D E K A L I S P Benutzerhandbuch

Inhaltsverzeichnis:

0. Einleitung

0.1 Besonderheiten von DEKALISP

DEKALISP ist ein sehr leistungsfähiges LISP-System für die 68000 Microprozessor - Zusatzkarte AP20 für den Apple II. Es ist weitgehend kompatibel zu INTERLISP <1> und besitzt unter anderem folgende für ein LISP-System nicht selbstverständlichen Eigenschaften:

- Eine Speicherverwaltung für den gesamten Adressraum von 16 MByte, die die verschiedenen Speicherbereiche dynamisch aufteilt und somit jegliche Einschränkungen - z.B der erlaubten Rekursionstiefe - vermeidet.

- Eine sehr schnelle Garbage-Collection für Listen, Atome und Strings.

- Unterstützung von upwards- und downwards-funargs, auch als closures bekannt. Dies erlaubt es, Funktionen als Argumente an andere Funktionon zu übergeben, oder als Wert einer Funktion zurückzugeben. Dabei können die jeweiligen Variablenbindungen der Funktionen erhalten bleiben. Das ermöglicht es z.B. Generatoren zu schreiben.

- Ein Macro-Mechanismus. Macros werden vor der Evaluierung erst noch expandiert.

- Ein extrem schneller Interpreter. Dies wird durch sogenanntes "shallow binding" der Variablen auf dem Stack ermöglicht. DEKALISP schlägt bei Benchmarks sogar die Interpreter die auf den speziell für LISP konstruierten LISP-Maschinen laufen.

- An Datentypen neben Listen und Atomen (literal und numerisch) auch Zeichenketten, die mit besonderen Stringfunktionen effizient verarbeitet werden können.

- Ein Paket von komfortablen Benutzerhilfen wie Strukturediter, Ablaufverfolgung und Prettyprinter.

0.2 Benötigte Hardware

DEKALISP benötigt mindestens einen Apple-kompatiblen Computer mit einer AP20 68000 CPU Karte mit 128 K RAM. Da DEKALISP jedoch die Verwendung des gesamten Adressbereichs des 68000-Prozessors unterstützt, kann auch zusätzlich mit einer oder mehreren AP26 Speicherkarten gearbeitet werden.

DEKALISP erlaubt den wahlweisen Betrieb mit dem 40-Zeichen-Bildschirmformat des Apples oder mit einer 80-Zeichen-Karte.

DEKALISP läuft unter DOS. Das bedeutet, das alle externen Speichermedien verwendet werden können, die von DOS ansprechbar sind. Ein Standard-Diskettenlaufwerk zum booten der Systemdiskette ist die Mindestvoraussetzung.

0.3 Zu diesem Benutzerhandbuch

0.3.1 Dies ist kein Lehrbuch

Dieses Handbuch ist ausschlie~lich nach systematischen Gesichtspunkten gegliedert und nicht nach didaktischen. In ihm ist die Bedienung von DEKALISP erl{utert und alle zur Verf}gung stehenden Funktionen werden beschrieben. Als LISP-Lehrbuch ist es jedoch v|llig ungeeignet. Dieses Handbuch ist zwar nicht f}r den Apple-Neuling ausgelegt, alles was von Ihnen erwartet wird ist jedoch nur die Vertrautheit mit der Tastatur und den Disketten.

Wir empfehlen Ihnen, sich ein Lehrbuch f}r LISP zu besorgen, sofern Sie nicht bereits mit dieser Sprache vertraut sind. Einige Literaturhinweise sind im Anhang zu finden. Kapitel 1 dieses Handbuches liefert die Informationen die notwendig sind, um die Beispiele aus dem gew{hlten Handbuch direkt am Computer auszuprobieren, was den Lerneffekt stark steigern d}rfte.

Leider gibt es f}r LISP keinen einheitlichen Standard. Einige h{ufig Auftretende Abweichungen vom von DEKALISP verwendeten INTERLISP-Dialekt sind in Kapitel 12 aufgef}hrt um erste Anpassungen von Programmen zu erm|glichen.

Alle von DEKALISP zur Verf}gung gestellten Funktionen werden in diesem Handbuch in den Kapiteln 2 bis 11 erl{utert. Es wurde versucht, sie nach m|glichst logischen Gesichtspunkten zu ordnen. Wenn Sie nach der Beschreibung einer Funktion suchen, deren Namen Ihnen bereits bekannt ist, so k|nnen Sie den Generalindex s{mtlicher Funktionen benutzen, der sich am Ende dieses Handbuches befindet. Zus{tzlich endet jedes Kapitel mit einem Kapitelindex.

0.3.2 Notationen und Konventionen

Einen wesentlichen Teil dieses Handbuches nehmen die Beschreibungen der DEKALISPFunktionen ein. Um diese so exakt und kurz wie m|glich zu halten, wird eine einheitliche Form benutzt, wie z.B:

```
(LIST x1 x2 .... xn)                                    eval, nospread
     LIST erzeugt eine Liste, deren toplevelElemente x1' bis xn'
     sind.

Beispiel:
(LIST '(A B) '(C D))
>  ((A B)(A B))
```

Eine Funktionsbeschreibung besteht also aus 4 Teilen:

einem Muster f}r den Funktionsaufruf, in dem der Funktionsname mit Gro~buchstaben geschrieben ist, und die Argumente mit Kleinbuchstaben. H{ufig haben die Argumente mnemonische Namen erhalten.

der Beschreibung des Funktionstyps, in diesem Beispiel eval,

nospread. eval bedeutet, da~ die Argumente erst evaluiert, und dann an die Funktion }bergeben werden. nospread bedeutet, da~ die Funktion beliebig viele Argumente erhalten darf. noeval und spread sind die jeweiligen Gegenteile.

einer verbalen Beschreibung der Funktion, der erforderlichen Argumente und des Resultates. In diesem Text wird auf die Argumente mit den Namen aus dem Muster Bezug genommen, wobei eventuell noch ein Apostroph auf den Namen folgt. In einem solchen Falle ist von dem evaluierten Argument die Rede.

eventuell einem oder mehreren Beispielen. Dabei f{ngt die Benutzereingabe in der ersten Spalte an und das von DEKALISP gelieferte Resultat ist hinter einem Pfeil > eingerückt.

1. Einführung in die Bedienung

1.1 Konfigurieren des Systems

Wenn Sie Ihr brandneues DEKALISP-System erhalten haben, so ist dieses noch nicht an Ihre spezielle Apple-Konfiguration angepasst. Statt dessen macht es folgende Annahmen, die nicht notwendigerweise korekt sein müssen:

- Ihre AP20 68000-Karte steckt in Slot 4 des Apples.
- Sie booten von Slot 6.
- Sie arbeiten mit dem 40-Zeichen Bildschirmformat des Apples.
- Ihr Drucker, falls vorhanden, wird über Slot 1 angesprochen.

Falls alle diese Punkte zutreffen, so brauchen Sie keine Konfigurierung vornehmen und können gleich in Kapitel 1.2 weiterlesen. Sollten Sie sich bei den ersten beiden Punkten nicht so sicher sein, so gehen Sie trotzdem zu Kapitel 1.2 und versuchen das LISP-System zu booten. Nur wenn das nicht klappt, müssen Sie hierher zurückkehren.

Die Größe des Speichers, der ihrer 68000 zur Verfügung steht, ob Sie also eine AP26 besitzen oder nicht, brauchen Sie dem LISP-System nicht mitzuteilen, das bestimmt es vor jedem Lauf selbstständig.

Da Sie dies noch lesen, nehmen wir an, daß Sie mit einer oder mehreren der vom LISP-System getroffenen Annahmen nicht zufrieden sind. Sie können sie mit dem auf der DEKALISP-Systemdiskette mitgelieferten Programm "LISP KONFIG" ändern. Dazu booten Sie bitte mit einer gewöhnlichen DOS-Diskette, legen die DEKALISP-Systemdiskette ein und führen dieses Programm mit RUN LISP KONFIG aus.

Auf dem Bildschirm erscheinen obige vier Punkte zusammen mit einem Cursor. Durch drücken der Leertaste bewegt sich dieser Cursor reihum von einem der Punkte zum nächsten. Wählen Sie sich einen Punkt aus, den Sie verändern wollen. Nun können Sie die Pfeiltasten benutzen, um die vom System getroffene Annahme zu ändern. Sie werden sofort begreifen, wie Sie das machen müssen und Sie werden auch merken, daß der Computer ein wenig dabei mitdenkt.

Nachdem Sie mit den Pfeiltasten eine Annahme korrigiert haben, können Sie entweder mit der Leertaste zu einer anderen Annahme springen, oder Sie drücken <RETURN>, falls Sie fertig sind. In diesem Falle werden Ihre Änderungen auf der Systemdiskette abgespeichert, die dazu natürlich noch immer im gleichen Laufwerk liegen muß.

Die Anpassung des LISP-Systems an Ihre Apple-Konfiguration ist jetzt beendet. Selbstverständlich kann Sie jederzeit wiederholt werden wenn sich an der Konfiguration etwas ändert, wenn Sie etwa nachträglich eine 80-Zeichen-Karte nachrüsten.

1.2 Starten von DEKALISP

Um das DEKALISP-System zu starten legen Sie lediglich die DEKALISP-Systemdiskette in Ihr Boot-Laufwerk ein und booten. Nach einiger Zeit meldet sich DEKALISP dann mit Versionsnummer und Copyright-Notice und nach weiterem Surren des Diskettenlaufwerkes erscheint dann auch ein \ und der Cursor.

Falls Sie eine andere Erfahrung machen, so kann das im wesentlichen 3 Gr}nde haben:

- Wenn sich DEKALISP meldet, der Cursor aber nicht wie beschrieben erscheint sobald das Diskettenlaufwerk aufh|rt zu laufen, so steckt der Controller f}r Ihr Boot-Laufwerk wahrscheinlich in einem anderen Slot, als das DEKALISP-System annimmt. Lesen Sie in Kapitel 1.1 was Sie dann machen m}ssen.

- Wenn sich DEKALISP nicht einmal meldet, so liegt das wahrscheinlich an ihrer AP2D. Sie steckt entweder in einem falschen Slot (siehe Kapitel 1.1) oder ist vielleicht sogar defekt.

- Es kann aber auch an der Systemdiskette selber liegen. Vielleicht sind die Daten auf ihr nicht mehr korrekt lesbar. Da dies nach l{ngerer der Benutzung der Systemdiskette wahrscheinlicher wird, sollten Sie sich sofort bei Erhalt der Diskette eine Kopie machen (aber das wissen Sie ja sicher schon). Kopien der Systemdiskette d}rfen Sie nat}rlich nicht an andere Personen weitergeben. Das verbietet das Urheberrecht (aber das wissen Sie ja erst recht).

Vielleicht gef{llt es Ihnen nicht, da~ Sie zum Starten von DEKALISP neu booten sollen? Sie brauchen das nicht zu tun; wenn Sie bereits DOS geladen haben, so gen}gt ein RUN DEKALISP.

Wir wollen Ihnen an dieser Stelle auch schon verraten, wie Sie DEKALISP wieder verlassen k|nnen. Das geht zum einen auf die brutale Weise mit RESET. Achten Sie also darauf, den RESET-Knopf nicht versehentlich zu dr}cken. Die sanfte Methode zum Verlassen von DEKALISP bedient sich der Funktion (EXIT). Wenn Sie diese Funktion aufrufen, so fragt DEKALISP sicherheitshalber nach, ob Sie es ernst meinen, denn wenn Sie DEKALISP erst einmal verlassen haben, so gibt es kein Zur}ck mehr. Sie sollten also vorher Ihre Daten und Funktionen abgespeichert haben. Lesen Sie dazu Kapitel 5.

1.3 Die Eingabesyntax

Wir wollen annehmen, da~ sich Ihr DEKALISP-System vorschriftsgem{~ gemeldet hat und das \ erschienen ist. Dieses \ ist das sogenannte Promtzeichen. Es will Ihnen signalisieren, da~ DEKALISP auf Ihre Eingaben wartet. Hinter dem \ steht ein wei~es Rechteck, der Cursor. Wundern Sie sich nicht, da~ der Cursor nicht blinkt, blinkende Cursors machen nur nerv|s.

Wo Sie jetzt wissen, da~ DEKALISP auf eine Eingabe wartet, werden Sie ihm sicher den Gefallen tun wollen. Also etwa:

```
( PLUS 1 1
```

Wenn Sie jetzt <RETURN> dr}cken, tut DEKALISP noch gar nichts, sondern es erscheint in der n{chsten Zeile wieder das \. Das liegt an der ge|ffneten runden Klammer. DEKALISP wartet grunds{tzlich solange, bis Sie alle runden Klammern geschlossen haben. Geben Sie also

```
)
```

ein (und dr}cken Sie <RETURN>). DEKALISP antwortet mit 2 und, arbeitsbeflissen wie es ist, mit einem neuen \. Es klappt also. Versuchen Sie es mit

```
(PLUS 1 2) (PLUS 2 3)
```

DEKALISP druckt 3 und 5. Sie k|nnen also mehrere Befehle auf einmal eingeben. Auch

```
(PLUS 1 2) (PLUS
2 3)
```

funktioniert.

Wenn Sie LISP bereits kennen, so wissen Sie bereits, da~ ein Klammernpaar und das darin eingeschlossene als Liste bezeichnet wird und da~ LISP eine solche Liste als Funktionsaufruf interpretiert. In unserem Beispiel ist (PLUS 1 2) ein Funktionsaufruf, PLUS ist die Funktion und 1 und 2 sind die Argumente.

Der ganze Dialog mit LISP besteht also darin, da~ Sie Funktionen mit Argumenten aufrufen und LISP die Funktion auf die Argumente anwendet und das Resultat zur}ckgibt. Wenn Sie das als Einschr{nkung empfinden, dann kennen Sie LISP nicht.

Die Argumente der Funktionen d}rfen ihrerseits wieder Funktionsaufrufe sein, also beispielsweise

```
(PLUS 1 (PLUS 2 (PLUS 3 4)))
```

Das Ergebnis ist 10. Hier tritt bereits die bekannteste und zugleich unbedeutenste Eigenschaft von LISP in Erscheinung: In LISP wimmelt es nur so vor Klammern. Daher der Name LISP: Lots of Irritating Single Parentheses, der darauf hinweist, da~ man leicht vergessen kann, eine ge|ffnete Klammer auch wieder zu schlie~en. (Es gibt Ger}chte, eigentlich st{nde LISP f}r List Processing Language.)

Durch zwei Eigenschaften hilft DEKALISP Ihnen, mit den vielen Klammern zurechtzukommen. Die eine haben Sie bereits kennengelernt: DEKALISP wartet geduldig, bis Sie die letzte Klammer geschlossen haben, auch wenn Sie vorher schon <RETURN> dr}cken. Die andere betrifft die Verwendung der spitzen Klammern < und >. Eine rechte spitze Klammer wird die letzte linke spitze Klammer geschlossen, zus{tzlich aber auch alle dazwischenliegenden runden linken Klammern.

```
(PLUS 1 <PLUS 2 (PLUS 3 4>)
```

liefert also auch 10. Ebenso tut es

```
(PLUS 1 (PLUS 2 (PLUS 3 4>
```

denn wenn zu einer rechten spitzen Klammer keine linke spitze Klammer geh|rt, so werden einfach alle noch offenen Klammern geschlossen. Diese sogenannten Metaklammern werden Sie noch sehr zu sch{tzen lernen.

Nun noch einige Details:

Leerzeichen haben in LISP die Funktion von Trennzeichen, sie trennen Atome voneinander. Die gleiche Wirkung haben in DEKALISP die Zeilenenden. Es ist also nicht m|glich, einen Atomnamen teilweise am Ende der einen Zeile und teilweise am Anfang der n{chsten zu haben. DEKALISP macht daraus zwei Atome.

Punkte als auch Kommata k|nnen bei der Erzeugung von dotted-pairs benutzt werden. Sie m}ssen dabei jedoch durch ein Trennzeichen von Atomnamen getrennt sein, um als deren Teil interpretiert zu werden.

In DEKALISP werden Anf}hrungsstriche " benutzt, um den Anfang und das Ende von Strings zu kennzeichnen. Innerhalb von Strings verlieren Trennzeichen ihre Bedeutung.

Da einigen Zeichen, wie zum Beispiel dem Leerzeichen oder den Anf}hrungsstrichen, bei der Eingabe besondere Bedeutungen zugeordnet sind, k|nnen Sie sie nicht ohne weiteres innerhalb von Atomnamen verwenden. Diese Einschr{nkung k|nnen Sie jedoch umgehen, indem Sie diesen Sonderzeichen das Prozentzeichen % voranstellen. Dadurch verlieren sie ihre spezielle Bedeutung. Durch %% kann auf diese Weise auch das Prozentzeichen in Atomnamen vorkommen.

Wenn Sie DEKALISP benutzen, werden Sie h{ufig den Funktionsaufruf

```
(QUOTE Argument)
```

ben|tigen, mit dem Sie die Evaluierung unterbinden k|nnen. DEKALISP stellt daf}r die Abk}rzung

```
' Argument
```

zur Verf}gung, die Ihnen wegen der H{ufigkeit des Auftretens viel Tipparbeit ersparen wird.

Die Eingabesyntax von DEKALISP ist Ihnen jetzt im wesentlichen bekannt: geschachtelte Listen sprich Funktionsaufrufe, in denen sich linke und rechte runde Klammern die Waage halten m}ssen, oder in denen Metaklammern das gleiche bewirken. Was aber wenn Sie sich vertippen? Dann hilft Ihnen:

1.4 Der Bildschirmeditor

Der Bildschirmeditor von DEKALISP ist grunds{tzlich in Aktion und reagiert auf Ctrl-Befehle. Das sind Befehle, die durch gleichzeitiges Dr}cken der <Control>-Taste und einer Buchstabentaste eingegeben werden.

1.4.1 Der Zeileneditor

Der Bildschirmeditor enth{lt einen gew|hnlichen Zeileneditor. Er reagiert auf: Pfeil rechts, Pfeil links, Ctrl-I, Ctrl-T und Ctrl-D. (Ctrl-I z.B. bedeutet gleichzeitiges dr}cken der <Control>-Taste und der <I>-Taste).

Mit den Pfeil-Tasten steuern Sie den Cursor. An der neuen Position des Cursors k|nnen Sie wie gewohnt Zeichen eingeben, wobei eventuell bereits dort stehende Zeichen }berschrieben werden.

Wollen Sie das }berschreiben verhindern, so dr}cken Sie Ctrl-I (I f}r Insert). Die Zeichen, die Sie nun tippen, werden vor dem unter dem Cursor stehenden Zeichen eingef}gt. Dabei verl{ngert sich nat}rlich die Zeile. Eventuell werden einige Zeichen }ber den rechten Bildschirmrand hinausgeschoben. Sie sind dann verloren. Der Einf}ge-Modus bleibt erhalten, bis sie erneut Ctrl-I (oder einen andere Ctrl-Befehl) tippen.

Mit Ctrl-D k|nnen Sie das Zeichen unter dem Cursor l|schen. Die dahinter liegenden Zeichen rutschen dann eine Position weiter nach vorne, so da~ das n{chste Zeichen unter dem Cursor zu liegen kommt. Jetzt k|nnen Sie Ctrl-D erneut dr}cken, solange bis alle zu l|schenden Zeichen verschwunden sind.

Wenn Sie alle Korrekturen vorgenommen haben, so k|nnen Sie die Zeile mit <RETURN> abschicken. Dabei spielt es keine Rolle, an welcher Position innerhalb der Zeile sich der Cursor gerade befindet. Es wird nicht, wie in BASIC oder CP/M, der Teil der Zeile hinter dem Cursor gel|scht. Wenn Sie das jedoch w}nschen, k|nnen Sie es mit Ctrl-T erreichen.

Am besten spielen Sie jetzt ein bi~chen mit diesem Zeileneditor herum, um ihn richtig kennenzulernen. Machen Sie in Ihrer Eingabe einen Fehler, gehen Sie mit den Pfeiltasten an die Position des Fehlers und korrigieren Ihn je nach Situation durch }berschreiben, Einf}gen oder L|schen.

1.4.2 Wiederverwendung von Zeilen

Der Zeileneditor ist n}tzlich, solange Sie eine Zeile noch nicht mit <RETURN> beendet haben. Nachher ist es zu sp{t, auch wenn Sie noch nicht alle Klammern geschlossen haben. Obwohl DEKALISP noch nicht auf Ihre fehlerhafte Eingabe reagiert, gibt es keine M|glichkeit mehr, sie noch r}ckg{ngig zu machen. Lassen Sie sie ruhig ausf}hren und sich notfalls eine Fehlermeldung geben.

Nun werden Sie die gleiche Eingabe nat}rlich in fehlerloser Form erneut eingeben wollen. Aber sicherlich finden Sie es {rgerlich, das gleich erneut einzutippen, das einige Zeilen weiter oben auf dem Bildschirm bereits steht, lediglich mit einem kleinen Fehler drin.

DEKALISP h{lt daf}r etwas Komfort f}r Sie bereit: mit Ctrl-O k|nnen Sie den Cursor nach oben bewegen, bis er auf der fehlerhaften Zeile steht. Diese brauchen Sie nur noch mit dem Ihnen bereits vertrauten Zeileneditor zu korrigieren und mit <RETURN> abzuschicken.

Der unter der Zeile liegende Bildschirm enth{lt nun jedoch bereits Text und dieser wird im folgenden lediglich mit den neuen Resultaten und Eingaben }berschrieben. Es ist also meistens sinnvoll, nach dem Benutzen von Ctrl-O mit Ctrl-C alle Zeilen unter

der, auf der der Cursor steht, zu löschen.

Vielleicht wollen Sie die weiter unten auf dem Bildschirm stehenden Zeilen aber auch dort belassen, denn genauso, wie sie mit Ctrl-O weiter oben stehende Zeilen wiederverwenden können, so können Sie mit Ctrl-L weiter unten stehende Zeilen wiederverwenden.

Dieses Konzept des Bildschirmeditors ist leistungsfähiger, als Sie es bisher vorgestellt bekommen haben. Sie werden das beim weiteren Umgang mit DEKALISP feststellen, wenn Sie sich folgendes Grundprinzip des Bildschirmeditors immer vor Augen halten: jedesmal, wenn Sie <RETURN> drücken, wird die Zeile, in der der Cursor gerade steht an DEKALISP abgeschickt, unabhängig davon, ob Sie sie gerade neu oder bereits früher eingetippt haben, oder ob Sie ein Resultat einer vorangegangenen Berechnung darstellt.

Die eventuell in der ersten Spalte des Bildschirmes stehenden Promtzeichen, (außer \ gibt es auch noch - und :) können Sie übrigens getrost ignorieren, DEKALISP tut es auch.

1.4.3 Aktivierung des Druckers

Normalerweise spielt sich der Dialog mit DEKALISP nur auf dem Bildschirm ab. Falls Sie jedoch alles auch schwarz auf weiß haben wollen, so können Sie den Drucker parallel alles drucken lassen, was Sie eingeben oder DEKALISP ausgibt. Dieses Verhalten schalten Sie mit Ctrl-P ein und bei nochmaligem drücken auch wieder aus. Das kennen Sie vielleicht aus CP/M.

Wohlgemerkt: Es wird alles gedruckt, was Sie eingeben und nicht alles, was auf dem Bildschirm erscheint. Das führt dazu, daß der Ausdruck auch noch dann aufschlußreich bleibt, wenn Sie viel mit dem Bildschirmeditor arbeiten.

Wie Sie den Drucker unter Programmkontrolle ein- und ausschalten, lesen Sie in Kapitel 5.

1.4.4 Die Befehle des Bildschirmeditors

Hier noch einmal alle Befehle auf einen Blick:

Pfeil links	-	Cursor eine Zeichenposition nach links
Preil rechts	-	Cursor eine Zeichenposition nach rechts
Ctrl-O	-	Cursor eine Zeile nach oben
Ctrl-L	-	Cursor eine Zeile nach unten
Ctrl-I	-	Einfügemodus einschalten bzw. ausschalten
Ctrl-D	-	Zeichen löschen
Ctrl-T	-	Ende der Zeile löschen
RETURN	-	Zeile an DEKALISP abschicken
Ctrl-C	-	Rest des Bildschirms löschen
Ctrl-S	-	Ausdruck anhalten

Dieser letzte Befehl wurde noch nicht erläutert. Er bewirkt, wie übrigens auch z.B im BASIC, daß ein gerade laufender Ausdruck unterbrochen wird, bis erneut eine Taste gedrückt wird. Hiermit

k|nnen Sie sich Lesepausen bei l{ngeren Ausdrucken verschaffen.

1.5 Im Falle von Fehlern

Sie sind es gewohnt, da~ DEKALISP auf Ihre Eingaben mit einem Wert und dem Promptzeichen \ antwortet. Ziemlich bald werden Sie jedoch durch eine andere Reaktion }berrascht werden: DEKALISP gibt ein "Piep" von sich, druckt eine Fehlermeldung aus und zeigt dann ein spezielles Promptzeichen (ein -) an.

Dies ist in keinster Weise verh{ngnisvoll, es bedeutet nur, da~ DEKALISP Ihre Eingabe nicht auswerten konnte. Sie k|nnen es einfach ignorieren und in gewohnter Weise weiterarbeiten, z.B die Eingabe mit dem Bildschirmeditor korrigieren und erneut abschikken. Wenn Sie allerdings das ge{nderte Promptzeichen irritiert, so k|nnen Sie mit

```
(TOPLEVEL)
```

aus der Ausnahmesituation zur}ckspringen. Das gewohnte \ erscheint wieder.

Was die Fehlermeldungen bedeuten und in welchen F{llen sie auftreten, k|nnen Sie in Kapitel 9 nachlesen. Dort steht auch, was es mit dem ge{nderten Promtzeichen auf sich hat. Das m}ssen Sie aber erst wissen, wenn Sie Programme debuggen wollen. Die Fehlermeldungen sind }brigens in Englisch, um besser zu den ebenfalls englischen LISP-Funktionsnamen zu passen.

Wenn Sie Programme oder rekursive Funktionen laufen lassen, kann es Ihnen auch passieren, da~ sich DEKALISP trotz l{ngeren Wartens }berhaupt nicht wieder meldet. Das ist ein Zeichen daf}r, da~ Sie entweder eine Endlosschleife programmiert haben, oder aber an eine rekursive Listenfunktion eine zirkul{re Liste }bergeben haben. Was es auch seien mag, Sie brauchen nicht f}r ewig auf DEKALISP zu warten, sondern k|nnen es durch Dr}cken von Ctrl-Q zur}ckrufen. Es unterbricht sich auf der Stelle und meldet sich mit dem Promptzeichen -. Jetzt gilt das bereits oben Gesagte; Sie k|nnen entweder auf Fehlersuche gehen (siehe Kapitel 9) oder normal weiterarbeiten.

Die Unterbrechungsm|glichkeit durch Ctrl-Q k|nnen Sie auch dann nutzen, wenn DEKALISP versucht, eine zirkul{re Liste auszudrucken. In diesem Falle bricht DEKALISP sofort den Ausdruck ab, geht aber nicht in die Fehlerbehandlungsroutine. Statt dessen gibt die aufgerufene Druckfunktion, z.B PRINT, ihr evaluiertes Argument an die aufrufende Funktion zur}ck, ganz so, als ob nichts geschehen w{re.

Kapitelindex

2. Listen

Wie Sie wissen, sind Listen die wichtigste Datenstruktur in LISP. (Wenn Sie das nicht wissen, so m}ssen Sie dringendst ein Lehrbuch f}r LISP lesen. Dieses Handbuch kann nicht als Einf}hrung dienen, da es die verschiedenen Funktionen nur knapp und pr{zise erl{utert und ein gewisses Vorwissen voraussetzt.)

Wegen der grundlegenden Bedeutung der Listen, stellt DEKALISP eine gro~e Anzahl von Funktionen f}r den Umgang mit Listen zur Verf}gung. Diese sind im folgenden in vier Gruppen eingeteilt: Funktionen zur Manipulation von Listen, Funktionen zum selektieren von bestimmten Elementen oder Teilen von Listen, Pr{dikatsfunktionen und Funktionen, die mit Propertylisten arbeiten. Ein direktes Auffinden von Beschreibungen spezieller Funktionen erm|glicht der Kapitelindex am Ende dieses Kapitels.

2.1 Manipulations

Die grundlegende Funktione zum Aufbauen von Listen ist:

(CONS x y) eval, spread

CONS erzeugt ein dotted-pair, d.h eine neue LISP-Zelle, mit den Elementen x' und y'. Wenn y' eine Liste ist, so wird x' als erstes Element vor die Liste gehängt.

Beispiel:
```
(CONS 'A 'B)
---> (A . B)
(CONS '(A B) '(C D))
---> ((A B) C D)
```

Mit den folgenden beiden Funktionen ist ein direkter Eingriff in die Bestandteile eines LISP-Knoten, den CAR- und den CDR-Teil, m|glich. Sie erlauben es, bereits bestehende Listenstrukturen nachtr{glich zu ver{ndern. Ein solches Vorgehen hat eine Menge von Nebenwirkungen, da alle Listen, die die auf diese Weise ge{nderte enthalten, sich ebenfalls {ndern. Die Anwendung der beiden Funktionen und generell auch aller anderen Funktionen, die bestehende Listen ver{ndern, ist deshalb sehr riskant und es erschwert die Fehlersuche. Sie sind jedoch trotzdem wegen ihrer effizienz von gro~er Bedeutung.

(RPLACA x y) eval, nospread

REPLACA (replace CAR) ersetzt den linken oder CAR-Teil der LISP-Zelle (dotted-pair) x' durch y'. (CAR x) wird also nachher y' ergeben.
Wenn x' ein literales Atom ungleich NIL ist, so wird durch RPLACA der globale Wert von x' auf y' gesetzt. Da NIL immer sich selbst als Wert haben soll, ist (RPLACA NIL y) nicht erlaubt, sondern liefert die Fehlermeldung

ATTEMPT TO RPLAC NIL.

Der Wert von RPLACA ist die ver{nderte Liste x', beziehungs-

weise das literale Atom x'.

Beispiel:
```
(SETQQ X (A B C))
--->  (A B C)
(SETQ Y (CONS 'D X))
--->  (D A B C)
(RPLACA X '(E F))
--->  ((E F) A B C)
X
--->  ((E F) A B C)
Y
--->  (D (E F) A B C)
(RPLACA X X)
--->  ((((((((((  ........ A B C) A B C) A B C)
```
Die Ergebnisliste ist eigentlich unendlich tief geschachtelt. Ausgedruckt werden nur soviele Ebenen, wie durch PRINTLEVEL spezifiziert ist (siehe Kapitel 5).

(RPLACD x y) eval, nospread

REPLACD (replace CDR) ist das Pendant zu REPLACA. Es ersetzt den rechten oder CDR-Teil der LISP-Zelle x' durch y'.
In dem Fall, da~ x' ein literales Atom ist, bewirkt RPLACD, da~ y' zur Propertyliste von x' wird (siehe 2.4). Auch hier darf x' nicht NIL sein. Fehlermeldung wie oben.
Der Wert von RPLACD ist die verÄnderte Liste x', beziehungsweise das literale Atom x'.

Beispiel:
```
(RPLACD '(A B C) '(D E))
--->  (A D E)
```

Vier Funktionen zum hintereinanderhÄngen von Elementen zu einer Liste:

(LIST x1 x2 .... xn) eval, nospread

LIST erzeugt eine Liste, deren top-level-Elemente x1' bis xn' sind.

Beispiel:
```
(LIST '(A B) '(C D))
--->  ((A B)(C D))
```

(APPEND x1 x2 .... xn) eval, nospread

APPEND liefert eine Liste der top-level-Elemente der Listen x1' bis xn'. Dazu werden die top-level-Elemente der Listen x1' bis xn-1' kopiert, alle tiefer liegenden Zellen bleiben unkopiert. Falls eins der Argumente x1' bis xn-1' keine Listen sind, so werden sie ignoriert. Ist xn' keine Liste, so endet die von APPEND erzeugte Liste mit einem dotted-pair.

Beispiel:
```
(APPEND '(A B) '(C D))
--->  (A B C D)
```

```
(APPEND '((A) B) 'C '(D E) 'F)
--->  ((A) B D E . F)
(APPEND '(A B . C) '(D E))
--->  (A B D E)
```

(NCONC x1 x2 .... xn) eval, nospread

NCONC liefert wie APPEND eine Liste der top-level-Elemente der Listen x1' bis xn'. Es arbeitet jedoch destruktiv, d.h. es erzeugt keine neuen LISP-Zellen, sondern ändert lediglich die Zeiger in den alten.

Beispiel:

```
(SETQQ X (A B))
--->  (A B)
(SETQQ Y ((C)))
--->  ((C))
(NCONC X Y)
--->  (A B (C))
X
--->  (A B (C))

(NCONC X X)
>  (A B (C) A B (C) A B (C) A ..........
```

hier erzeugt NCONC eine Zirkulärliste.

(NCONC1 liste x) eval,spread

NCONC1 hängt an liste' x' als top-level-Element an, indem es den CDR-Teil der letzten LISP-Zelle von liste' verändert. Die so entstehende Liste ist der Wert von NCONC1. Die gleiche Wirkung könnte man mit (NCONC liste (LIST x)) erreichen.

Beispiel:

```
(SETQQ X '(A B))
--->  (A B)
(NCONC1 X '(C D))
--->  (A B (C D))
X
--->  (A B (C D))
```

Zum Anfügen eines neuen Elementes an den Anfang einer Liste kann neben CONS auch ATTACH benutzt werden:

(ATTACH x liste) eval,spread

Durch ATTACH wird x' als neues erstes Element vor die Liste liste' gehängt und die so entstehende Liste wird als Wert zurückgegeben. liste' muß eine Liste sein, sonst gibt es eine Fehlermeldung. ATTACH verändert die Liste liste'.

ATTACH arbeitet nach einem anderen Prinzip, wie das folgende Beispiel zeigen soll:

```
(SETQQ X (A B C))
--->  (A B C)
(CONS 'D X)
--->  (D A B C)
```

```
X
--->  (A B C)
(ATTACH 'D X)
--->  (D A B C)
X
--->  (D A B C)
Es gilt also:
(EQ y (CONS x y))  --->  NIL
(EQ y (ATTACH x y))  --->  T
```

So wie diese gibt es viele DEKALISP-Funktionen, die ihre Argumente verändern. Aus diesem und anderen Gründen ist es oft sinnvoll, eine Kopie einer Liste zu erzeugen mit:

(COPY liste) eval, spread

COPY erzeugt eine Kopie von liste', die bzgl. EQUAL gleich zu ihr ist, nicht jedoch bzgl. EQ. Es werden alle LISP-Zellen von liste' kopiert, nicht jedoch eventuell enthaltene Zeichenketten.

Einige Listenfunktionen gibt es in zwei Versionen, von denen die eine ihre Argumente verändert, die sogenannte destruktive Version, und die andere nicht. Dies gilt z.B für:

(REMOVE x liste) eval, spread

REMOVE erzeugt eine neue Liste, die alle top-level-Elemente von liste' in der alten Reihenfolge enthält, die nicht im Sinne von EQUAL gleich zu x' sind. Der Wert von liste' bleibt unverändert.

(DREMOVE x liste)

DREMOVE ist die destruktive Version von REMOVE. Sie löscht alle top-level-Elemente in liste', die EQUAL zu x' sind. Dies geschieht durch verändern der Zeiger in liste', so daß sich deren Wert ändert. Der Wert von DREMOVE ist die derart modivizierte Liste.

(REVERSE liste) eval, spread

REVERSE liefert eine Liste, die alle top-level-Elemente von liste' in umgekehrter Reihenfolge enthält. Dazu werden die LISP-Zellen der obersten Ebene neu erzeugt.

(DREVERSE liste) eval, spread

DREVERSE liefert wie REVERSE liste' mit umgedrehter Reihenfolge der top-level-Elemente. Im Gegensatz zu REVERSE werden jedoch keine neuen LISP-Zellen erzeugt. liste' wird dabei verändert, ist jedoch nicht gleich dem Wert von DREVERSE.

Beispiel:

```
(SETQQ X (A B C))
--->  (A B C)
(DREVERSE X)
--->  (C B A)
X
--->  (A)
```

2.2 Selektion

Die beiden wichtigsten Funktionen mit denen man auf Teile von Listen zugreifen kann, sind CAR und CDR:

(CAR x) eval, spread

CAR greift direkt auf den Inhalt des linken Teils des LISP-Knotens x' zu und liefert ihn als Wert. Dies ist insbesondere:

- wenn x' eine Liste ist: das erste top-level-Element der Liste.
- wenn x' ein dotted-pair ist: der linke Teile dieses dotted-pairs.
- wenn x' ein literales Atom ist (eigentlich keine LISP-Zelle, hier aber dennoch erlaubt): der globale Wert des Atoms (siehe Kapitel 3).

Wenn keiner dieser Fälle zutrifft, wenn x' also ein numerisches Atom oder eine Zeichenkette ist, so wird eine Fehlermeldung gegeben.

Beispiel:
```
(CAR '(A B C))
---> A
(CAR '(A . B))
---> A
```

(CDR x) eval, spread

CDR liefert den rechten Teil des LISP-Knotens x'. Das bedeutet:

- wenn x' eine Liste ist: der Rest der Liste x' ohne das erste top-level-Element. Dieser Rest ist NIL, wenn x' eine Liste mit nur einem Element ist.
- wenn x' ein dotted-pair ist: den rechten Teil des dotted-pairs.
- wenn x' ein literales Atom ist: die Propertyliste des Atoms (siehe 2.4).

In allen anderen Fallen (x' numerisches Atom oder Zeichenkette) wird eine Fehlermeldung gegeben.

Beispiel:
```
(CDR '(A B C))
---> (B C)
(CDR '(A . B))
---> B
```

CAR und CDR sind in gewisser Weise das Gegenstück zu CONS. CONS baut einen LISP-Knoten auf und CAR und CDR zerpflücken ihn:

```
(CAR (CONS 'A 'B))
---> A
(CDR (CONS 'A 'B))
---> B
```

```
(CONS (CAR '(A B)) (CDR '(A B)))
--->  (A B)
```

CAR und CDR dienen dazu, das erste Element bzw. den Rest einer Liste zu erhalten. Oft ist man jedoch z.B an dem dritten Element oder allen Elementen au"er den ersten vier interessiert. Solche Werte k|nnen durch mehrmalige Anwendung von CAR und CDR erzeugt werden. Die gleiche Wirkung l{"t sich durch folgende Funktionen erzielen:

```
(CAAR x)    statt    (CAR (CAR x))
(CADR x)    statt    (CAR (CDR x))
(CDAR x)    statt    (CDR (CAR x))
(CDDR x)    statt    (CDR (CDR x))
(CAAAR x)   statt    (CAR (CAR (CAR x)))
    .
    .
    .
(CDDDDR x)    statt    (CDR (CDR (CDR (CDR x))))
```

Diese 28 Funktionen haben Namen, die zwischen einem C und einem R eine vierstellige Kombination von A's und D's einschlie"en. Aus diesen Namen l{"t sich ablesen, welche CAR-CDR-Kombination sie bewirken.

Beispiel:

```
(SETQQ X (A ((B) C) D E)
--->  (A ((B) C) D E)
(CADR X)
--->  ((B) C)
(CAAADR X)
--->  B
(CDDDR X)
--->  (E)
(CDDDDR X)
--->  NIL
```

Au"er den CAR-CDR-Kombinationen gibt es noch viele weitere M|glichkeiten auf bestimmte Elemente oder Teillisten einer Liste zuzugreifen:

(LAST liste) eval, spread

LAST liefert das letzte top-level-Element von liste'. Wenn die Liste mit einem dotted-pair endet, so ist dies der Wert von LAST. LAST gibt NIL zur}ck, wenn liste' kein Liste ist.

(LASTN liste n) eval, spread

LASTN liefert eine Liste, die die letzten n' top-level-Elemente (dotted-pair am Ende gilt als top-level-Element) von liste' enth{lt. Davor ist als erstes Element der Ergebnisliste noch eine Liste der Anfangselemente von liste' geh{ngt, die nicht unter den n' letzten sind. Diese Liste wird NIL, wenn n' gleich 1 oder 0 ist. Als nulltes Element einer Liste wird NIL genommen. Negative n' f}hren zu einer

Fehlermeldung. Wenn n' gr|~er ist, als liste' Elemente enth{lt oder liste' keine Liste ist, so ist der Wert von LASTN NIL.

Beispiel:
```
(LASTN '(A B C D E) 3)
--->  ((A B) C D E)
(LASTN '(A B) 0)
--->  (NIL NIL A B)
```

(NTH liste n) eval, spread

NTH ergibt die Teilliste von liste', die mit dem n'ten Element beginnt. Hat liste' weniger als n' Elemente, so gibt NTH NIL zur}ck. Das nullte Element wird als NIL angesehen, negative n' sind nicht erlaubt. Den gleichen Effekt wie (NTH liste n) hat (CADR (LASTN liste n)).

Beispiel:
```
(NTH '(A B C D E) 3)
--->  (C D E)
```

Listen k|nnen auch verwendet werden, um Mengen darzustellen, also nicht angeordnete Sammlungen von lauter verschiedenen Elementen. Von solchen Listen bilden die beiden folgenden Funktionen die Vereinigung bzw. den Durchschnitt.

(INTERSECTION liste1 liste2) eval, spread

INTERSECTION bildet den Durchschnitt von liste1 und liste2, wenn diese zwei Mengen darstellen. Dazu wird eine Liste erzeugt, die alle top-level-Elemente aus liste1' enth{lt, die auch in liste2' vorkommen. Atome gelten als leere Mengen. Wenn liste1 und liste2 identisch sind, so enth{lt das Ergebnis von INTERSECTION kein top-level-Element doppelt.

Beispiel:
```
(INTERSECTION '(A B C) '(C E A F))
--->  (A C)
(INTERSECTION '(A A B C) '(A B))
--->  (A A B)
(SETQQ X '(A A B))
--->  (A A B)
(INTERSECTION X X)
--->  (A B)
```

(UNION liste1 liste2) eval, spread

2.3 Pr{dikatsfunktionen

Pr{dikatsfunktionen sind Funktionen, mit denen Bedingungen getestet werden k|nnen. Sie liefern den Wahrheitswert einer bestimmten Bedingung. Dabei gilt die Konvention, da~ NIL den Wahrheitswert "falsch" und alles andere den Wahrheitswert "wahr" hat. Normaler-

weise liefern die Prädikatsfunktionen das Atom T um Wahrheit zu symbolisieren, die erweiterte Konvention erlaubt es den Funktionen jedoch, gleichzeitig noch interessante Daten zurück zu liefern, wenn das Prädikat wahr ist. Sie können für den Wahrheitswert "falsch" auch das Atom F benutzen, da dieses nämlich zu NIL evaluiert.

(NOT x) eval, spread
(NULL x) eval, spread

Diese beiden Funktionen negieren einen Wahrheitswert, liefern also T, wenn x' gleich NIL ist, und NIL in allen anderen Fällen.

Im folgenden werden die Prädikatsfunktionen beschrieben, die sich auf Listen beziehen.

(LISTP x) eval, spread

LISTP evaluiert zu x', wenn x' eine Liste ist, sonst zu NIL. Als Liste gilt jede Struktur, die durch mehrere Aufrufe von CONS erzeugt werden könnte. Somit sind außer Atomen und Zeichenketten alle LISP-Objekte Listen.

(NLISTP x) eval, spread

NLISTP evaluiert zu T, wenn x' keine Liste ist, sonst zu NIL.

(EQUAL x y) eval, spread

EQUAL dient dazu, die Gleichheit der beiden LISP-Objekte x' und y' zu testen. Dabei können folgende Fälle eintreten:

- x' oder y' ist literale Atome. Sie sind gleich, wenn sie den gleichen Printnamen haben (siehe 3.1.1).
- x' oder y' ist numerische Atome. Sie sind gleich, wenn ihr Zahlenwert übereinstimmt.
- x' oder y' ist Zeichenketten. Sie sind gleich, wenn ihre Zeichenfolgen identisch sind.
- x' und y' sind Listen. Sie sind gleich, wenn sie durch die gleiche Folge von CONS-Aufrufen aus den gleichen Atomen und Zeichenketten aufgebaut sind. Als Kriterium für die Gleichheit kann auch gelten, daß der durch (PRINT x) erzeugte Ausdruck identisch zu dem von (PRINT y) erzeugten ist.

Wenn nach obigen Kriterien x' und y' gleich sind, so evaluiert EQUAL zu T, sonst zu NIL.

Wenn EQUAL als Argumente zwei gleiche zirkuläre Listen übergeben bekommt, so kann es diese Gleichheit nicht feststellen, sondern hängt sich statt dessen in einer Endlosschleife auf. In diesem Falle hilft nur eine Unterbrechung von der Tastatur aus durch Drücken von Ctrl-Q. (siehe Kapitel 9)

Wenn zwei Listen mit EQUAL verglichen werden, so müssen diese Listen vollständig rekursiv durchlaufen werden, um einen Ver-

gleich auf allen Ebenen durchzuführen. Es gibt eine Funktion, die das nicht tut, sondern lediglich überprüft, ob die beiden Listen an der gleichen Stelle im Speicher stehen, in welchem Fall sie natürlich zwangsläufig gleich sind. In Fällen, wo es interessiert, ob zwei Listen die selben und nicht nur die gleichen sind, sollte also verwendet werden:

(EQ x y) eval, spread

EQ überprüft, ob die Zeiger auf die beiden LISP-Objekte x' und y' gleich sind. Für Atome bedeutet das das gleiche wie der von EQUAL durchgeführte Test, Listen und Zeichenketten müssen jedoch nicht nur gleich, sondern sogar identisch sein in dem Sinne, daß sie an der gleichen Stelle im Speicher abgelegt sind. Wenn dieser Test positiv verläuft, gibt EQ T zurück, sonst NIL.

Beispiel:
```
(SETQQ X '(A B C))
--->  (A B C)
(EQUAL '(A B C) '(A B C))
--->  T
(EQ '(A B C) '(A B C))
--->  NIL
(EQ X X)
--->  T
```

(EQP x y) eval, spread

EQP hat exakt die gleiche Wirkung wie EQ. DEKALISP stellt es nur zur Verfügung, da es in INTERLISP vorhanden ist.

(NEQ x y) eval, spread

NEQ evaluiert zu T, wenn die Zeiger auf x' und y' gleich sind und sonst zu NIL. Es ist damit genau die Umkehrung von EQ, entspricht also (NOT (EQ x y)).

(MEMBER x liste) eval, spread

MEMBER sucht in liste' nach einem top-level-Element, das im Sinne von EQUAL gleich zu x' ist. Wenn kein solches existiert, liefert MEMBER NIL, sonst den mit dem ersten Auftreten von x' beginenden Rest von liste'.

(MEMB x liste) eval, spread

MEMB ist im wesentlichen identisch mit MEMBER, führt den Vergleich der top-level-Elemente von liste' mit x' jedoch über EQ durch.

2.4 Propertylisten

Mit jedem literalen Atom au"er NIL ist eine Propertyliste verbunden. Di Erzeugen des Atoms auf NIL initialisiert. Auf Sie kann direkt mit den Funktionen CDR und RPLACD zugegriffen werden:

(CDR atom) eval, spread

Wenn atom' ein literales Atom ist, so liefert CDR die gesamte Propertyliste von atom'. Normalerweise wirkt CDR jedoch auf Listen (siehe Kapitel 2.2).

(RPLACD atom liste) eval, spread

Wenn atom' ein literales Atom ungleich NIL ist, so wird die Propertyliste von atom' durch liste' ersetzt. Auch RPLACD hat eine andere Bedeutung, wenn atom' kein literales Atom ist (siehe Kapitel 2.1).

Normalerweise arbeitet man mit Propertylisten jedoch mit speziellen Propertylistenfunktionen zum Setzen, Abfragen und L|schen bestimmter Eigenschaften (Propertys). Von diesen Funktionen wird folgendes Format f}r die Propertylisten verwendet:

(Bezeichner1 Eigenschaft1 Bezeichner2 Eigenschaft2 .....)

Die Propertylisten sind also Listen, deren top-level-Elemente abwechselnd Bezeichner f}r die Eigenschaft und die Eigenschaft selber sind.

DEKALISP stellt die folgenden drei Funktionen zum Arbeiten mit Propertylisten zur Verf}gung:

(PUT atom bez eig) eval, spread

PUT setzt bei dem literalen Atom atom' die Eigenschaft mit dem Namen bez' auf den Wert eig'. Dazu wird jedes zweite top-level-Elemente (beginnend mit dem ersten) von atom' mit bez' }ber die Funktion EQ verglichen. Wenn eine }bereinstimmung gefunden wird, so wird der bisherige Wert der Eigenschaft bez' (d.h. das n{chste top-level-Element der Propertyliste) durch eig' ersetzt. Findet PUT keinen Bezeichner auf der Propertyliste von atom', der gleich bez' ist, so werden bez' als neues erstes und eig' als neues zweites top-level-Element vor die alte Propertyliste geh{ngt. atom' darf nicht NIL sein. Der Wert von PUT ist eig'.

(REMPROP atom bez) eval, spread

REMPROP entfernt von der Propertyliste von atom' alle Eigenschaften mit dem Bezeichner bez'. Es werden sowohl die Bezeichner als auch die Eigenschaften entfernt. Wiederum darf atom' nicht NIL sein. Der Wert von REMPROP ist bez', wenn ein solcher Bezeichner auf der Propertyliste vorkam und sonst NIL.

(GETP atom bez) eval, spread

GETP sucht auf der Propertyliste von atom' nach dem ersten Bezeicher, der gleich (EQ) zu bez' ist und liefert den zugeh|rigen Eigenschaftswert. Wenn GETP keinen entsprechenden Bezeichner findet oder wenn atom' kein literales Atom ist, so liefert es NIL.

Beispiel:

```
(PUT 'PETER 'MUTTER 'MARIA)
--->  MARIA
(PUT 'PETER 'BRUDER '(OTTO HANS HERMANN))
--->  (OTTO HANS HERMANN)
(CDR 'PETER)
--->  (BRUDER (OTTO HANS HERMANN) MUTTER MARIA)
(GETP 'PETER 'MUTTER)
--->  MARIA
```

3. Atome

Atome sind ebenso wie die Zeichenketten die Grundbausteine der Listen. Um zu testen, ob ein Listenelement ein Atom ist, gibt es:

(ATOM x) eval, spread
liefert x', wenn x' ein Atom ist, sonst NIL.

Es gibt zwei Sorten von Atomen, die literalen und die numerischen. Jeder Sorte ist ein getrennter Abschnitt in diesem Kapitel gewidmet.

3.1 Literale Atome

Literale Atome sind diejenigen Atome, die einen Namen haben, der nicht als Zahl interpretiert werden kann. Man kann testen, ob ein LISP-Objekt ein literales Atom ist:

(LITATOM x) eval, spread
liefert x', wenn x' ein literales Atom ist, sonst NIL.

Wie in der Natur sind auch in DEKALISP Atome nicht unspaltbar. Die Elementarteilchen sind Printname, Wert, Propertyliste und Funktionsdefinition.

3.1.1 Printname

Jedes Atom hat einen Namen, der bei der Eingabe und beim Ausdrucken (deshalb Printname oder p-name) verwendet wird. In DEKALISP darf dieser Name eine beliebige Zeichenfolge mit einer Länge bis zu 65535 sein, mit der einzigen Einschränkung, daß er nicht als Zahl interpretiert werden kann. Sonst wäre das Atom nämlich ein numerisches Atom. Damit der Printname nicht als Zahl interpretiert werden kann, darf er nicht mit einer Ziffer, mit - oder mit + anfangen.

Die eine Funktion des Namens ist es, ein Atom bei der Eingabe genau zu bezeichnen. Jedesmal, wenn DEKALISP den Printnamen eines literalen Atoms einliest, ersetzt es diesen durch das entsprechende Atom. Dies ist eindeutig möglich, da es zu jedem Printnamen nur ein Atom gibt. Trifft DEKALISP bei der Eingabe auf einen vorher noch nicht benutzten Printnamen, so erzeugt es ein neues Atom, dem es dann diesen Namen zuordnet.

Bei der Eingabe eines Printnamens gibt es Schwierigkeiten, wenn der Name Zeichen enthalten soll, die eine spezielle Funktion bei der Eingabe haben, wie z.B das Leerzeichen oder die Klammern. Damit diese Zeichen ihre Funktion verlieren und statt dessen normal in den Printnamen eingebaut werden, muß ihnen ein Prozentzeichen vorangestellt sein. Dieses Prozentzeichen % wird dabei selbst nicht Bestandteil des Namens. Auch das läßt sich aller-

dings durch %% erreichen.

Beim Ausdrucken eines Printnamens, der spezielle Zeichen enthält, gibt es zwei Optionen, entweder der Name wird normal mit allen speziellen Zeichen Ausgedruckt, oder aber den speziellen Zeichen wird auch bei der Ausgabe automatisch ein % vorangestellt. Letzteres ist notwendig, wenn das Ausgegebene in der gleichen Form wieder eingelesen werden können soll (z.B bei der Verwendung von Textdateien).

Printnamen können nicht nur bei der Ein- und Ausgabe verwendet werden, es gibt auch Funktionen, die mit Printnamen genauso arbeiten, wie mit gewöhnlichen Zeichenketten. Siehe dazu Kapitel 4.

Das Normale ist es, daß ein neues Atom dadurch erzeugt wird, daß sich in der Eingabe ein neuer Printname befindet. Unter diesem Printnamen ist das Atom in Zukunft immer anzusprechen. Es gibt jedoch auch eine Möglichkeit, Atome zu erzeugen, die zwar einen Printnamen haben, unter diesem aber nicht erreichbar sind.

(GENSYM)

GENSYM erzeugt ein Atom mit einem Printnamen der Form

Annnn

Dabei ist nnnn eine Dezimalzahl. Bei jedem neuen Aufruf von GENSYM ist nnnn eine um eins höhere Zahl, als bei dem vorhergegangenen Aufruf. Auf diese Weise ist sichergestellt, daß GENSYM Atome mit lauter verschiedenen Printnamen erzeugt. Der Printname wird bei der Eingabe nicht erkannt.

Beispiel:

```
(SETQ X (GENSYM))
--->  A0000
(SET X 56)
--->  56
(SETQ A0000 11)
--->  11
(EVAL X)
--->  56
A0000
--->  11
```

Zwei weitere Funktionen erzeugen literale Atome:

(PACK liste) eval, spread

Die Zeichenketten, die sich beim ausdrucken der top-level-Elemente von liste' ergeben würden, werden zu einem Printnamen hintereinandergehängt. PACK hat als Wert das Atom mit diesem Printnamen, entweder ein schon bestehendes oder ein von PACK mit dem Printnamen neu erzeugtes. Dies kann entweder ein numerisches oder ein literales Atom sein, jenachdem, ob der entstandene Printname mit einer Ziffer bzw. einem Vorzeichen beginnt und somit als numerisch identifiziert wird, oder nicht. Bei der Bestimmung der Zeichenketten für die top-level-Elemente von liste' wird PRIN1 verwendet, d.h. es werden keine Anführungsstriche oder Prozentzeichen einge-

baut.

Beispiel:
(PACK '(A B C "DEF"))
---> ABCDEF

Eine ähnliche Funktion hat:

(MKATOM) eval, spread

MKATOM hat wie PACK ein Atom mit berechnetem Printnamen als Wert. Anders als PACK verwendet MKATOM jedoch nicht die Zeichenfolgen der top-level-Elemente, sonder die Zeichenfolge, die sich beim Ausdrucken von x' selbst ergibt. Auch hier werden keine " oder % eingebaut.

Beispiel:
(MKATOM '(A B C "DEF))
---> %(A% B% C% DEF%)

In etwas allgemeinerer Form ist UNPACK das Gegentei von PACK:

(UNPACK x flag) eval, spread

UNPACK hat als Wert eine Liste, die als top-level-Elemente lauter Atome hat, deren Printnamen nur ein Zeichen lang sind. Diese Aneinanderreihung von Atomen entspricht der Zeichenfolge, die entstehen würde, wenn x' ausgedruckt würde. Wenn ein zweites Element flag angegeben wird und flag' nicht gleich NIL ist, so wird die Zeichenfolge verwendet, die auch von PRINZ beim Ausdrucken erzeugt würde, in die also an den entsprechenden Stellen Prozentzeichen und Anführungsstriche eingefügt wurden. Sonst werden diese Zeichen nicht erzeugt.

Beispiel:
(UNPACK 'ABC)
---> (A B C)
(UNPACK '(A B% C "DEF" (G)))
---> (%(A % B % C % D E F % %(G %) %))
(UNPACK '(A B% C "DEF" (G)) T)
---> (%(A % B %% % C % %" D E F %" % %(G %) %))

3.1.2 Wert

Dadurch, daß literale Atome Werte zugewiesen bekommen können, nehmen sie in LISP die Funktion von Variablen an. Über sie läuft die Übergabe von Argumenten an Funktionen und sie können Zwischenergebnisse speichern.

Der Zugriff auf den Wert eines literalen Atoms geschieht im allgemeinen durch Evaluierung des Atoms. Diese ist automatisch, wenn ein literales Atom in Ausdrücken vorkommt, kann jedoch durch QUOTE unterdrückt werden (siehe Kapitel 6).

Wenn ein literales Atom neu erzeugt wird, so besitzt es zunächst noch keinen Wert. Der Versuch, ein solches Atom zu evaluieren führt zu der Fehlermeldung "Unbound Atom".

Die wesentliche Funktion, um einem literalen Atom einen Wert zuzuweisen, ist SET:

(SET atom wert) eval, spread
SET weist dem literalen Atom atom' den Wert wert' zu und gibt wert' zurück.

(SETQ atom wert) noeval, spread
SETQ weist dem literalen Atom atom den Wert wert' zu und gibt wert' zurück. SETQ unterscheidet sich von SET nur dadurch, daß atom nicht evaluiert wird.

(SETQQ atom wert) noeval, spread
SETQQ weist dem literalen Atom atom den Wert wert zu und gibt wert zurück. Bei SETQQ wird also auch wert nicht evaluiert.

Das erste Argument von SET, SETQ und SETQQ muß ein literales Atom sein, sonst wird die Fehlermeldung

Argument not Atom - SET

erzeugt. Wenn versucht wird, NIL einen Wert zuzuweisen, so gibt das die Fehlermeldung

Attempt to SET NIL

NIL hat grundsätzlich sich selbst als Wert.

So überraschend es klingen mag, ein literales Atom hat nicht nur einen Wert, sondern eventuell mehrere. Jedesmal, wenn ein literales Atom als lokale Variable in einer Funktion verwendet wird, wird der zu dem Zeitpunkt des Aufrufes dieser Funktion bestehende Wert auf einen Stapelspeicher gerettet. Dann kann das Atom einen neuen lokalen Wert erhalten, der verloren geht, sobald die Funktion wieder verlassen wird.

Normalerweise kann auf die gesicherten Werte eines aktuell als lokale Variable benutzten Atoms nicht zugegriffen werden. DEKALISP erlaubt es jedoch, auf den globalen Wert zuzugreifen. Der globale Wert ist der Wert, zu dem das Atom immer dann evaluiert, wenn es nicht als lokale Variable verwendet wird. Die beiden Funktionen CAR und PRLACA, mit denen der globale Wert gelesen bzw. gesetzt werden kann, arbeiten normalerweise mit Listen (siehe Kapitel 2.1 und 2.2). Hier wird nur die Wirkung beschrieben, wenn das erste Argument der Funktionen zu einem literalen Atom evaluiert.

(CAR atom) eval, spread
liefert den globalen Wert des literalen Atoms atom'. Wenn

dem Atom noch kein Wert zugewiesen wurde, so ist der Wert von CAR NOBIND.

(RPLACA atom wert) eval, spread
RPLACA ersetzt den globalen Wert des literalen Atoms atom' durch den neuen Wert wert'. atom' darf nicht NIL sein. wert' wird von RPLACA zurückgegeben.

3.2 Numerische Atome

Wie der Name schon sagt, dienen numerische Atome dazu, Zahlen darzustellen. Da dies ihr einziger Zweck ist, können sie weder Propertyliste noch Funktionsdefinition besitzen.

Der Printname eines numerischen Atoms ist die dezimale Darstellung seines Zahlenwertes. Der Zahlenwert ist damit genauso ein festgelegter Bestandteil eines numerischen Atoms, wie der Printname eines literalen Atoms und kann nicht geändert werden. Darüberhinaus werden numerische Atome nicht eindeutig gespeichert, d.h. es können beliebig viele Atome mit dem gleichen Zahlenwert, sprich Printnamen existieren.

Bei der Eingabe erkennt DEKALISP numerische Atome daran, daß sie mit einer Ziffer oder einem Vorzeichen +,- beginnen.

Numerische Atome werden sowohl durch die Eingabefunktionen, als auch durch arithmetische Funktionen erzeugt.

Für den Test, ob ein LISP-Objekt ein numerisches Atom ist, gibt es:

(NUMBERP x) eval, spread
liefert x', wenn x' ein numerisches Atom ist, sonst NIL.

3.2.1 Zahlenbereich

Der einzige numerische Typ, der derzeit implementiert ist, sind ganze Zahlen. Ganze Zahlen werden mit bis zu 56 Bits im Zweierkomplement dargestellt, haben also einen Wertebereich von -2^{55} bis $+2^{55}-1$. Betragsmaessig grosse (lange) Zahlen, die zu ihrer Darstellung mehr als 24 Bit benoetigen, belegen dabei einen LISP-Knoten und werden durch einen Zeiger auf diesen Knoten vertreten. Kleine ganze Zahlen dagegen stehen direkt anstelle des Zeigers, was sowohl Rechengeschwindigkeit als auch Speicherplatz spart.

Fuer eine spaetere Implementation ist auch die Realisierung von Gleitkommazahlen geplant. Reflektiert wird dies zur Zeit bereits dadurch, dass viele der arithmetischen Funktionen doppelt vorhanden sind: Mit einem vorgestellten "I" (z.B. IPLUS) werden als

Argumente nur ganze Zahlen akzeptiert und auch der Wert ist eine ganze Zahl. Die generellen Funktionen (z.B. PLUS) fuehren zwar zur Zeit noch direkt auf diese Integerfunktionen, werden aber spaeter ihre Argumente auf deren Typ hin ueberpruefen und entsprechend eine Funktion fuer Integer- oder Gleitkommaarithmetik aufrufen.

3.2.2 Arithmetische Funktionen

Das folgende sind die allgemeinen Rechenfunktionen. Sie erwarten, daß ihre Argumente zu numerischen Atomen evaluieren und lösen andernfalls eine Fehlermeldung aus.

(PLUS zahl1 ... zahln) — eval, no-spread
Liefert die Summe aller Argumente. (PLUS) ergibt null.

(DIFFERENCE zahl1 zahl2) — eval, spread
Liefert das erste Argument minus dem zweiten Argument.

(TIMES zahl1 ... zahln) — eval, no-spread
Liefert das Produkt aller Argumente. (TIMES) ergibt 1.

(QUOTIENT zahl1 zahl2) — eval, spread
Liefert den Quotienten aus dem ersten und zweiten Argument.

(ABS zahl) — eval, spread
Liefert den Absolutwert des Arguments.

(MINUS zahl) — eval, spread
Liefert das Produkt des Arguments mit -1.

Das folgende sind Integerversionen vom oben beschriebenen Funktionen:

(IZEROP zahl)	eval, spread
(IMINUSP zahl)	eval, spread
(IEQP s-expr1 s-expr2)	eval, spread
(ILESSP zahl1 zahl2)	eval, spread
(IGREATERP zahl1 zahl2)	eval, spread
(IPLUS zahl1 ... zahln)	eval, no-spread
(IDIFFERENCE zahl1 zahl2)	eval, spread
(ITIMES zahl1 ... zahln)	eval, no-spread
(IQUOTIENT zahl1 zahl2)	eval, spread
(IABS zahl)	eval, spread
(IMINUS zahl)	eval, spread

Folgende Funktionen sind zusaetzlich nur fuer Integerarithmetik definiert:

(ADD1 zahl) — eval, spread
Liefert das um eins erhoehte ganzzahlige Argument.

(SUB1 zahl) eval, spread
Liefert das um eins verminderte ganzzahlige Argument.

(IREMAINDER zahl1 zahl2) eval, spread
Liefert den Divisionsrest der ganzzahligen Division des ersten durch das zweite Argument.

3.2.3 Prädikatsfunktionen

Zur Untersuchung von numerischen Objekten stehen folgende Prädikatsfunktionen zur Verfuegung:

(FIXP atom) eval, spread
Liefert T fuer ganze Zahlen, sonst NIL.

(FLOATP atom) eval, spread
Liefert T fuer Gleitkommazahlen, sonst NIL.

(SMALLP atom) eval, spread
Liefert T fuer kurze Zahlen (siehe oben), sonst NIL.

Zum Vergleich von Zahlen gibt es folgende Praedikate:

(ZEROP zahl) eval, spread
Liefert T, wenn das Argument null ist, sonst NIL.

(MINUSP zahl) eval, spread
Liefert T, wenn das Argument negativ ist, sonst NIL.

(EQP s-expr1 s-expr2) eval, spread
Wenn eines der beiden Argumente nicht numerisch ist, fuehrt diese Funktion auf die Funktion EQ, d.h. auf den Vergleich der beiden Zeiger, andernfalls liefert sie T, wenn beide Argumente von gleichem Wert sind, sonst NIL.

(LESSP zahl1 zahl2) eval, spread
Liefert T, wenn das erste Argument echt kleiner als das zweite ist, sonst NIL.

(GREATERP zahl1 zahl2) eval, spread
Liefert T, wenn das erste Argument echt groesser als das zweite ist, sonst NIL.

4. Zeichenketten

Zeichenketten sind keine weitere Form von Atomen, sondern ein spezieller Datentyp. Deshalb haben Sie auch keinen Printnamen, Wert, Propertyliste oder Funktionsdefinition. Sie bestehen lediglich aus einer Reihe von Zeichen.

Die Zeichenketten in DEKALISP, die im folgenden auch Strings genannt werden, bestehen aus bis zu 65535 Zeichen, genauso wie die Printnamen von Atomen. Die Verwandtschaft zwischen Strings und Printnamen ist }berhaupt sehr eng, so da~ die meisten Zeichenkettenfunktionen genausogut mit Printnamen arbeiten k|nnen, wie mit Strings.

Wenn Sie Strings eingeben wollen, so m}ssen Sie diese in Anf}hrungsstriche " setzen. Der erste Anf}hrungsstrich in einer Eingabe kennzeichnet den Anfang eines Strings, der n{chste das Ende, der }bern{chste den Anfang des n{chsten Strings und so weiter. Daran erkennen Sie schon, da~ es nicht ohne weiteres m|glich ist, innerhalb eines Strings einen Anf}hrungsstrich zu verwenden. Hier dient wiederum das Prozentzeichen %, da~ die spezielle Wirkung eines ihm folgenden Zeichens aufhebt. Wenn Sie Strings eingeben, brauchen Sie allerdings nicht, wie bei der Eingabe von Listen, auch Leerzeichen oder Klammern ein Prozentzeichen voranzustellen. Nur " und % haben innerhalb von Strings eine spezielle Bedeutung.

Beim Ausdrucken von Zeichenketten gibt es, wie beim Ausdrucken von Printnamen, zwei M|glichkeiten. Entweder die Zeichenketten werden so ausgedruckt, wie sie eingegeben werden, also eingeschlossen in Anf}hrungsstriche und eventuell mit Prozentzeichen, oder es werden nur die eigentlichen Zeichen des Strings gedruckt. Siehe dazu Kapitel 5.

Ob ein gegebenes LISP-Objekt eine Zeichenkette ist, k|nnen Sie testen mit:

(STRINGP x) eval, spread
liefert T, falls x' eine Zeichenkette ist, sonst NIL.

Denken Sie daran, da~ Zeichenketten zwar keine Listen sind, aber dennoch auch keine Atome.

Zum Vergleich zweier Zeichenketten untereinander gibt es:

(STREQUAL string1 string2) eval, spread
STREQUAL liefert T, falls string1' gleich string2' ist, sonst NIL. Zwei Zeichenketten sind gleich, wenn sie aus der gleichen Folge von Zeichen bestehen, d.h wenn sie beim Ausdrucken das gleiche Druckbild liefern.

Auch mit der Funktion EQUAL k|nnen Strings verglichen werden, da diese ihrerseits die Funktion STEQUAL aufruft, wenn ihre Argumente zu Strins evaluieren.

(MKSTRING x) eval, spread

MKSTRING hat als Wert eine Zeichenkette, die aus der Zeichenfolge besteht, die sich beim Ausdrucken von x' ergeben w}rde. Hierbei wird die Option benutzt, die keine " und % druckt.

Beispiel:
```
(MKSTRING '(A 12 "CD" ((E% F>
---> "(A 12 CD ((E F)))"
```

(MIDSTRING string n m) eval, spread

Der Wert von MIDSTRING ist eine Zeichenkette, die einen Teil der Zeichenkette string' darstellt. Welcher Teil genommen wird, bestimmen n' und m'. n' bestimmt die Position innerhalb von string', die den Anfang des Ergebnisstrings bedeutet, und m' bestimmt die Position des letzten Zeichens. Wenn n' oder m' negativ sind, so wird die Position vom Ende von string' aus bestimmt, sonst vom Anfang aus. Wenn die durch n' bestimmte Position hinter der durch m' bestimmten liegt, so liefert MIDSTRING die leere Zeichenkette "". Wenn eine Angabe f}r m fehlt, oder m' gleich NIL ist, so wird daf}r die letzte Zeichenposition in string' angenommen.

Beispiel:
```
(MIDSTRING "ABCDEFGHIJ" 3 7)
---> "CDEFG"
(MIDSTRING "ABCDEFGHIJ" -3)
---> "HIJ"
(MIDSTRING "ABCDEFGHIJ" -3 -7)
---> ""
```

(CONCAT x1 x2 x3 ..... xn) eval, nospread

Wenn ein xi' keine Zeichenkette ist, so wird es durch CONCAT zuerst in eine solche wie durch PRIN1 umgewandelt. Der Wert von CONCAT ist die Zeichenkette, die durch Hintereinanderh{ngen aller Zeichenketten x1' bis xn' entsteht.

Beispiel:
```
(CONCAT "ABC" DEF (GH I))
---> "ABCDEF%(GH% I%)"
(CONCAT)
---> ""
```

5. Ein/Ausgabe

5.1 Textdateien

Die wichtigsten beiden Textdateien sind der Bildschirm und die Tastatur, eine weitere der Drucker. Daneben k|nnen Sie aber auch Dateien auf Ihrem externen Speichermedium, wahrscheinlich Diskette, vielleicht Harddisk, verwalten. Sie kennen das Konzept von Textdateien wahrscheinlich bereits aus Ihrem DOS-Handbuch.

Dateien haben einen Namen. F}r den Bildschirm und die Tastatur ist der Name T. Hier kann es trotz des identischen Namens zu keinen Verwechselungen kommen, da in die eine Datei nur geschrieben und aus der anderen nur gelesen werden kann. Der Drucker hat den Dateinamen LST, das steht f}r listing-device.

Sie sehen schon, da~ die Namen von Dateien literale Atome sind. W{hrend die Namen f}r Bildschirm, Tastatur und Drucker festgelegt sind, k|nnen Sie die Namen f}r die Textdateien auf Ihrem externen Speichermedium selbst w{hlen.

10. Der Listeneditor

Zum bequemeren Umgang mit Listen, zum Anzeigen, Aendern, Einfuegen und Loeschen von Listenelementen in Funktionsdefinitionen, Propertylisten und Variablenwerten enthaelt das Paket EDIT Funktionen zum Editieren solcher Listen. Das Paket ist als Text vorhanden und kann bei Bedarf erweitert und an die Beduerfnisse des Benutzers angepasst werden. In das LISP-System eingeladen wird das Paket durch den Funktionsaufruf (LOAD 'EDIT).

Dem Benutzer stehen nun primaer die drei Funktionen (ED-FN atom), (ED-PROP atom) und (ED-VAL atom) zur Verfuegung, um die Funktionsdefinition, die Propertyliste oder den Wert eines geeigneten Atoms (das Argument wird bei allen drei Funktionen nicht evaluiert) zu editieren. In jedem Fall muss es sich um eine Liste (evtl. auch die leere Liste NIL) handeln. Bei einem Fehler (z.B. wenn versucht wird, die Funktionsdefinition einer interpreterinternen Funktion zu editieren) wird eine entsprechende Mitteilung ausgegeben.

Alle drei Editorfunktionen benutzen eine gemeinsame Funktion mit dem Namen EDIT. Um dem Benutzer jeweils einen handlichen Ausschnitt aus der zu editierenden Liste zur Verfuegung zu stellen, arbeitet EDIT jeweils mit Teillisten der Gesamtliste, wobei die Gesamtliste als TOP bezeichnet wird, das erste Element der Gesamtliste mit TOP.1, das zweite Element der Gesamtliste mit TOP.2 etc., das erste Element von TOP.1 mit TOP.1.1 usw.

Waere die zu editierende Liste z.B. ((A (B C)) D (E F)), so liessen sich folgende Teillisten benennen:

```
TOP:     ((A (B C)) D (E F))
TOP.1:   (A (B C))
TOP.1.2: (B C)
TOP.3:   (E F)
```

Atomare Listenelemente lassen sich auf diese Weise in EDIT nicht adressieren, sodass z.B. TOP.1.2.1 und TOP.2 unzulaessig sind.

Bei der Arbeit mit EDIT wird innerhalb einer Schleife immer zunaechst die aktuelle Teilliste angezeigt, wobei der besseren Uebersicht wegen nichtatomare Listenelemente durch das Symbol "&" ersetzt werden; dann kann der Benutzer eines der im folgenden beschriebenen Kommandos eingeben, um andere Teillisten auszuwaehlen oder um die aktuelle Teilliste zu veraendern.

Zahlen: Wechsel der aktuellen Teilliste

Die Eingabe einer positiven Zahl n macht das n-te Element der aktuellen Teilliste zur neuen aktuellen Teilliste, sofern es sich um eine Liste handelt. Anderfalls - oder wenn die Zahl n die Laenge der aktuellen Teilliste ueberschreitet - wird eine Fehlermeldung ausgegeben und die alte aktuelle Teilliste bleibt bestehen. Die Eingabe einer negativen Zahl arbeitet genauso, zaehlt die Listenelement aber von rechts statt von links wie bei positiven Eingaben. Die Eingabe einer Null macht die direkt uebergeordnete Teilliste zur neuen aktuellen Teilliste. War bereits TOP die aktuelle Teilliste, so erscheint eine Fehlermeldung und TOP

bleibt aktuelle Teilliste.

+: Neue aktuelle Teilliste rechts der alten

Die Eingabe von + sucht das naechste nicht-atomare Listenelement rechts der alten aktuellen Teilliste, aber innerhalb derselben uebergeordneten Teilliste, und macht es zur neuen aktuellen Teilliste. War z.B. TOP.2.4 aktuelle Teilliste, so kann TOP.2.5 zur neuen aktuellen Teilliste werden, sofern es sich um eine Liste handelt. Kann kein entsprechendes Element gefunden werden, z.B. auch, weil die alte aktuelle Teilliste das letzte Element der ihr direkt uebergeordneten Teilliste ist, so wird eine Fehlermeldung ausgegeben und die alte aktuellen Teilliste beibehalten.

-: Neue aktuelle Teilliste links der alten

Dieses Kommando arbeitet ganz analog zu +, sucht die neue aktuelle Teilliste jedoch links der alten.

G: GO zu einer beliebigen neuen aktuellen Teilliste

Zu diesem Kommando gehoert eine Liste von Zahlen, die die Position der neuen aktuellen Teilliste angibt. Die Wirkung dieses Kommandos ist diegleiche als wenn ausgehend von TOP als aktueller Teilliste die Zahlen der zum GO-Kommando gehoerenden Liste nacheinander als Kommando eingegeben werden. Ist die Liste leer (=NIL), so bleibt TOP aktuelle Teilliste.

?: Anzeigen der aktuellen Teilliste

Nach Fehlermeldungen und nach den Kommandos P (PRINT) und L (LINELENGTH) wird die aktuelle Teilliste nicht angezeigt, da sie sich nicht aendert. Mit dem Kommando ? kann diese Anzeige explizit angefordert werden.

C: CHANGE der aktuellen Teilliste oder eines Elements

Zu diesem Kommando gehoeren zwei Parameter: Der zweite Parameter ist eine S-Expression, die entweder die ganze aktuelle Teilliste ersetzt - wenn der erste Parameter eine 0 ist - oder ein Element dieser Liste. Im letzteren Fall gibt der erste Parameter die Stelle des Elements innerhalb der Liste an (positive Zahlen bewirken Zaehlung von links, negative von rechts). Z.B. wuerde das Kommande C 2 X das zweite Element von links innerhalb der aktuellen Teilliste durch das Atom X ersetzen.

I: INSERT eines Elementes in die aktuelle Teilliste

Auch zu diesem Kommando gehoeren zwei Parameter: Die Nummer des Elementes, nach dem eingefuegt werden soll (positiv oder negativ; Null bewirkt Einfuegen als neues erstes Listenelement) und die S-Expression, die eingefuegt werden soll.

D: DELETE eines Elementes aus der aktuellen Teilliste

Dieses Kommando benoetigt als Parameter die Nummer (positiv oder negativ) des Elementes innerhalb der aktuellen Teilliste,

das entfernt werden soll.

V: Zusammenfassen von Elementen zu einer Unterliste

Dieses Kommando benoetigt als Parameter die Nummern (positiv oder negativ) des ersten und des letzten Elements der aktuellen Teilliste, die zu einer Unterliste zusammengefasst werden sollen.

^: Integrieren einer Unterliste in die aktuelle Teilliste

Dieses Kommando nimmt als Argument die Nummer eines Elementes des aktuellen Teilliste, bei dem es sich um eine Liste handeln sollte, und hebt die Elemente dieser Liste um eine Ebene an, macht sie also zu Elementen der aktuellen Teilliste. Ist an der bezeichneten Stelle keine Liste, sondern ein Atom, so wird dieses geloescht.

L: LINELENGTH festlegen

Sollen Ausdrucke des PRINT-Kommandos im Zusammenhang mit dem CHANGE-Kommando weiterverwendet werden (Benutzung des Bildschirmeditors), so ist es u.U. vorteilhaft, die Zeilenlaenge voruebergehend herabzusetzen, um bequemer kurze Einfuegungen in mehrzeilige Ausdruecke machen zu koennen. L gefolgt von einer Zahl wirkt wie ein Aufruf von LINELENGTH. Das Symbol * als Parameter wird durch den Wert des Atoms ED-MAXLINLEN ersetzt. Dieser Wert ist beim Einladen auf 79 initialisiert.

P: PRINT der aktuellen Teilliste

Die aktuelle Teilliste wird auf dem Bildschirm ausgegeben, ohne dass dabei nicht-atomare Elemente durch das Symbol "%" ersetzt werden. Sofern eine Funktion PP (Pretty Printer) vorhanden ist, wird diese zum Ausdruck verwendet, sonst PRINT.

ACCEPT: Beenden mit Annahme der Aenderungen

Die Funktion EDIT wird verlassen und erhaelt als Wert die editierte Liste mit allen durchgefuehrten Aenderungen. Die aufrufende Funktion bindet diese Liste je nach Fall als Funktionsdefinition, Propertyliste oder Wert an das jeweilige editierte Atom.

ABORT: Beenden ohne Annahme der Aenderungen

Die Funktion EDIT wird verlassen und erhaelt als Wert das Atom EDIT% ABORTED. Die aufrufende Funktion laesst Funktionsdefinition, Propertyliste und Wert des jeweiligen editierten Atoms unangetastet.

11. Ablaufverfolgung

Zum Austesten von LISP-Funktionen ist es oft nuetzlich, eine Moeglichkeit zu haben, bei jedem Aufruf der Funktion die Argumente und beim Ruecksprung den Wert auf den Bildschirm auszugeben. Dies ist die Aufgabe des Funktionspakets TRACE. Die Funktionen liegen als Text vor und koennen bei Bedarf vom Benutzer angepasst werden. Das Einladen ins LISP-System geschieht mit dem Funktionsaufruf (LOAD 'TRACE).

Getraced werden koennen nur in LISP selbst geschriebene, nicht compilierte Funktionen. Fuer eine Funktion FN, auf die dies zutrifft, liefert der Aufruf (EXPRP 'FN) den Wert T und mit (GETD 'FN) ist die Funktionsdefinition als LAMBDA- oder NLAMBDA-Ausdruck verfuegbar.

Das Tracen einer Funktion wird durch einen Funktionsaufruf der Form (SETTRACE fn-name eingang ausgang) veranlasst, wobei alle Argumente evaluiert werden. "fn-name" gibt den Namen der zu tracenden Funktion an und die optionellen Argumente "eingang" und "ausgang" duerfen evaluierbare S-Expressions sein, die vor bzw. nach der Evaluierung des eigentlichen Funktionskoerpers berechnet werden. Dies eroeffnet die Moeglichkeit, jeweils beim Aufruf die an die Parametervariablen gebundenen Werte und beim Ruecksprung den Funktionswert auf den Bildschirm auszugeben. Dazu sind im Trace-Paket die Funktionen (PRINTARGS) und (PRINTRESULT) enthalten. Da diese beiden Atome nicht nur mit der Funktionsdefinition versehen sind, sondern auch noch die Liste mit sich selbst als einzigem Element als Wert haben, ist es nicht noetig, fuer eingang und ausgang '(PRINTARGS) und '(PRINTRESULT) anzugeben, sondern es reichen auch einfach die Atome allein.

(PRINTARGS) druckt die Liste der an die Parametervariablen gebundenen Werte, und zwar unabhangig davon, ob der Bindungsmechanismus eval oder no-eval, spread, half-spread oder no-spread ist, (PRINTRESULT) druckt das Ergebnis der getraceten Funktion.

Sei beispielsweise die Funktionsdefinition

```
(DE FAK (N)
  (COND ((ZEROP N) 1)
        (T (TIMES N (FAK (SUB1 N))))))
```

gegeben. Dann wird diese durch den Auruf (SETTRACE 'FAK PRINTARGS PRINTRESULT) entsprechend modifiziert, sodass bei jeder Ausfuehrung der Definition die entsprechenden Trace-Meldungen auf dem Bildschirm erscheinen. Gleichzeitig wird auf der Propertyliste von FAK unter dem Indikator TRACELEVEL ein Zaehler eingerichtet, der die Rekursionstiefe angibt. Der Aufruf (FAK 3) fuehrt dann z.B. zu folgender Bildschirmausgabe:

```
==> FAK (3)
 ==> FAK (2)
  ==> FAK (1)
   ==> FAK (0)     Die Rekursionstiefe wird durch Einrueckung
   <== FAK 1               deutlich gemacht
  <== FAK 1
 <== FAK 2
<== FAK 6
6                Ausgabe des Funktionswertes durch den Interpreter
```

Die urspruengliche Funktionsdefinition wird durch den Aufruf (CLRTRACE fn-name) wiederhergestellt. Dabei wird ausserdem der Indikator TRACELEVEL mit dem zugehoerigen Wert von der Property-liste des Funktionsnamens geloescht. Funktionswert sowohl von SETTRACE als auch von CLRTRACE ist im Erfolgsfall der Funktions-name, sonst (wenn z.B. unter dem angegebenen Namen keine oder keine akzeptable Funktionsdefinition existiert) NIL.