

# MUMATH Anleitung

in PCNEWS Nr. 13 <http://pcnews.at> gab es kurze Anleitungen zu MUMATH/MUSIMP, die ich hier ebenfalls veröffentlichen darf. Vielen Dank an Franz Fiala!

## Kurz-Anleitung zum Arbeiten mit "muMATH-83"

[mumath.txt](#)

```
Kurz-Anleitung zum Arbeiten mit "muMATH-83" + Übungs- und Demo-
Texte
TGM_96: MUMATH.TXT, TGM_97: MUMATH.ARC
Dr. Fridbert Widder, #326
Auf der PCC-TGM Diskette #1187 befinden sich das Programm
MUSIMP.COM
und die vier "memory image files"
ALGEBRA.SYS
CALCULUS.SYS
MATSOL.SYS
PDS.SYS

Um algebraische Umformungen zu machen bzw., um die vier
Rechentechnik-
Übungen CLES1.ALG bis CLES4.ALG zu bearbeiten, startet man am
besten mit
(der DOS-Eingabe):
MUSIMP ALGEBRA

Nach kurzer Lade-Zeit kommt die System-Meldung und das "?", das
eine
muSIMP-Eingabe erwartet.

Man probiert zuerst gleich einmal den in Abschnitt 6-2
beschriebenen
"interaction cycle" aus: Abschließen der muSIMP-Eingaben mit dem
";"
nicht vergessen - sonst geschieht nichts!
(Verlassen des Programms mit: SYSTEM();)

Oder/und man geht der Reihe nach die "Calculator mode LESSsons"
durch -
mit dem "ReaD Select"-Kommando:
RDS(CLES1, ALG, A);
gelangt man sofort "in" die erste Übung (, wenn sich CLES1.ALG auf
der
Diskette im angegebenen Laufwerk A befindet).
```

Diese "CLESSons" erklären "sich" und den Umgang mit muMATH (und muSIMP)  
 (fast) "von selbst" - trotzdem, oder weil's mir so gut gefallen hat,  
 reproduziere ich in der Folge die (mir wichtig erscheinenden) Text-Teile  
 dieser Übungen (und danach die Demonstrations-Beispiels-Beschreibungen,  
 die man am Ende der diversen "muMATH source files" findet).

(Vorher noch ein Hinweis zu den weiteren bereitgestellten "memory image files": Der Zweck der zwei Dateien CALCULUS und MATSOL.SYS ist offensichtlich - PDS.SYS hilft beim Studieren der Programmier-Lektionen PLES1 bis PLES5 und PLES7 und PLES8.PDS. "SYS"-Dateien werden wesentlich schneller geladen, als "source files" eingelesen werden könnten!)

Bevor nun die (ein wenig gekürzten) Texte zu den Rechen-Modus-Übungen und anschließend die Texte zu den "muMATH"-Demonstrationsbeispielen folgen, füge ich\*) noch ein "Inhalts-Verzeichnis" dieser Zusammenfassung ein -- die Seiten-Numerierung wurde so gewählt, daß man diese Seiten als Ergänzung zu den Seiten 8-1 und 8-2 des Handbuches ansehen kann.

\*) F.

Widder

Graz, 15.

9.1988

INHALT der Zusammenfassung der Übungs- und Demonstrations-Beispiele

"muMATH-83"

Seite Inhalt:

8-3 CLES1: Übungsablauf, Eingeben und Auswerten einfacher algebraischer

Formeln, Zuweisungen (von Werten an Variable)

8-5 CLES2: Kontrollvariable RDS, ECHO, BELL, POINT, PBRCH, Funktion

RADIX, Faktorielle, #PI, #I, #E

8-7 CLES3: Variable (mit bzw. ohne Wertzuweisung - der 'Operator),

Funktionen EXPAND, EXPD, FCTR, DIVOUT, PQUOT, PREM, PGCD,

PARFRAC, NUM, DEN, COEFF, CODIV, BASE, EXPON, CONJ,  
 RATIONALIZE,  
 Speicherplatz-Abfrage RECLAIM();  
 8-9 CLES4: Aktuelle Werte von Variablen - EVAL, ESUB  
 Kontrollvariable (Übersicht - FLAGS());) - NUMNUM, ...  
 DENDEN  
  
 8-12 Ergänzungen, NEWLINE, ....  
  
 8-13 ARITH.MUS: Grundlagen der Arithmetik, Funktionen ABS, MIN,  
 GCD, LCM  
  
 8-14 ALGEBRA.ARI: Funktionen EXPAND etc..., NUM, DEN  
  
 8-15 EQN.ALG: Der Gleichungs-Operator == , Umformen von  
 Gleichungen,  
 SOLVE.EQN: Lösen mit SOLVE(Glg, Var) - vgl. S. 8-21:  
 LINEQN.MAT  
  
 8-16 ODE.SOL: "Ordinary Differential Equations"  
 8-17 ODE.NTH: " --- " - of N-TH order"  
 8-18 ODE.MORE: " --- " - , MORE about"  
  
 8-19 ARRAY.ARI: (Spalten- und Zeilen-)Vektoren, Matrizen  
 8-20 MATRIX.ARR: Matrix-Operationen  
 8-21 LINEQN.MAT: Lineare Gleichungssysteme - Matrix-Inversion  
 ABSVAL.ALG: ABS als Präfix-Operator  
  
 8-22 LOG.ALG: Logarithmus-Funktionen  
  
 8-23 TRG.ALG: Trigonometrische Funktionen - Umkehrfkt.: ATRG.TRG  
  
 8-25 ATRG.TRG und die hyperbolischen Funktionen in der Datei  
 HYPER.ALG  
  
 8-26 DIF.ALG: Differentiation, TAYLOR-Entwicklung  
  
 8-27 INT.DIF: bestimmte und unbestimmte Integrale  
 8-28 INT.MORE.INT: Mehr darüber - " -  
 LIM.DIF: Grenzwertbestimmung  
  
 8-29 SIGMA.DIF: Summationen  
  
 8-30 VEC.ARR: Vektor-"In"- und "Ex"-Produkt  
  
 8-31 VEC.DIF.VEC: Vektor-Differential- und Integral-Operationen,  
 (DIV, CURL, GRAD, ...)

Anm.d.Red.: Den Abdruck der hier nur zusammenfassend erwähnten  
 Übungs- und Demonstrationsbeispiele finden Sie in der Datei  
 MUMATH.DOC.

# PROGRAMMIEREN mit muSIMP

[musimp.txt](#)

```
PROGRAMMIEREN mit muSIMP
TGM_96: MUSIMP.TXT, TGM_97: MUSIMP.ARC
Dr. Fridbert Widder, #326

muSIMP (und ähnliche Programm-Sprachen wie z.B. ST-MATH für den
ATARI-ST)
sind gewissermaßen LISP-"Dialekte". Wesentlich dabei ist, daß
Daten und
Operationen mit "Zeigern" versehen sind, welche die Verknüpfungen
und die
Lage im Speicher bestimmen. Das gestattet sehr platzsparende und
schnelle
Daten-Transfers und Listen-Operationen.
```

## 1.1 STRUKTUR der DATEN

Es gibt drei elementare Grundstrukturen:

NAMEn  
INTEGERS = Zahlen  
NODEs = Knoten

Namen und Integers werden gemeinsam als "ATOM"e bezeichnet - sie  
haben  
selbst folgende Unterstruktur:

NAME: Ein Name besteht aus drei Zeigern - die beiden ersten Zeiger  
erhält  
man durch Anwendung der Funktionen FIRST bzw. REST:

- 1.Zeiger (FIRST-->) weist auf den Wert des Namens; bei der ersten  
Verwendung eines Namens weist der 1.Zeiger auf den Namen  
selbst.
- 2.Zeiger (REST -->) weist auf die "property"-Liste; bei der ersten  
Verwendung wird dem Namen die "leere Eigenschaft" FALSE  
beigegeben.
- 3.Zeiger weist auf Funktions-Definition, falls der Name eine  
Funktion  
definiert - sonst auf FALSE.

Hinter diesen drei Zeigern steht noch die ASCII-Zeichenkette des  
"Namens"

(= des "Bezeichners" -) des Namens; der Bezeichner, unter dem wir uns diese Variable merken, darf bis zu 254 Zeichen enthalten. Gewöhnlich beginnen Namen mit Buchstaben - setzt man die Zeichen zwischen "-Zeichen, dann sind beispielsweise auch "137" und "" vom Typ NAME (" " ist der leere Name).

INTEGER: ist aus verarbeitungs-technischen Gründen analog strukturiert:

- 1. Zeiger --> Wert der Zahl
- 2. Zeiger --> Vorzeichen (als "property") der Zahl: + = FALSE  
 - = TRUE

Daher also Vorzeichentest durch REST(zahl)& -

Eingabe!

- 3. Zeiger weist auf den Speicherbereich, in dem die Zahl binär gespeichert

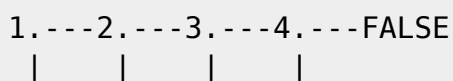
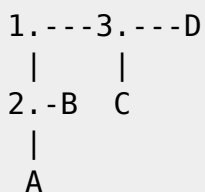
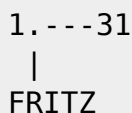
ist. (Kein Zugriff durch den Benutzer)

Mit den "recognicer"-Funktionen NAME(argum) und INTEGER(argum) kann man feststellen, von welchem Typ das ATOM "argum" ist.

NODE: Ein Knoten besteht aus ZWEI ZEIGERN, die auf die Adressen beliebiger Daten-Strukturen zeigen - sowohl auf Atome, als auch wieder auf Knoten, so daß beliebig komplexe Netzwerke entstehen können.

Eine einfache graphische Darstellung: Knoten-Nummer => Zahl.  
 1. Zeiger (FIRST) => |  
 2. Zeiger (REST) => ---

Einfache Beispiele:



A    B    C    D

Die dritte Ketten-Struktur hat einen eigenen Namen - es ist eine LISTE.  
 Listen spielen in muLISP eine zentrale Rolle - daher gibt es neben dem  
 ; Zeichen zum Abschluß eines Rechen-Ausdruckes auch den Listen-Ausgabe  
 & Befehl. - Am besten dazu ein Beispiel probieren:

```

    Falls Sie nicht ohnehin muSIMP geladen haben, dazu also
    MUSIMP PDS eingeben (- lädt den Programm-Demo System-file -
  )
                                und beispielsweise den Unterschied zwischen den
                                Ausgabe-Befehlen ; und & ausprobieren:
? a: -5$
? a;
? a&

```

Viele weitere Übungen zu den Grund-Strukturen sind im muMATH-

Paket PLES1.PDS enthalten - einzulesen mit  
 RDS(PLES1,PDS<,Laufw.>);

In der Folge beziehe ich mich auf diese erste "Programming LESSon 1".

Dort findet man eine spezielle Art der EINGABE von NODE-Strukturen genau erklärt - es handelt sich um die "dotted pair"-Darstellung, die wie folgt eingegeben wird:  
 Beginn mit dem (= Präfix-Operator) '  
 ein Knoten, dessen 1. bzw. 2. Zeiger auf A bzw. B weist, wird dargestellt durch:

```
(A . B)
```

Abschluß (am besten immer) mit dem Listen-Abschluß-Symbol &

Probieren Sie, einfache Knoten- und Listen-Gebilde einzugeben - und zu analysieren! - Dazu dienen die SELECTOR-Funktionen FIRST, REST etc...

Diese können beliebig geschachtelt werden - z.B.  
 FIRST(REST(FIRST(xy)))&

Die wichtigsten Kombinationen liegen aber schon als fertige Funktionen vor:

```

SECOND() = FIRST(REST())           (auch als FREST zu bezeichnen)
THIRD()  = FIRST(REST(REST()))     (analog: FRREST) -

```

```
RREST, FFIRST, RFIRST, FFREST, RFREST, RRREST, FFFIRST, FRFIRST,
RFFIRST,
RRFIRST.
```

Man kann sich (eingeladene) existierende muMATH-Funktionen mit dem `DISPLAY('funktionsname)$` - Befehl ansehen .

Man kann aber auch eigene Funktionen erstellen. Dazu (und auch sonst zum muSIMP-Programmieren) eignet sich gut der

### 1.2 muSIMP "PDS"-EDITOR

(Wie der Name sagt, im PDS.SYS-file integriert - beschrieben in PLES1.PDS)

Er bietet die wesentliche Erleichterung, mit den "Pfeiltasten" auch die Zeilen wechseln zu können, der Insert/Delete-Umschalter wirkt u.s.w.

Um eine Funktion, beispielsweise eine "recognizer"-Funktion namens NODE zu

erzeugen, gibt man (bei geladenem PDS.SYS) ein:

```
EDIT ('NODE) &
```

- der Schirminhalt verschwindet zugunsten des Editor-Displays: in der

ersten Zeile steht bereits `FUNCTION ()`,

und in der 2. das abschließende `ENDFUN$`

- dazwischen schreibt man seine Funktion hinein, im Beispiel etwa:

```
FUNCTION NODE (X),
  NOT ATOM (X),
ENDFUN$
```

Verlassen des PDS-Editors mit [Ctrl-K] !

Darauf wird einem angeboten, mit "redefine" die editierte Funktion zu

übernehmen - oder auszusteigen.

Um die Funktion NODE als "Quell-Datei" zu sichern:

```
FLAGSAVE('NODE) $
```

Um sie auf Diskette (Laufw. A) zu schreiben: `WRITE('NODE, 'MUS, A)`  
&

Bevor ich mit dem Programmieren und den weiteren muMATH -

"PLES"sons  
weitermache, möchte ich einen Übersicht-Abschnitt  
"GRUNDFUNKTIONEN"  
einfügen.

## 2. GRUNDFUNKTIONEN

Bereits bekannt sind u.a. die Zuweisungsfunktion :

und die Terminatoren, die nach der Auswertung des Ausdruckes  
folgende

Ausgabe bewirken:

```
$ keine Ausgabe am Bildschirm
; Ausgabe - womöglich in mathematischer Notation
& Listen-Ausgabe
```

Um die Unterschiede an Beispielen zu demonstrieren, seien folgende  
Namen, Zahlen und Listen definiert:

NAM1: MAX

NAM2: MORITZ

ZAHL1: 100

ZAHL2: -23

ZAHL3: 0

L1: LIST(1,2,3,4,5,6)

L2: LIST(KARIN,ELKE,THEA,HEIDI)

L3: LIST(3,ANDREA,4,5,BEATE)

L4: LIST(NAM1,ZAHL1,ZAHL2,ZAHL3)

L5: LIST(10,11,LIST(12,13),LIST(14),15)

Dabei wurde die LISTen-Erzeugungsfunktion LIST(ob1,ob2,...,obN)  
benutzt,  
die im Abschnitt 2.2 noch einmal beschrieben wird.

### 2.1 Die Terminatoren ; und &

bewirken bei Zahlen und Namen genau die gleiche Ausgabe -  
Unterschiede gibt  
es bei Listen:

L1& --> (1 2 3 4 5 6)

L1; --> 1 (2, 3, 4, 5, 6)

; unterteilt eine Liste in FIRST- und (REST)-Teil, dabei werden  
Elemente

durch Beistriche getrennt. Ein weiterer Unterschied bei  
geschach-  
telten Listen:

```
L5& --> (10 11 (12 13) (14) 15)
L5; --> 10 (11, 12(13), 14(), 15)
```

& zeigt Anfang und Ende jeder Ebene durch ( und ) an.  
 ; zeigt die REST-Listen in ( ) an -- einzelne Atome, die als Liste  
 eingegeben wurden, werden mit der leeren Liste ()  
 angezeigt.

In jedem Falle werden Listen vor der Ausgabe ausgewertet, also:

```
L4; --> MAX (100, -23, 0)
```

## 2.2 Selektions- und Konstruktions-Funktionen

wurden z.T. auch schon im 1.Abschnitt behandelt. So die  
 SELEKTIONS-  
 Funktionen: FIRST, REST, SECOND, ... RRREST

Mit diesen kann man beispielsweise beliebige Elemente aus Listen  
 heraus  
 schälen:

```
RRREST(L2)& --> (HEIDI)
```

Von den KONSTRUKTIONEN-Funktionen kennen wir LIST(ob1, ... obN)  
 durch den  
 Gebrauch in der Praxis.

Um einen Knoten (NODE) zu konstruieren, gibt es auch die Funktion

ADJOIN(ob1,ob2) -- sie liefert den Knoten in der "dotted pair"  
 Darstellung

```
ADJOIN(a,b)& --> (a . b)
ADJOIN(1,FALSE)& --> (1)
ADJOIN(NAM1,L1)& --> (MAX 1 2 3 4 5 6)
```

Während bei Listen die REST-Zelle immer auf einen Knoten oder auf  
 FALSE  
 zeigt, können bei dieser Konstruktion auch Paare von Atomen  
 erzeugt werden.

Man beachte den Unterschied zwischen der Konstruktion eines  
 Knotens mit

```
ADJOIN bzw. durch die direkte Konstruktion mittel "evaluator" ':
                                                    '(a . b) &
```

Vergleiche dazu die "Programming LESSon"2 -- s.S. PLES2-5 !

REVERSE(liste) dreht die Reihenfolge der Listen-Elemente um:

```
REVERSE(L5)& --> (15 (14) (12 13) 11 10)
```

REVERSE(liste1, liste2) dreht liste1 um und vereinigt sie mit liste2:

```
REVERSE(L1,L2)& --> (6 5 4 3 2 1 KARIN ELKE THEA HEIDI)
```

Wird aber statt liste1 nur ein Atom eingegeben, dann wird nur liste2 ausgegeben:

```
REVERSE(7,L1)& --> (1 2 3 4 5 6)
```

aber bei REVERSE(L1,7)& --> (6 5 4 3 2 1 . 7)

OBLIST()& konstruiert eine Liste, in der ALLE NAMEN aufscheinen, die zur Zeit im Speicher sind ! Schon beim Start sind das eine ganze Menge - später selbst eingeführte scheinen am Schluß der Liste auf.

### 2.3 ARITHMETISCHE FUNKTIONEN

Neben den Operatoren +, -, \* gibt es auch Funktionen PLUS(a,b), DIFFERENCE(a,b), TIMES(a,b). Und neben / gibt es die (drei) Divisions-Funktionen:

```
QUOTIENT(5,2)& --> 2
```

```
QUOTIENT(-5,2)& --> -3
```

```
MOD(5,2)& --> 1
```

```
MOD(-5,2)& --> 1
```

und als Kombination der beiden: DIVIDE(var1,var2) --> (quot . mod)

```
DIVIDE(5,2)& --> (2 . 1)
```

### 2.4 LOGISCHE FUNKTIONEN

NOT(obj) gibt dann und nur dann TRUE, wenn obj auf FALSE zeigt.

Bekannt sind die Wirkungen der "Infix"-Operatoren AND und OR.

### 2.5 MODIFIZIERUNGS-FUNKTIONEN

Sie greifen direkt auf Knoten zu und können diese (manchmal gefährlich!) verändern.

REPLACEF(o1, o2) ersetzt die FIRST-Zelle von "o1" durch einen Zeiger auf "o2". Wenn "o1" ein Atom ist, wird die FIRST-Zelle des Atoms ersetzt: gefährlich wäre z.B.:

REPLACEF(1,2) !  
 "o1" = Liste: erstes Element der Liste durch "o2" ersetzt  
 "o1" = Knoten: das linke Element wird durch "o2" ersetzt.

REPLACER(o1, o2) ersetzt die REST-Zelle von "o1" durch Zeiger auf "o2".

CONCATEN(liste, objekt) verbindet eine Liste mit beliebigem Datenobjekt.

## 2.6 PROPERTY-FUNKTIONEN

Aufbau und Verwendung von "Property"-Listen.

Eine Propertyliste wird immer unter einem NAMEN gespeichert - die REST-Zelle des Namens enthält den Zeiger auf den Anfang der P-Liste. In der FIRST-Zelle eines Knotens ist der Indikator, in der REST-Zelle der zum Indikator passende Eigenschaftswert. Den Aufbau besorgt die Funktion

PUT(name, indikator, eigenschaft)

Beispiel: Erstellen einer Telefon-Liste "telefon":

```
PUT(telefon,HANS,12345)&
PUT(telefon,MAX,23456)&
PUT(telefon,BOB,34567)&
```

Zum Ansehen von Propertylisten dient die Funktion REST(name).

Die Antwort im Beispiel:

```
REST(telefon) ---> ((BOB . 34567) (MAX . 23456) (HANS . 12345))
```

Die Speicherbelegung sieht kompliziert aus:

NODE-Nummer	Inhalt der FIRST-Zelle	Inhalt der REST-Zelle
1	Zeiger auf node2	Zeiger auf node3
2	Zeiger auf BOB	Zeiger auf 34567

3	Zeiger auf node4	Zeiger auf node5
4	Zeiger auf MAX	Zeiger auf 23456
5	Zeiger auf node6	Zeiger auf node7
6	Zeiger auf HANS	Zeiger auf 12345

Als Eigenschaften können selbstverständlich auch Namen und sogar Listen eingegeben werden.

Änderungen in schon aufgebauten P-Listen besorgt ebenfalls die Funktion

PUT(name, indikator, property). Sollte also "BOB" eine neue Telefon-Nummer bekommen haben:

```
PUT(telefon, BOB, NNN)
```

Arbeiten mit Propertylisten.

ASSOC(NAM, PLIST) prüft, ob der Indikator "NAM" in der P-Liste "PLIST"

enthalten ist; dabei muß aber die P-Liste selbst (und nicht bloß ihr Name) als zweiter Parameter übergeben werden. Im Beispiel:

```
ASSOC(HANS, REST(telefon)) ---> (HANS . 12345)
ASSOC(HANS, telefon) ---> telefon
ASSOC(MITZI, REST(telefon)) --> FALSE
```

GET(name, indikator) liest den Eigenschaftswert aus der P-Liste, die

unter "name" eingerichtet wurde, der dem eingegebenen Indikator zugeordnet ist; wenn keiner vorhanden: FALSE

```
GET(telefon, HANS) ---> 12345
GET(telefon, Maxi) ---> FALSE
```

## 2.7 "SUBATOMARE" FUNKTIONEN

Wie der Name andeutet, können diese (drei) Funktionen ATOME "spalten".

LENGTH(objekt) - wenn objekt ein NAME ist, ---> Anzahl der Zeichen  
 - wenn object = INTEGER ---> Anzahl der Ziffern  
 (bezogen auf die geltende Basis)

- wenn objekt = LISTE ---> Anzahl der Listenelemente.

EXPLODE(atom) - NAME / INTEGER ---> in einzelne Zeichen / Ziffern aufgespalten (Listenform der Ausgabe)

Umgekehrt kann man die einzelnen Bestandteile wieder zum ATOM verschmelzen mit COMPRESS(liste).

## 2.8 TEST-Funktionen

Dienen zur Abfrage von (Eigenschaften von) Datenobjekten - als Ergebnis erhält man die Antwort TRUE bzw. FALSE.

NAME(obj) prüft, ob es sich bei "obj" um einen NAMEN handelt  
 -  
 NAME(LISTE1) ---> FALSE  
 NAME('LISTE1) ---> TRUE (zu "" vgl. auch 2.10!)

INTEGER(obj) INTEGER-Prüfung

ATOM(obj) ATOM-Prüfung

EMPTY(obj) dient dazu, leere Listen zu erkennen. Nur wenn "obj" gleich  
 FALSE ist, wird als Ergebnis TRUE geliefert.  
 EMPTY() ist der Funktion NOT() identisch.

POSITIVE(obj) ---> TRUE, wenn "obj" eine positive Zahl ist; analog  
 NEGATIVE() für negative Zahlen-Prüfung und  
 ZERO(o) ---> TRUE, wenn "o" Null ist.

EVEN(obj) Abfrage auf gerade Zahlen.

## 2.9 VERGLEICHS-Funktionen

geben (wie Testfunktionen) TRUE oder FALSE als Antwort, wenn der Vergleich stimmt oder nicht.

GREATER(zahl1,zahl2)  
 LESSER(zahl1,zahl2)

Das Ergebnis ist sicher FALSE, wenn eines der Vergleichsobjekte keine Zahl

ist. Dieselben Abfragen realisiert man i.a. einfacher durch die Infix-Operatoren ">" und "<", z.B.: zahl1 > zahl2&

objekt1 EQ objekt2 -- Der Infix-Operator "EQ" prüft auf Gleichheit (von

Namen, Zahlen, Listen). Beispiele:

NAM1 EQ MAX --> TRUE

100 EQ ZAHL1 -> TRUE

LIST(1,2,3) EQ LIST(1,2,3) ---> FALSE, weil bei der Auswertung dieser

stehende - Funktion zwei an verschiedenen Speicherplätzen

werden also verschiedene! - Listen erzeugt werden. Listen

nur durch folgende Zuweisung identisch gemacht:

LISTE6:LISTE1\$ - danach gibt

LISTE1 EQ LISTE6 tatsächlich TRUE.

Der Infix-Operator "=" wirkt etwas "schächer" als "EQ"; die Vergleichs-

Objekte müssen nicht mehr identisch sein - es genügt, wenn sie gleich sind,

um als Antwort TRUE zu bekommen. Das hat nur bei Listen eine Bedeutung:

LIST1 = LIST2 ---> TRUE, wenn LIST1 strukturell gleich LIST2 ist!

MEMBER(objekt,liste) prüft, ob "objekt" in "liste" enthalten ist.

ORDERP(name1,name2) vergleicht, ob "name1" in der OBLIST() vor "name2"

OBLIST() steht. (Jeder neu eingeführte Name wird in die

OBLIST() aufgenommen und zwar in der Reihenfolge der Einführung:

OBLIST().) früher eingeführte stehen weiter rechts in der

## 2.10 ZUWEISUNGS-Funktionen

Der Infix-Operator ":" ist schon bekannt. Links muß immer ein NAME stehen -

rechts kann ein beliebiges Daten-Objekt stehen; bei der Zuweisung mit ":"

wird dann die FIRST-Zelle des NAMEs entsprechend modifiziert.

Z.B.:

```
? NAME2 : SUSI &
@: SUSI
```

Nach & und "return"-Eingabe geschieht folgendes: Wenn "NAME2" noch nicht in der OBLIST() enthalten ist, wird er initialisiert, indem seine FIRST-Zelle auf sich selber zeigt. Das gleiche geschieht mit "SUSI". Dann werden "NAME2" und "SUSI" als Parameter des ":"-Operators interpretiert --> die FIRST-Zelle von "NAME2" wird mit der Adresse von "SUSI" belegt.

Der Prefix-Operator "'" bewirkt, daß NICHT der WERT einer Variable, sondern ihr Name, ihre Bezeichnung übernommen wird:

```
NAME2 ---> SUSI
'NAME2 ---> NAME2
'ZAHL1 ---> ZAHL1
```

Der QUOTE-Operator ' verhindert also die Auswertung des Daten-Objektes, vor dem er steht.

Etwas "trickreicher" wirkt die ASSIGN(name,objekt) Funktion; ein Beispiel:

```
? NAME2:SUSI&
@: SUSI
```

```
? SUSI:BLOND$
```

```
? SUSI&
@: BLOND
```

und jetzt:

```
? ASSIGN(NAME2,'?)&
@: ?
```

```
? NAME2&
@: SUSI
```

```
? SUSI&
@: ?
```

Die ASSIGN-Funktion belegt nicht wie ":" die FIRST-Zelle von "name",

sondern verändert die FIRST-Zelle der FIRST-Zelle von "name"!

PUSH(objekt,liste) legt das "objekt" auf die "liste", die gewissermaßen

als ein Stapel dient:

```
? STAPEL&
```

```
@: STAPEL
```

```
? PUSH(KARIN,STAPEL)$
```

```
.
```

```
.
```

```
? PUSH(HEIDI,STAPEL)$
```

```
? STAPEL&
```

```
@: (HEIDI, . . KARIN . STAPEL)
```

Die "Umkehr"-Funktion POP(liste) nimmt das oberste Element von der "liste":

```
? POP(STAPEL)&
```

```
@: HEIDI
```

```
? STAPEL&
```

```
@: ( . . KARIN . STAPEL)
```

Eine "echte" Liste wird mit PUSH(... ,stap) erzeugt, wenn der Wert von

"stap" am Anfang auf FALSE gesetzt wird.

Leider fand ich keine Zeit mehr, diese meine "Betrachtungen" über das Programmieren mit MuMath / MuSimp (in diesem Jahre) weiterzuführen und zu einem Ende zu bringen; vielleicht klappt's 1989!

F.Widder, Graz,

15.10.88

(F.Widder, Inst.für Theoret.Physik, Univ.platz 5, 8010 GRAZ)

From:

<https://hc-ddr.hucki.net/wiki/> - **Homecomputer DDR**

Permanent link:

[https://hc-ddr.hucki.net/wiki/doku.php/cpm/mumath\\_einstieg?rev=1285161058](https://hc-ddr.hucki.net/wiki/doku.php/cpm/mumath_einstieg?rev=1285161058)

Last update: **2010/09/21 22:00**

