als eine Spezialform, die eine Funktion erzeugt. Aus dem Ausdruck `(lambda (x) (list x x))` macht sie (im Prinzip) den Quelltext

```
[7, sub
    {
          do
          {
              my $function = ${\_LIST};
              my $code = $function->[1];
              my $_X = (shift());
              &$code(${_X}, ${_X});
          }
    }
]
```

plus Kontrolle auf korrekte Argumentenzahl etc. Dazu ruft es für die Elemente im Funktionskörper jedesmal `&GeneratePerlSource()` auf.

Außerdem ist `lambda` kompetent für Entscheidungen über Endrekursion. Wenn es ans Ende des Funktionskörpers stößt, übergibt es dem Quelltext-Operator als optionales 2. Argument `goto` mit. Siehe *lambda.pl*.

2.8 Primitive Funktionen

Betrachten wir eine Primitive Funktion – also eine, die wir nicht in Lisp realiseren können oder wollen – z. B. list:

```
$_LIST = &MakeFunction(\&MakeList)
```

(Die Funktion `&MakeList` gibts schon, die gehört zu den Butter-und-Brot-Funktionen in unserem Lisp.) Kürzer geht's kaum. Probe? Probe:

```
Lisp> (list 'apfel'birne)
```

⇒ `(apfel birne)`

Eine weitere Vereinfachung bildet die Hilfsfunktion `&deffunc("name", sub { ...})`, die den Funktionsnamen nach Perl übersetzt, und die von sich aus weiß, daß der Wert vom Typ Funktion sein soll. – Einige Funktionen, z. B. Vergleichsfunktionen, definieren wir nicht einzeln, sondern am Stück in einer `for`-Schleife. Das ist kürzer und weniger tippfehleranfällig.

2.8.1 Die vier Grundrechenarten

Das *Multiplizieren* zweier rationaler Zahlen ist trivial (Zähler mal Zähler durch Nenner mal Nenner und kürzen). Zum *Addieren* werden die Brüche gleichnamig gemacht, die Zähler addiert und die Summe anschließend ebenfalls gekürzt. Die Funktion

```
&add2($z1, $n1, $z2, $n2)
```

addiert und

```
&mul2($z1, $n1, $z2, $n2)
```

multipliziert Zähler/Nenner-Paare unter Berücksichtigung der Genauigkeit, das heißt: wenn die eine Zahl ungenau ist, wird die andere ungenau gemacht, und nur die Zähler werden berechnet. (Ausnahme: falls eine genaue `0` mit einer ungenauen Zahl multipliziert wird, kommt eine genaue `0` heraus.)

Das Addieren zweier komplexer Zahlen geschieht durch Addition der reellen und der imaginären Komponenten, die *Subtraktion* durch Addition mit dem Negativwert. – Das Multiplizieren geht nach der Regel "jeder mit jedem". Die Funktion `&add4(...)` addiert, `&mul4(...)` multipliziert Real/Imaginär-Gruppen.

Zum Dividieren multiplizieren wir mit dem Kehrwert. Mit Herrn Binomi fällt das gar nicht schwer: Um zur Zahl $(a + b\mathtt{i})$ den Kehrwert $\frac{1}{(a+b\mathtt{i})}$ zu finden, erweitern wir den Bruch um $(a-b\mathtt{i})$, ergibt nach der 3. binomischen Regel $\frac{(a-b\mathtt{i})}{(a^2+b^2)}$ oder $\frac{a}{(a^2+b^2)} - \frac{b}{(a^2+b^2)}\mathtt{i}$ – reelle und imaginäre Komponente kürzen, und fertig. Das erledigt für uns die Funktion `&Kehrwert(...)`. Siehe *number.pl*.

2.9 Call-with-current-Continuation

Diese Funktion, abgekürzt `call/cc`, dient dem "geordneten Rückzug" aus einem komplizierten Geflecht aus Funktionsaufrufen. Sie verwendet eine lokale Variable namens `$CallCCResult` und eine private Lisp-Funktion mit dem Perl-Namen `$exitfunc`, die einfach nur ihr Argument in `$CallCCResult` schreibt und anschließend "stirbt".

Die `call/cc` erwartet als Argument einen LAMBDA-Ausdruck, dem sie als Argument `$exitfunc` mit auf die Reise gibt. Innerhalb des LAMBDA-Ausdrucks hat der Verbraucher freie Bahn – wenn er der Meinung ist, es sei zu viel des Guten, ruft er die Ausstiegsfunktion auf, und diese liefert dann das Resultat der Rechenoperation. Beispiel? Beispiel:

```
Lisp> (**define** reelle_wurzel
         (lambda (x)
             (call/cc (lambda (abbruch)
                 ; der "Abbruch" wird zur Ausstiegsfunktion.
                 (if (< x 0) (abbruch 'geht-nicht!))
                 ; die Ausstiegsfunktion
                 ; "Abbruch" verhindert den Rest.
             (sqrt x)))))

⇒ REELLE_WURZEL

Lisp> (reelle_Wurzel 4)

⇒ 2.0

Lisp> (reelle\_Wurzel -4)

⇒ GEHT-NICHT!
```

Wie ist es gemacht? Die Perl-Funktion `eval` sorgt dafür, daß nicht der "Tod" von Lisp eintritt, sondern nur der Abbruch der aktuellen Funktion. Anschließend schaut `call/cc` nach, ob der LAMBDA-Ausdruck etwas nennenswertes hervorgebracht hat – falls nein, hat die Ausstiegsfunktion zugeschlagen, und das Resultat steht im `$CallCCResult`. Nur, wenn da auch nichts drinsteht, gab es ein echtes Problem, das in der Variablen `$@` steht. – Der Code im Kern: `{eval &$c($exitfunc)} || $CallCCResult || die ($@)}`

Kapitel 3

Makros

Die Beliebtheit von LISP steht nicht nur für seine Mächtigkeit, sondern auch für seine Einfachheit. Ist unser LISP schon einfach? Na ja, es gäbe da noch einige Verbesserungsmöglichkeiten, z. B. lokale Bindungen mit dem `let`-Konstrukt.

Das Makro erlaubt uns, aus einer relativ menschenfreundlichen Syntax einen regulären Ausdruck zu machen, z. B. aus `(let ((a pi) (b 2)) (* a b))` den Ausdruck `((LAMBDA (a b) (* a b)) pi 2)`. Das Makro ist die Patentante des Präprozessors bei C.

Für das Makro legen wir eine Assoziativliste `%Macro` an, deren Schlüssel die Makro-Namen und deren Werte LISP-Funktionen enthalten. (Makros programmieren wir prinzipiell in LISP.) Wir realisieren den Makro-Generator als Spezialform:

```
$SpecialSource{"MACRO"} = sub
{
  my $expr = shift;
  my $sym = &CAR($expr);
  my $MacroFunc = &CADR($expr);
  "do { \$Macro{'"
    . &GetSymbolName($sym)
    . "'} = "
    . &GeneratePerlSource($MacroFunc)
    . ";"
    . &GenerateQuotedSource($sym) # Symbol als Rückgabewert
    . " } "
}
```

Und jetzt müssen wir `&GeneratePerlSource` so erweitern, daß er zuvörderst auf Makro testet und ggf. umformt:

```
while (&IsPair( $expr)
```

```
        && $Macro{&GetSymbolName(&CAR( $expr))})
{
  my  $MacroSym = &CAR( $expr);  # 'let
  my  $MacroName = &GetSymbolName( $MacroSym);  # "LET"
  my  $MacroFunc =  $Macro{ $MacroName};  #  #<code ...>
  my  $PerlCode = &GetFunctionCode( $MacroFunc);  # code ...
  $expr = & $PerlCode( $expr)   # Expansion!
}
```

Nun können wir das Makro `let` definieren:

```
(macro let
  (lambda (expr)
    (apply
     (lambda (bindings . body)
         (cons
             (cons 'lambda
                   (cons (map car bindings) body))
             (map cadr bindings)))
    (cdr expr))))
```

⇒ `LET`

```
Lisp> (let ((a 1) (b 2)) (+ a b))
```

⇒ `3`

Na ja, war relativ einfach. Bei komplizierten Makros brauchen wir aber zur Vorsorgeuntersuchung etwas, das uns die Makro-Expansion einmal "trocken" vorführt. Die Funktion `macrocode` übergibt den Code eines Makros. Der Ausdruck `(macrocode 'let)` z. B. übergibt den Code des Makros `let`. Damit können wir die Umformung direkt beobachten, anstatt über Fehler zu rätseln:

```
Lisp> ((macrocode 'let) '(let ((a 1) (b 2)) (+ a b)))
```

⇒ `((LAMBDA (A B) (+ A B)) 1 2)`

Die `quasi-quote` erlaubt die Evaluierung einzelner Ausdrücke innerhalb ihres Arguments mit den Schlüsselwörtern `unquote` und `unquote-splicing` (abgekürzt: `,`<expr> und `,@`<expr>). Das Makro `define` erspart uns die lästige LAMBDA-Anweisung: `(define (function arg1 ...) ...)`. Das Makro `let*` erzeugt einen verschachtelten `let`-Ausdruck, bei dem jedes Symbol dem nachfolgend definierten bekannt ist; bei `letrec` kennen die Symbole einander und sich selbst – nützlich bei lokalen Funktionen. Das Makro `named-lambda` macht Gebrauch davon und erzeugt eine Funktion, bei der Selbstaufrufe auch nach Umtaufe funktionieren. `cond` überführt eine übersichtliche Fallunterscheidung in einen verschachtelten `if`-Ausdruck, `case` macht aus einer switch-ähnlichen Syntax einen `cond`-Ausdruck. Das Makro `do` erzeugt eine abweisende bedingte Programmschleife.

Aber cave!

Der innere Ablauf bei der Makro-Umformung zeigt eine Einschränkung auf: Beim Kompilieren einer Funktion werden alle Makros vollständig angewendet. Also muß das Makro vorher bekannt sein. Sollten Sie eine Funktion definieren, die ein Makro verwendet, und das aufgerufene Makro erst später definieren, dann "denkt" der Compiler, es handele sich um eine normale Funktion – ein Aufruf des Makros als Funktion führt zum Fehler.

Eine andere Gefahr besteht darin: es ist ohne weiteres möglich, ein Makro zu bauen, das Nutzdaten berechnet – durch direkten oder versteckten Aufruf von `(eval ...)`. Das tut es aber zur Kompilationszeit. Im Testlauf auf Kommandozeile funktioniert alles, aber zur Programmlaufzeit sind die Daten nicht mehr aktuell. Und dann suchen Sie mal die Ursache.

Das zwingt zur Erkenntnis: Das Makro wirkt innerhalb von Lisp eigentlich als *Störenfried.* Für die reine Funktionalität wäre es absolut verzichtbar. Seine einzige Daseinsberechtigung ist seine Benutzerfreundlichkeit.

Falls benutzerdefinierte Makros wirklich erforderlich sind, empfehle ich diese Vorgehensweise: Definieren Sie alle nötigen Makros "am Stück". Stellen Sie die Definitionen an den Anfang Ihres Programms. Packen Sie sie in eine eigene Datei. Vermeiden Sie anderweitige Verwendung der Makro-Symbole, auch im mit quote "entschärften" Zustand. Andernfalls sehe ich lange und erfüllte Abende der Fehlersuche voraus.

Kapitel 4

Zusammenfassung

Der einzige größere Flaschenhals in diesem Lisp ist die Benutzereingabe, bei der der eingegebene Klartext zunächst in ein Lisp-Datum umgewandelt wird, dieses wiederum in Perl-Quelltext, und dieser zu Code.

Den zweitgrößte Flaschenhals bildet `(eval<expr>)`, bei dem das Lisp-Datum über Klartext zu Code gemacht wird.

Eine weitere Bremse für den Durchsatz ist der Datentyp Liste, für den es in Perl keine unmittelbare Entsprechung gibt.

Abgesehen davon beschränkt sich der Mehraufwand auf die Typ-Kontrolle bei geschätzten 95% aller Funktionen. In summa stellen wir fest: dies Lisp ist immerhin fast so schnell wie Perl. Und das ist ja wohl recht ordentlich, gell?

4.1 Was der Meister sich verkniff

Die Zahlenoperationen sind so genau, wie Perl eben zuläßt. Ziffernfolgen mit beliebiger Stellenzahl müßten ausdrücklich programmiert werden, und das wäre bei den Bordmitteln von Perl vielleicht eine winzige Spur zu langsam ...

Unser Lisp verzichtet auf einen Entlauser (siehe unten, Glossar: E). Im Fehlerfall "stirbt" das Programm und fällt zurück auf die Eingabezeile. Seine Daten hat es nicht vergessen, allerdings den Absturzort. Für den Notfall gibt es immer noch den Perl-Debugger, und für den Profi bleibt die Möglichkeit, diesen Lisp-spezifisch anzupassen.

Es gibt keinen Support, ich habe auch noch *Familie.* Betrachten Sie dies Lisp als Hack vom Hacker für Hacker. Fehlerrückmeldungen sind natürlich hochwillkommen!

4.2 Ausblick

Lisp ist eine einfache und zugleich mächtige Programmiersprache, aber keine schnelle. Ein vernünftiger Mensch würde z. B. *niemals* Bitmapmanipulationen damit realisieren. Das Heimstadion von Lisp ist die Logik. Die professionelle Applikation Schematext z. B. ist in Lisp programmiert. Sie verwaltet große Bestände von Internet-Dokumenten mitsamt einem komplizierten Geflecht von Hyperlinks (siehe Glossar: H). Für alles, was über die Logik hinaus geht, ruft diese Applikation Fremdprogramme auf, z. B. Help-Compiler von Drittherstellern. – Andere Anwendungsbereiche sind z. B. maschinelle Roh-Übersetzungen in Fremdsprachen – jedenfalls, solange es noch keine vollständige Abbildung menschlicher Sprache in ein formales Modell gibt. (Wenn ich schon nicht alles Gesprochene verstehe ... aber lassen wir Politik und Betriebsrat außen vor.)

Wenn ein Perl-Programm vor komplizierten logischen oder logistischen Problemen steht, kann es unser Lisp einbinden. Die Funktion `&CommandLisp` bildet eine pflegeleichte Klartext-Schnittstelle. Siehe unten: Technik.

Perl erlaubt es, den vollständigen Quelltext dieses Lisp innerhalb eines Batchfiles auszuführen. (Vor ein paar Jahren hätten wir über die Vorstellung gelacht.) Das ermöglicht praktische Lisp-Übungen in Schule, Studium und Wohnzimmer. Für den Beruf empfehle ich es nur bedingt, denn es gibt keinen Support – siehe oben.

Anhang A

Besonderheiten

Dies Lisp enthält die Funktionen, die im Revised Report als wesentlich (*essential*) gekennzeichnet sind. Für die normale Lisp-Syntax lesen Sie den Revised Report oder irgend ein Lehrbuch zu Scheme. Hier finden Sie, was darüber hinausgeht, was also dort nicht vorkommen kann.

Speicherverwaltung – der Revised Report trifft genauere Aussagen zur Symbolbindung. Unter Perl gibt es keine Pointer, sondern Referenzen. Sollte den Anwender nicht weiter berühren.

Dies Lisp ist geeignet für den Aufruf aus regulären Perl-Programmen heraus, sozusagen im Serverbetrieb. Als Schnittstelle dient die Funktion `&CommandLisp(...)`. Sie erhält als Argument einen Klartext, z. B. Erweiterungen, die in IEEE 1178 aufgenommen sind.

Assoziativlisten (hash lists) haben die Schreibweise `hash("Schluessel1" Wert1 ..."Schluesseln" Wertn)`. Kennzeichnend ist das Kürzel `#hash` vor der Klammer. Diese Erweiterung erfolgte wegen der günstigen Gelegenheit und geht über IEEE 1178 hinaus. Programme mit Assoziativlisten sind nicht mehr portabel.

A.1 Erweiterungen

Die funktion `parse` macht aus einem Wort einen Lisp-Ausdruck und übergibt als Resultat eine Liste mit dem resultierenden Ausdruck und den Rest des Worts. Das Wort muß einer syntaktisch vollständigen Benutzereingabe entsprechen, sonst erfolgt ein Abbruch mit Fehlermeldung. Mit dieser Erweiterung könnten Sie z. B. eine Funktion namens `call-with-input-string` definieren.

Die Funktion `try-to-eval` wertet ihr Argument aus. Ein Fehler führt nicht zum Programmabbruch, sondern die Funktion übergibt ein Fehler-Objekt. – Die Funktion `error-object?` ermittelt, ob ein Objekt ein Fehlerobjekt ist. – Die Funktion `error->string` überführt ein Fehlerobjekt in ein Wort.

Anhang B

Änderungsgeschichte

Version 1.2

Der Parser kann jetzt Assoziativ-Listen unmittelbar einlesen, und zwar in der Schreibweise `#hash("key1"value1 ...)`. Die Schlüssel müssen unmittelbare Worte sein. Die Assoziativliste muß ausbalanciert sein. Die Werte werden evaluiert, es sei denn, die Assoziativliste wird mit der `'`Quote bzw. `‘`Quasiquote eingegeben.

So ist es möglich, eine Assoziativ-Liste unmittelbar in eine Datei zu schreiben und zurückzulesen.

Die Erweiterung eines Lisp-Parsers ist kritisch, denn die Typ-Syntax ist ein tragendes Element. In anderen Implementierungen habe ich erweiterte Schreibweisen vorgefunden wie `#.` oder `#!`, deren Namensphilosophie wohl aus der Unix-Welt stammt. Ich war so frei und habe mich für die Namensphilosophie von Lisp entschieden und das selbsterklärende Kürzel `#hash` gewählt und in Kauf genommen, daß es mehrbuchstabig ausfällt.

Ihnen ist klar, daß diese Schreibweise ebensowenig portabel ist wie der Datentyp.

Version 1.1a

Lisp läd ein zu rekursiver Programmierung, aber der Speicherbedarf ist fürchterlich. Ein Ausweg ist die Umsetzung von Endrekursionen zu Schleifen, die im Revised[5] Report übrigens spezifiziert ist. Der Mechanismus ist einfach. Die Befehlsfolge eines Ausdrucks lautet sinngemäß: *call subroutine with arguments; return to caller with result.* Diese immer gleiche Befehlsfolge wird im *letzten* Ausdruck ersetzt durch die funktionsgleiche Anweisung *jump to subroutine with arguments.*

Im Normalfall steigt der Durchsatz nur geringfügig. Aber bei Rekursionen schlägt die Quantität um in Qualität. Wenn eine Funktion an letzter

Stelle im Funktionskörper sich selbst aufruft, dann gibts eine *Wiederholschleife*.

Angefordert wird die Endrekursion von der Spezialform `lambda`. Berücksichtigt wird sie vom Quelltextgenerator, der entweder eine Absolut-Sprunganweisung einfügt oder an die Anforderung die Spezialformen `and`, `or`, `if`, `begin` durchreicht.

Dabei findet das Perl-Statement `goto` Verwendung, das von Informatikern skeptisch betrachtet wird, weil ein Programm mit Sprunganweisungen unübersichtlich und nicht wartungsfreundlich ist. Hier greift das Kriterium nicht, denn hier wird der Code grundsätzlich nur genutzt und nicht gewartet.

In Konsequenz wurde das Makro `do` sowie einige hochsprachlich definierte Funktionen umgeschrieben (`member`, `reverse`). Die Spezialform `**until**` wurde gelöscht. – Außerdem wurde das Makro `do*` aus den Grundfunktionen entfernt. Sie finden es ersatzweise in `autoload.lsp`. Es wird nicht weiter gepflegt.

Nachbemerkung: trotz der erwähnten Spezifikation im R^5RS – ein Fünfer-Lisp gibts hier nicht.

Version 1.1

In der ersten Fassung enthielt dies Lisp genau den Funktionsumfang, den Scheme als wesentlich ansieht. Aber naheliegenderweise sollte es nicht weniger können als Perl.

Weiterhin waren einige Unvereinbarkeiten mit dem Revised Report sowie einige z. T. krasse Fehler unerkannt geblieben. Mithilfe der Testsuite 3.99 von Aubrey Jaffer sind die Fehler behoben – bis auf eine Kleinigkeit: die Leere Liste wird weiterhin (und erlaubterweise) als falsch bewertet. Demzufolge evaluiert `(not ())` zu `#t` und nicht zu `#f`, wie die Testsuite es erwartet. Die zwei letzten Fehlermeldungen lasse ich also stehen.

Der Datentyp `hash` erlaubt schnelleren Zugriff auf Schlüssel-Werte-Paare als die Listen vom Typ `((var1 val1) (var2 val2) ...)`. Die Funktion `(hash "string1" val1 ...)` erzeugt eine *Assoziativliste*. Mit `(hash-set! h "key"val)` setzen Sie ein neues Schlüssel-Wert-Paar ein, mit `(hash-ref h "key")` rufen Sie es ab. Falls der Schlüssel nicht vorkommt, gibt diese Funktion `#f` zurück. Mit der Funktion `(hash-keys h)` erhalten Sie eine Übersicht der Schlüssel, mit `(hash-values hash)` eine Übersicht der Werte.

Falls Sie mit Eigenschaften (properties) arbeiten wollen, bietet sich anstelle der klassischen `assoc`-Listen die wesentlich schnellere Assoziativlisten des Typs `hash` an. Der Schlüssel muß aber ein Wort sein. Die neue Funktion `object->string` erzeugt aus einem beliebigen Lisp-Objekt dessen Bildschirm-Repräsentation als Wort. Aber Achtung, für die Eindeutigkeit der Schlüssel müssen Sie selbst sorgen, denn bekanntlich sind die Bildschirm-Repräsentationen nicht umkehrbar-eindeutig!

Die Funktion `load` erlaubt optional als dritten Parameter eine Funktion, die dann anstelle `eval` auf die eingelesenen Ausdrücke angewendet wird; beispielsweise eignet sich der Ausdruck `(load "test.lsp"(lambda (p) (write (pp p)) (newline) (read-char)))`, um einen Quelltext Ausdruck für Ausdruck auf dem Bildschirm zu betrachten.

Der Durchsatz steigt durch effizienteren Bau einiger Routinen. Die Funktion `&EvalExpr()` verzichtet darauf, einen Ausdruck auf zyklische Liste zu überprüfen. Die Funktion `&Lisp2PerlName()` verwendet jetzt die Perl-Funktion `s///g`. Die Funktion `&MakeList()` wurde überarbeitet.

Die Leere Liste `()` ist nicht mehr der Wahrheitswert falsch, sondern bildet einen eigenen Datentyp. Sie ist folglich nicht mehr identisch mit `#f`. Aber sie wird weiterhin wie `#f` als "falsch"bewertet.

Der *Mustervergleich* ist primitiv programmiert: `(match "abcdef" "C" "i")` greift das Muster `/C/` aus dem Wort `"abcdef"` ab – wegen des `"i"` nimmt es keine Rücksicht auf Groß-Klein-Schreibung – und übergibt als Resultat eine Liste der Bestandteile vor dem Treffer, den Treffer selbst, und dem Bereich dahinter: `("abcdef")`. Wenn es keinen Treffer erzielt, übergibt es `#f`.

In Abweichung von der Reinen Lehre ist jetzt die Lokale Sichtweise von Variablen eingebaut, und zwar mit dem Makro `(local ((var1 val1) ...) ...)`, das syntaktisch analog `let` funktioniert, und das eine neue Spezialform `lambda-local` aufruft.

Die Eingabe von Zeichenketten darf jetzt mehrzeilig ausfallen. Außerdem ist ein Fehler behoben: der Parser hat in der ersten Fassung die Sonderzeichen `"` etc. nicht zuverlässig verarbeitet.

Die Funktion `read-line` liest eine Zeile von Tastatur oder Port und übergibt den Inhalt als Wort. Merkwürdigerweise nicht im Revised Report erwähnt; möglicherweise (?) deshalb, weil mit `read-char` programmierbar.

Die Symbole sind nun in einer Tabelle zusammengefaßt. Gleichnamige Symbole kommen nur einmal vor. Damit kann die Funktion `eq?` ëhrlich"werden und auch bei Symbolen auf wirkliche Identität testen anstatt wie bisher auf Typ- und Wert-Gleichheit. Erhebliche Speicherersparnis und damit reduzierte Speicherbereinigung gibts vermutlich nur, wenn man intensiv mit Eigenschaften (properties) arbeitet. (Ich hab da ein Projekt am Laufen mit deutscher Grammatik.)

Version 1.0

Wenn dies Lisp gestartet wird, lädt es zuerst die Datei `Autoload.lsp` – sofern vorhanden. Hier können Sie Ihr Lisp nach Gusto ausbauen.

Sie können dies Lisp direkt mit einem Lisp-Programm aufrufen. Nach dem Autoload wird der Pfad der Programmdatei ermittel, das Lisp wechselt in den Pfad und führt die Programm-Datei aus. Dabei berücksichtigt es die

Pfad-Separatoren \ und /.

Falls Sie dies Lisp auf einem Betriebssystem verwenden, das andere Pfad-Separatoren verwendet, passen Sie das entsprechende Muster am Ende des Quelltextes an. Falls Ihr Betriebssystem zuläßt, daß Pfad-Separatoren als normale Buchstaben in Pfad- und Dateinamen auftauchen, kann das Muster natürlich versagen.

Anhang C

Fehler und Probleme

C.1 Bekannte Fehler

Dies Lisp kennt keine Nur-Lese-Daten, wie der Revised Report es fordert. Beispiel:
`(define (g) '(constant list));`
`(g)` ⇒ `(CONSTANT LIST);`
`(set-car! (g) 'changed)` – hier wäre eine Fehlermeldung gefordert;
`(g)` ⇒ `(CHANGED LIST)`
die Funktiong wurde inhaltlich geändert.

Die Funktion `map` erzeugt keine Fehlermeldung, wenn Sie auf eine unordentliche Liste angewendet wird, also wenn das innerste `cdr` nicht `()` ist.

Die Funktionen `car` und `cdr` erzeugen keine Fehlermeldung, wenn ihr Argument die Leere Liste `()` ist, sondern übergeben als Resultat die Leere Liste. Damit ist dies Lisp zwar funktionsfähig, weicht aber ab vom Revised[4] Report, und die Portabilität der Programme leidet bei Ausnutzung dieses Fehlverhaltens. Ich arbeite dran.

Bei einem Programmabbruch werden geöffnete Dateien nicht geschlossen.

Die Funktion `call-with-current-continuation (call/cc)` kann z. Z. nur als Ausstiegsfunktions verwendet werden. Beim Verschachteln kann aus dem inneren `call/cc`-Konstrukt nur die innere Ausstiegsroutine sinnvoll eingesetzt werden, nicht die äußere. Falls die äußere aufgerufen wird, wird trotzdem *(!)* die innere ausgeführt.

C.2 Behobene Fehler

Bei Division durch Null erfolgte nicht zuverlässig Fehlermeldung. Der Fehler ist behoben.

Die Funktion `number->string` arbeitete fehlerhaft bei Zahlenbasen oberhalb 10. Der Fehler ist behoben.

Das Makro `cond` verarbeitet jetzt das Schlüsselwort `=>` korrekt. Der Quelltext für das Makro ist etwas länger ausgefallen, die Expansionen sind dagegen im ganzen etwas kürzer und eleganter.

Das Verhalten der Funktionen `eq?` und `eqv?` ist jetzt auch bei nichtleeren Worten identisch, wie es der Revidierte Bericht fordert.

Die Funktion `string->number` behandelte seine Zeichenfolgen bisher zu tolerant. Ab sofort übergibt sie den Wert `#f`, falls ein Wort nicht vollständig zu einer Zahl umzuformen ist.

Die Funktionen `list-ref` und `list-tail` waren fehlerhaft. Die Funktionen `min` und `max` ignorierten das letzte Argument. Die Funktion `length` hat eine Liste beim Messen der Länge "aufgefressen", hinterher hatte sie den Wert `()`. Die Fehler sind behoben.

Das Makro `case` expandiert zu einem `let`-Konstrukt, bei dem es zu einem Konflikt mit den Namen im Ausdruck kommen konnte. Außerdem verwendete es die Vergleichsfunktion `equal?` anstelle `eqv?`. Die Fehler sind behoben.

Bei Version 1.1 wurde der Datentyp `hash` eingeführt. Die Typ-Abfrage `hash?` fehlte dabei. Das ist bei Einführung eines neuen Typs aber nicht Kür, sondern Pflicht. Der Fehler ist behoben. – Die Funktion `make-hash` ist umgetauft in `hash` entsprechend den Scheme-Konventionen. Siehe `string` oder `list` oder `vector`, die ebenfalls Nutzdaten als Argument erhalten. Falls Sie die Funktion `make-hash` bereits nutzen und auf die aktuelle Lisp-Version umsteigen wollen, behelfen Sie sich mit `(define make-hash hash)`. – Ebenfalls entsprechend Scheme-Konventionen übergibt `hash-ref` bei Nichtexistenz des Schlüssels den Wert `#f` anstelle `nil`.

Die Funktion `eqv?` funktioniert jetzt auch mit Zahlen. Die Funktion `eq?` funktioniert jetzt korrekterweise auch mit `#t`, `#f` und `nil` bzw. `'()` sowie mit Buchstaben (chars).

Das Makro `do` arbeitete unzuverlässig: die Iteration der Initialvariablen war sequentiell eingebettet in den Körper der Wiederholschleife – dämlicherweise, denn es konnte vorkommen, daß eine Iteration auf var_i Bezug nahm auf eine andere Iterationsvariable var_k, die bereits iteriert hatte. Der Fehler ist behoben.

Die Funktion `positive?` betrachtet die Null nicht mehr länger als positiv. Der Revised Report verweist bei numerischen Funktionen auf den IEEE Standard 754-1985 (IEEE Standard for Binary Floating-Point Arithmetic; IEEE, New York, 1985). Die Quelle kenne ich nicht selbst, sondern beziehe mich auf die Validierungssuite von Aubrey Jaffer. *Achtung!* Funktionale Änderung gegenüber früheren Versionen!

Die Funktion `substring` erwartet jetzt entsprechend dem Revised Report die Start- und End-Position der Buchstaben im Wort. (Bisher erwartete die Funktion Startposition und Länge.) *Achtung!* Funktionale Änderung gegenüber früheren Versionen!

Die Funktionen `close-input-port` und `close-output-port` funktionieren jetzt auch bei bereits geschlossenen Ports.

Die Funktion `map` verarbeitet nun auch mehrere Listen als Argument.

Die `quasiquote` ist jetzt schachtelbar entsprechend den Vorgaben des Revised Report.

Wenn innerhalb eines Lambda-Ausdrucks mehrere `define`-Konstrukte stehen (z. B. für lokale Funktionen), dann haben sich diese bisher nur in rückwärtiger Richtung gegenseitig erkannt. Jetzt erkennen sie sich auch vorwärts.

Aufgrund eines Verständnisproblems beim Autor hat der Evaluierer beim Verarbeiten eines Vektors `#(`$\texttt{expr}_1$ `...)` die enthaltenen Ausdrücke `expri` nicht evaluiert. Jetzt tut er es. Um die Evaluierung zu verhindern, verwenden Sie die `quote` bzw. `quasiquote`, also: `´#(`$\texttt{expr}_1$ `...)` bzw. `` `#(`$\texttt{expr}_1$ `...)`. Achtung! Syntaktische Änderung gegenüber früheren Versionen!

Die Multiplikation `*` arbeitete fehlerhaft bei Aufruf ohne Argument. Jetzt produziert `(*)` korrekterweise `1`. – Die Funktionen `lcm` und `gcd` verlangten früher mindestens ein Argument. Der Fehler ist behoben: `(lcm)` produziert `1`, `(gcd)` produziert `0`. Keine Ahnung, warum das so sein muß; intuitiv hätte ich auf umgekehrt getippt: seit wann ist die Null ein Teiler?

Aufgrund einer versehentlich gelöschten Programmzeile fehlte in der ersten Fassung die Funktion `char?`. Die Funktion `make-vector` fehlte ebenfalls. Beide Fehler sind behoben.

Bei Eingabe eines einzelnen Punkts in der REP meldete dies Lisp bisher `single dot is ambigous`. Das war falsch. Richtig ist: ambiguous. Der orthographische Fehler ist behoben.

Anhang D

Funktionen

\+ - / die vier Grundrechenarten. Beispiel: `(- 4 3 5)` ⇒ `-4`

< <= > >= arithmetische Vergleichsfunktionen. Beispiel: `(< 4 11 16 1.435e20)`⇒ `#t`

A

abs ermittelt den Absolutwert einer Zahl. Beispiel: (abs -13) ⇒ 13

angle ermittelt den Winkel einer komplexen Zahl in geometrischer Darstellung. Beispiel: (angle -1) ⇒ 3.1415926535898

append hängt mehrere Listen zusammen. Beispiel: (append '(a b) '(c d) '(e f)) ⇒ (a b c d e f)

apply wendet eine Funktion auf eine Liste an. Beispiel: (apply + '(3 4 5)) ⇒ 12

assoc, assq, assv durchsucht eine Liste von Listen nach einem bestimmten Element. `assoc` verwendet als Vergleichsfunktion `equal?`, `assv` verwendet `eqv?`, `assq` verwendet `eq?`. Beispiel: `(assoc 'a '((1 2) (a b) (apfel birne)))` ⇒ `(a b)`

B

boolean? stellt fest, ob sein Argument ein Wahrheitswert ist. Beispiel: `(boolean? nil)` ⇒ `#t`

C

call-with-input-file, call-with-output-file dient zum Dateiverkehr. Die Funktionen erwarten als Argumente einen gültigen Dateinamen und einen Lambda-Ausdruck mit einem Port als Argument. Die Datei wird geöffnet, der Lambda-Term mit dem entsprechend Port als Argument ausgeführt, und schließlich der Port geschlossen. Beispiel:

```
(call-with-input-file
     "c:\\config.sys"
     (lambda (port)
             (read port)))
```

⇒ `DEVICE=C:\WINDOWS\HIMEM.SYS`

(Das Ergebnis steht ohne Gänsefüßchen, denn es ist ein Symbol. Ich staune manchmal selbst, wieviele Sonderzeichen akzeptiert werden.)

car ermittelt das erste Element einer Liste. Komplement zu `cdr`. Beispiel: `(car '(a b c))` ⇒ `(a)`

cdr ermittelt den Rest einer Liste ohne das erste Element. Komplement zu `car`. Beispiel: `(cdr '(a b c))` ⇒ `(b c)`

caar, cadr, cdar, cddr, caadr, . . . , cddddr Kombinationen aus `car` und `cdr` – insgesamt 28 Kombinationen. `cadr` z. B. ist so definierbar: `(define (cadr l) (car (cdr l)))`. Beispiel: `(cadr '(a b c))`⇒ `b`

ceiling ermittelt die nächsthöhere Ganzzahl. Beispiel: `(ceiling 5.2)` ⇒ `6`

char? stellt fest, ob ein Objekt ein Buchstabe ist. Beispiel: `(char? #\a)` ⇒ `#t`

char->integer findet den ASCII- bzw. ANSI-Wert eines Buchstaben. Hinweis: Verwendet die Perl-Funktion `ord()`. Beispiel: `(char->integer #\a)` ⇒ `65`

char-alphabetic? stellt fest, ob ein Buchstabe alphabetisch ist. Beispiel: `(char-alphabetic? #\9)` ⇒ `#f`

char<=?, char<?, char=?, char>=?, char>? vergleicht Buchstaben. Beispiel: `(char=? #\a #\A)`⇒ `#f`

char-ci<=? , char-ci<?, char-ci=?, char-ci>=?, char-ci>? vergleicht Buchstaben ohne Rücksicht auf Groß-Kleinschreibung.
Beispiel: `(char-ci=? #\a #\A)` ⇒ `#t`

char-downcase findet den passenden Kleinbuchstaben.
Beispiel: `(char-downcase #\A)` ⇒ `#\a`

char-lower-case? stellt fest, ob ein Buchstabe kleingeschrieben ist. Beispiel: `(char-lower-case? #\A)` ⇒ `#f`

char-numeric? stellt fest, ob eine Buchstabe eine Ziffer darstellt. Beispiel: `(char-numeric? #\8)` ⇒ `#t`

char-upcase findet den passenden Großbuchstaben. Beispiel: `(char-upcase #\a)`⇒ `#\A`

char-upper-case? stellt fest, ob ein Buchstabe großgeschrieben ist. Beispiel: `(char-upper-case? #\A)` ⇒ `#t`

char-whitespace? stellt fest, ob ein Buchstabe ein Leerzeichen ist. Beispiel: `(char-whitespace? #\newline)` ⇒ `#t`

close-input-port, close-output-port schließt eine geöffnete Eingabe- bzw. Ausgabe-Datei. Beispiel: `(close-input-port port)` ⇒ `#t`. Seiteneffekt: der Eingabeport port wird geschlossen.

complex? stellt fest, ob ein Objekt eine komplexe Zahl ist.
Beispiel: `(complex? 5+2i)` ⇒ `#t`

cons erzeugt einen Knoten. Beispiel: `(cons 'a 'b)` ⇒ `(a . b)`

current-input-port übergibt das Dateiobjekt, das für die Tastatureingabe zuständig ist. Beispiel: `(current-input-port)` ⇒ `#<Port>`

current-output-port übergibt das Dateiobjekt, das für die Bildschirmausgabe zuständig ist. Beispiel: `(current-output-port)` ⇒ `#<Port>`

D

defined? stellt fest, ob ein Symbol definiert ist. Beispiel:
`(defined? 'cons)` ⇒ `#t`

display schreibt ein Objekt auf Bildschirm oder Port. Wörter erscheinen ohne Gänsefüßchen, Buchstaben ohne Schrägstrich und Raute. Beispiel: `(display '(äbc"#\a))` ⇒ `#t`. Seiteneffekt: schreibt `(abc a)`auf den Bildschirm.

E

eof-object? stellt fest, ob ein Objekt ein Dateiende-Objekt ist. Beispiel: `(eof-object? result)` ⇒ `()`

eq?, eqv?, equal? vergleicht Objekte. `eq?` Vergleicht auf Identität, wobei gleichnamige Symbole als identisch gelten. `eqv?` vergleicht auf gleichen Typ und gleichen Wert (außer Strings), `equal?` vergleicht Listen rekursiv. Beispiel: `(equal? '(a b) '(a b))` ⇒ `#t`

error-object? stellt fest, ob ein Objekt ein Fehler-Objekt ist. Beispiel: `(error-object? #f)`⇒ `#f`

error->string macht aus einem Fehler-Objekt ein Wort.

even? stellt fest, ob eine Zahl gerade ist. Beispiel: `(even? 15)` ⇒ `#f`

exact? stellt fest, ob eine Zahl genau ist. Beispiel: `(exact? 15)` ⇒ `#t`

exact->inexact wandelt eine genaue Zahl um in eine ungenaue. Beispiel: `(exact->inexact 4)` ⇒ `4.0`

exit – verläßt Lisp und Perl. Beispiel: `(exit)` ⇒ Seiteneffekt: Ende des Programms.

exp – berechnet den Exponenten zur Eulerschen Zahl e. Beispiel. `(exp 1)`⇒ `2.718281828459`

F

floor ermittelt die nächstkleinere Ganzzahl. Beispiel: `(floor 5.2)` ⇒ `5`

for-each wendet eine Funktion auf eine Liste von Argumenten an. Funktionsgleich map. Beispiel: `(for-each list '(a b))` ⇒ `((a) (b))`

G

gcd ermittelt den ggT (größten gemeinsamen Teiler, greatest common divisor). Beispiel: `(gcd 30 15 6)` ⇒ `3`

H

hash erzeugt eine Assoziativ-Liste. Beispiel: `(hash "Gemüt" '(hungrig durstig muede))` ⇒ `#Hash("Gemüt" (HUNGRIG DURSTIG MUEDE))`

hash? stellt fest, ob ein Objekt eine Assoziativ-Liste ist. Beispiel: `(hash? (hash "Gemüt\(hungrig durstig muede)))` ⇒ `#t`

hash-keys liefert eine Übersicht der Schlüssel einer Assoziativ-Liste. Beispiel: `(hash-keys zustand)` ⇒ `("Gemüt")`

hash->list macht aus einer Assoziativ-Liste eine Liste. Beispiel: `(hash->list zustand)` ⇒ `("Gemüt" (hungrig durstig muede))`

hash-ref ermittelt den Wert eines Schlüssels aus einer Assoziativ-Liste. Beispiel: `(hash-ref Zustand "Gemüt")` ⇒ `(hungrig durstig muede)`

hash-set! setzt ein Schlüssel-Wert-Paar in einer Assoziativ-Liste. Beispiel: `(hash-set! Zustand "Gemüt\(satt zufrieden trotzdem muede))` ⇒ `(satt zufrieden trotzdem muede)`. Seiteneffekt: der Schlüssel `"Gemüt"` in der Assoziativliste `ZUSTAND` wurde geändert.

hash-values liefert eine Übersicht der Werte einer Assoziativ-Liste. Beispiel: `(hash-values zustand)` ⇒ `((SATT ZUFRIEDEN TROTZDEM MUEDE))`

I

inexact? stellt fest, ob eine Zahl ungenau ist. Beispiel: `(inexact? 15)` ⇒ `#f`

imag-part ermittelt den imaginären Anteil einer komplexen Zahl. Beispiel: `(imag-part 5+2i)` ⇒ `2`

inexact->exact wandelt eine ungenaue Zahl um in eine genaue. Beispiel: `(inexact->exact 4.0)` ⇒ `4`

input-port? stellt fest, ob ein Objekt das Dateiobjekt einer Eingabedatei ist. Beispiel: `(input-port? port)` ⇒ `#t`

integer->char ermittelt einen Buchstaben nach ASCII bzw. ANSI. Verwendet die Perl-Funktion chr(). Beispiel: `(integer->char 65)` ⇒ `#A`

integer? stellt fest, ob eine Zahl ganzzahlig ist. Beispiel: `(integer? 5.2)` ⇒ `#f`

L

lcm ermittelt das kgV (kleinste gemeinsame Vielfache, least common multiple). Beispiel: `(lcm 30 15 6)` ⇒ `30`

length berechnet die Elementzahl einer Liste. Beispiel: `(length '(a b))` ⇒ `2`

list bildet eine Liste. Beispiel: `(list 1 2 'apfel)` ⇒ `(1 2 apfel)`

list->string bildet aus einer Buchstaben-Liste ein Wort. Beispiel: `(list->string '(#\W #\o #\l #\f))` ⇒ `"Wolf"`

list->vector erzeugt einen Vektor aus den Elementen einer Liste. Beispiel: `(list->vector '(a b c))` ⇒ `#(a b c)`

list-ref ermittelt ein Listenelement aus seiner Position in der Liste. Beispiel: `(list-ref '(a b c) 1)` ⇒ `b`

list? stellt fest, ob ein Objekt eine ordentliche Liste ist. Beispiel: `(list? '(a . b))` ⇒ `#f`

load lädt einen Quelltext. Beispiel: `(load äutoload.lsp")` ⇒ `#t`. Seiteneffekt: der Quelltext autoload.lsp wird ausgeführt.

log ermittelt den natürlichen Logarithmus (also zur Basis der Eulerschen Zahl e). Beispiel: `(log 1)` ⇒ `0`

M

magnitude ermittelt den Absolutwert einer Komplexen Zahl. Beispiel: `(magnitude 3+4i)` ⇒ `5.0`

make-polar erzeugt eine Komplexe Zahl mit dem ersten Argument als Absolutwert und dem zweiten als Winkel. Beispiel: `(make-polar 5 2)` ⇒ `-2.0807341827357+4.54648713412841`

make-rectangular erzeugt eine Komplexe Zahl mit dem ersten Argument als reellem und dem zweiten als imaginärem Anteil. Beispiel: `(make-rectangular 5 2)` ⇒ `5+2i`

make-string erzeugt ein neues Wort, wahlweise mit einem Vorgabewert. Beispiel: `(make-string 5 #\x)`⇒ `"xxxxx"`

make-vector erzeugt einen Vektor, wahlweise mit einem Vorgabewert. Beispiel: `(make-vector 5 'a)`⇒ `#(a a a a a)`

map wendet eine Funktion auf die Elemente einer Liste an. Funktionsgleich for-each. Beispiel: `(map list '(a b))` ⇒ `((a) (b))`

match prüft ein Wort auf das Vorkommen eines Musters. Beispiele: `(match äbcdefCi")` ⇒ `(äbcdef")`; `(match äbcX")` ⇒ `#f`

max ermittelt den größten Zahlenwert. Beispiel: `(max 3 5 4)` ⇒ `5`

member, memq, memv stellt fest, ob ein Element in einer Liste vorkommt. member vergleicht mit equal?, memq vergleicht mit eq?, memv vergleicht mit eqv?. Beispiel: `(member 'b '(a b c))` ⇒ `(b c)`

min ermittelt den kleinsten Zahlenwert. Beispiel: `(min 3 5 4)` ⇒ `3`

modulo berechnet den Rest bei Division. Resultat hat Vorzeichen des Nenners. Beispiel: `(modulo 14 3)` ⇒ `2`

N

negative? stellt fest, ob eine Zahl negativ ist. Beispiel: `(negative? -2)` ⇒ `#t`

newline schreibt einen Zeilenwechsel ins Display oder einen Port. Beispiel: (newline) ⇒ `#t`. Seiteneffekt: Zeilenwechsel.

not negiert einen Wahrheitswert. Funktionsgleich null?. Beispiel: `(not (= 1 1))` ⇒ `#f`

null? stellt fest, ob ein Objekt die Leere Liste ist. Funktionsgleich not. Beispiel: `(null? ())` ⇒ `#t`

number->string macht aus einer Zahl ein Wort. Beispiel: `(number->string 435.2)` ⇒ `"435.2"`

number? stellt fest, ob ein Objekt eine Zahl ist. Beispiel: `(number? 'a)` ⇒ `#f`

O

object->string übergibt die Bildschirm-Repräsentation eines Objekts. Wenn ein drittes Argument gegeben ist, erscheinen Worte ohne Gänsefüßchen und Buchstaben ohne Raute und Schrägstrich. Beispiele:
`(object->string '(ab "cd" #\e))` ⇒ `"(AB "cd" #\e)"`
`(object->string '(ab "cd" #\e) #t)` ⇒ `"(AB cd e)"`

odd? stellt fest, ob eine Zahl ungerade ist. Beispiel: `(odd? 5)` ⇒ `#t`

open-input-file öffnet eine Eingabedatei, übergibt das Dateiobjekt. Beispiel: `(define x (open-input-file "test.txt"))`⇒ `x`. Seiteneffekt: die Datei Eingabedatei test.txt wird geöffnet.

open-output-file öffnet eine Ausgabedatei, übergibt das Dateiobjekt. Beispiel: `(define y (open-output-file "test.out"))` ⇒ `y`. Seiteneffekt: die Datei Ausgabedatei `test.out` wird geöffnet.

output-port? stellt fest, ob ein Objekt das Dateiobjekt einer Ausgabedatei ist. Beispiel: `(output-port? p)` ⇒ `#t`

P

pair? stellt fest, ob ein Objekt ein Knoten ist. Beispiel: `(pair? '(a . b))` ⇒ `#t`

parse macht aus einem Wort einen Lisp-Ausdruck. Übergibt als Resultat den Ausdruck und den Rest des Worts, das nicht verarbeitet wurde. Beispiel: `(parse "(a b c) def")` ⇒ `((a b c) "def")`

peek-char liest einen Buchstaben von Tastatur oder Port, ohne den Dateizeiger vorwärts zu stellen. Beispiel: `(peek-char port)` (der nächste Buchstabe in der Datei ist a) ⇒ `#a`. Kein Seiteneffekt. Insbesondere erzeugt der nächste Aufruf `(peek-char port)` wiederum den Buchstaben `#a`.

positive? stellt fest, ob eine Zahl positiv ist. Beispiel: `(positive? 15)` ⇒ `#t`

procedure? stellt fest, ob ein Objekt eine Funktion ist. Beispiel: `(procedure? list)` ⇒ `#t`

Q

quotient dividiert zwei Zahlen ganzzahlig. Beispiel: `(quotient 14 3)` ⇒ `4`

R

rational? stellt fest, ob eine Zahl rational ist. Funktionsgleich number?. Beispiel: `(rational? 15)` ⇒ `#t`

read liest einen Ausdruck von Tastatur oder Port. Beispiel: (read) *(Benutzereingabe: apple)* ⇒ `apple`

read-char liest einen Buchstaben von Tastatur oder Port. Beispiel: (read-char) *(Benutzereingabe: a)* ⇒ `#a`

read-line liest eine Zeile von Tastatur oder Port und übergibt ihn als Wort. Beispiel: `(read-line)` *(Benutzereingabe: aber nein, ich bin ein Arbeitsbär)* ⇒ `"aber nein, ich bin ein Arbeitsbär
"`

real? stellt fest, ob eine Zahl reell ist. Funktionsgleich `number?`. Beispiel: `(real? 15)` ⇒ `#t`

real-part ermittelt den reellen Anteil einer komplexen Zahl. Beispiel: `(real-part 5+2i)` ⇒ `5`

remainder berechnet den Rest bei Division. Resultat hat Vorzeichen des Dividenden. Beispiel: `(remainder 14 3)`⇒ `2`

reverse erstellt eine Liste mit umgekehrter Reihenfolge. Beispiel: `(reverse '(a b c))` ⇒ `(c b a)`

round rundet eine Zahl kaufmännisch. Beispiel: `(round 4.2)` ⇒ `4`

S

set-car! ersetzt den CAR einer Liste durch ein anderes Element. Destruktiv. Beispiel: `(set-car! x 'apfel)`⇒ `apfel`. Seiteneffekt: das erste Element der Liste x ist mit dem Symbol `apfel` überschrieben.

set-cdr! ersetzt den CDR einer Liste durch ein anderes Element. Destruktiv. Beispiel: `(set-cdr! x '(birne))`⇒ (birne). Seiteneffekt: der Rest der Liste `x` (ohne erstes Element) ist mit der Liste birne überschrieben. (Es dürfte jetzt den Wert `(apfel birne)` haben.)

string bildet aus Buchstaben ein Wort. Beispiel: `(string #\W #\o #\l #\f)` ⇒ `"Wolf"`

string->list bildet aus einem Wort eine Liste von Buchstaben. Beispiel: `(string->list "Wolf")` ⇒ `(#\W #\o #\l #\f)`

string->number bildet aus einem Wort eine Zahl. Beispiel: `(string->number "45.2")` ⇒ `45.2`

string->symbol erzeugt ein aus einem Wort Symbol. Erlaubt auch Sonderzeichen. Beispiel: `(string->symbol "Wolf")` ⇒ `Wolf`

string-append hängt Wörter zusammen. Beispiel: `(string-append äb c")` ⇒ `äbc"`

string<=?, string<?, string=?, string>=?, string>? vergleicht Wörter. Beispiel: `(string=? "Wolf" "wolf")` ⇒ `#f`

string-ci<=?, string-ci<?, string-ci=?, string-ci>=?, string-ci>? vergleicht Wörter ohne Rücksicht auf Groß- und Kleinschreibung. Beispiel: `(string-ci=? "Wolf" "wolf")` ⇒ `#t`

string-length berechnet die Buchstabenzahl eines Worts. Beispiel: `(string-length "abc")` ⇒ `3`

string-ref ermittelt einen Buchstaben aus seiner Position im Wort. Beispiel: `(string-ref "abc" 2)` ⇒ `#\c`

string-set! überschreibt einen Buchstaben in einem Wort. Destruktiv. Beispiel: `(string-set! "abc" 1 #\x)`⇒ "axc". Seiteneffekt: das Wort `abc` wird körperlich überschrieben mit `axc`.

string? Stellt fest, ob ein Objekt ein Wort ist. Beispiel: `(string? "abc")` ⇒ `#t`

substring findet einen Teil eines Worts. Beispiel: `(substring "abcdef" 2 4)` ⇒ `"cde"`

symbol->string ermittelt den Namen eines Symbols. Beispiel: `(symbol->string 'apple)` ⇒ `"APPLE"`

symbol? stellt fest, ob ein Objekt ein Symbol ist. Beispiel: `(symbol? 'apple?)` ⇒ `#t`

T

truncate schneidet den Nachkommateil einer Zahl ab. Beispiel: `(truncate 4.2)` ⇒ `4`

try-to-eval funktioniert wie eval, erzeugt aber im Fehlerfall keinen Programm-Abbruch, sondern übergibt ein Fehler-Objekt. Beispiele:
`(try-to-eval 'nil)` ⇒ `()`
`(try-to-eval 'noel)` ⇒ `#<ERROR unbound 'NOEL'>`.

V

vector erzeugt einen neuen Vektor aus seinen Argumenten. Beispiel: `(vector 'a 'b)` ⇒ `#(a b)`

vector->list macht aus einem Vektor eine Liste. Beispiel: `(vector->list #(a b c))` ⇒ `(a b c)`

vector-length ermittelt die Zahl der Elemente eines Feldes. Beispiel: `(vector-length #(a b c))` ⇒ `3`

vector-ref ermittelt ein Element eines Feldes aus seiner Position. Beispiel: `(vector-ref #(a b c) 1)` ⇒ `b`

vector-set! überschreibt ein Element eines Feldes. Destruktiv. Beispiel: `(vector-set! v 0 'apple)` ⇒ `apple`. Seiteneffekt: das erste Element des Feldes v ist nun das Symbol `apple`.

vector? stellt fest, ob ein Objekt ein Vektor ist. Beispiel: `(vector? #(a b c))` ⇒ `#t`

W

write schreibt einen Ausdruck auf Bildschirm oder Port. Wörter erscheinen in Gänsefüßchen, z. B. "Wort", Buchstaben mit Raute und Schrägstrich, z. B. `#\b`. Kein Zeilenwechsel. Beispiel: `(write "abc")` ⇒ `#t`. Seiteneffekt: auf dem Bildschirm steht `"abc"`.

write-char schreibt einen Buchstaben ohne Raute und Schrägstrich auf Bildschirm oder Port. Beispiel: `(write-char #\a)` ⇒ `#t`. Seiteneffekt: auf dem Bildschirm steht `a`.

Z

zero? stellt fest, ob eine Zahl gleich Null ist. Beispiel: `(zero? 0)` ⇒ `#t`

Anhang E

Glossar

Debugging Neudeutsch: Suche nach Denkfehlern im Programm, mit Abstand die arbeitsintensivste Phase beim Programmieren. Fällt bei Lisp kürzer aus als bei Perl, entfällt aber nie ganz.

Entlausen StarckDeutsch: Debugging (siehe oben).

Hyperlink Neudeutsch: Vollelektronischer Ersatz für Wollfäden beim Buch in Papierform. Vorteile: geht schneller und bequemer; Nachteile: versagt bei Wollfäden in der Maus, also sofort in den Papierkorb damit!