muSIMP/muMATH-79
Reference Manual

Disclaimer

The Soft Warehouse makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, The Soft Warehouse reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of The Soft Warehouse to notify any person or organization of such revision or changes.

muSIMP/muMATH-79 is distributed exclusively by

Microsoft

10800 N.E. Eighth, Suite 819,
Bellevue, WA 98004

Table of Contents

muSIMP-79 REFERENCE MANUAL

muMATH-79 Module Documentation and Source Listings

muSIMP/muMATH-79 Function and Variable Name INDEX

LITERATURE FILES*

File	Version	K-Bytes	Contents
CATALOG.TXT	10/29/79	10	Software catalog and availability
ORDER.TXT	01/01/80	6	Software price list and order form
LICENSE.TXT	01/01/80	11	Software license agreement
FILES.TXT	03/31/80	5	A list of all machine readable files
INTERACT.TXT	10/30/79	5	Interactive use of muSIMP and muMATH
LESSONS.TXT	10/30/79	4	Use of the on-line lesson files

PROGRAM FILES

File	Version	K-Bytes	Contents
MUSIMP79.COM	03/23/80	7	Machine-language muSIMP-79 nucleus
MUSMORE.MUS	11/01/79	11	Completion of muSIMP-79
ARITH.MUS	07/16/79	17	Basic arithmetic package
TRACE.MUS	07/26/79	4	Trace package for debugging programs
ALGEBRA.ARI	07/16/79	12	Basic algebra package
EQN.ALG	03/31/80	2	Equation simplification package
SOLVE.EQN	03/31/80	4	Equation solving package
ARRAY.ARI	03/31/80	5	Array package
MATRIX.ARR	01/14/80	6	Matrix package
LOG.ALG	07/16/79	2	Logarithmic package
TRGPOS.ALG	07/16/79	4	Trigonometric package (Part I)
TRGNEG.ALG	07/16/79	4	Trigonometric package (Part II)
DIF.ALG	07/16/79	3	Symbolic differentiation package
INT.DIF	07/16/79	7	Symbolic integration package (Part I)
INTMORE.INT	07/16/79	7	Symbolic integration package (Part II)

CALCULATOR-MODE LESSONS

File	Version	K-Bytes	Contents
CLES1.ARI	10/30/79	7	Rational arithmetic & assignment
CLES2.ARI	10/30/79	6	Factorials & fractional powers
CLES3.ALG	10/30/79	15	Polynomial expansion & factoring
CLES4.ALG	10/30/79	10	Continued fractions, bases & exponents
CLES5.ALG	10/30/79	12	Complex variables & substitution

PROGRAMMING-MODE LESSONS

File	Version	K-Bytes	Contents
PLES1.TRA	11/01/79	18	Data structure and function definition
PLES2.TRA	11/01/79	9	Data composition and recursion
PLES3.TRA	11/01/79	13	List and set operations
PLES4.TRA	11/01/79	12	Control constructs, loops, and block
PLES5.TRA	11/01/79	19	Property lists and function evaluation

*These files are included only if there is sufficient space on the disk.

FILE NAMING CONVENTION

1. Files of type COM are unprintable directly-executable machine-language program COMmand files.

2. Files of type SYS are unprintable machine-language memory-images which can be LOADed from within muSIMP.

3. Files of type DOC are printable files DOCumenting the usage of a program file having the same first name.

4. Files of type TXT are printable text files containing information implied by the first name.

5. Files having a first name of the form CLESn are interactive Calculator-mode LESsons to be executed from within muSIMP, in the order indicated by the numeric suffix n.

6. Files having a first name of the form PLESn are interactive Programming-mode LESsons to be executed from within muSIMP, in the order indicated by the numeric suffix n.

7. All files listed above of a type other than COM, SYS, DOC, and TXT are muSIMP program source files. The type name denotes the first three letters of the first name of the most immediate prerequisite program file. For example:
   a) The muSIMP file named ALGEBRA.ARI implements algebra, requiring the muSIMP file named ARITH.MUS as a prerequisite.
   b) File ALGEBRA.DOC is the reference documentation for usage of the facilities implemented by file ALGEBRA.ARI.
   c) Files CLES3.ALG and CLES4.ALG are interactive calculator-mode lessons teaching use of the facilities implemented by file ALGEBRA.ARI.

% RDS ()$

General Information

Congratulations on your purchase of the muSIMP/muMATH-79 Symbolic Mathematics System. This package is a revolutionary and sophisticated software system for 8080 and Z80 based microcomputers. Therefore some degree of study and patience is required to properly build a system from muMATH source files, and then save the result as a memory image file. However once built and saved, it is a simple matter to load and interact with the system at any later time as described in file INTERACT.TXT.

Unfortunately your first task is the rather uninteresting one of building the system as explained in the remainder of this file. The immediately following background information is provided to make the process seem less mysterious.

The muSIMP-79tm Programming Language

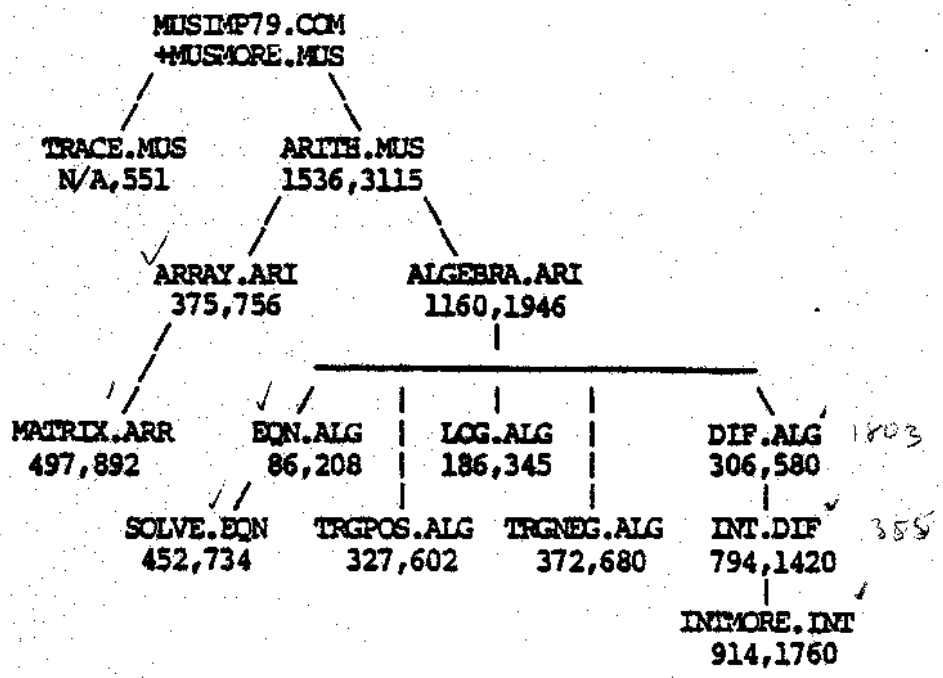
The muSIMP-79 (micro-computer Structured IMPlementation language) system provides a high level programming language suitable for a wide variety of applications. It is implemented using an efficient and versatile interpreter requiring 7K bytes of machine code. The current version of muSIMP also requires a bootstrap file which is loaded immediately after the machine coded portion. The interpreter is distributed as two disk files:

MUSIMP79.COM	an executable COMmand file
MUSMORE.MUS	the muSIMP bootstrap file

The file named FILES.TXT indexes and briefly describes the muSIMP documentation package. The documenation is only distributed in printed form. FILES.TXT also lists the lesson files required to become proficient in the programming language. Since using muMATH in the calculator mode requires no programming knowledge, learning muSIMP can be safely postponed.

The muMATH-79tm Symbolic Math System

The muMATH-79 System provides the facilities to perform a wide variety of symbolic mathematical operations efficiently and accurately on a computer. It is implemented as a set of muSIMP program packages. These source file packages are organized in a very modular fashion in order to accommodate both differing mathematical needs and differing computer memory sizes. More sophisticated mathematical packages require prerequisite files as indicated by the following dependency diagram. Each file requires those above it in the diagram as a prerequisite. (The significance of the numbers will be explained below.)



System Generation Procedure

If you are proficient in the use of the microcomputer's disk operating system (DOS), the following procedure should be sufficient explanation to build and save a complete muMATH system. However if you are a novice or questions arise, additional information can also be found in files BACKUP.TXT and INTERACT.TXT, and in the documentation provided with the disk operating system.

I. Generate a muSIMP/muMATH-79 backup disk.

Using the computer's DOS, transfer a copy of the source files depicted in the above diagram from the Soft Warehouse master diskette to a backup diskette. Since the total disk space required to store the files is approximately 96 K bytes, more than one diskette may be required. Often it is convenient to generate a DOS on the backup disk(s).

II. Build a muSIMP-79 System.

Execute the MUSIMP79 COMmand file by the entering the following DOS command:

MUSIMP79

It should then display a version/copyright message. Jot down the number following the word "SAVE" in the message for later use. The bootstrap file will begin to load automatically. After about 5 minutes load time, the system should respond with the "? " prompt character indicating that muSIMP has been successfully constructed.

III. Build a muMATH-79 System.

Two muSIMP commands are required to build a muMATH system. All commands must be terminated by a semicolon and a carriage return in order to initiate the action.

The source file "name"."type" on disk "drive" is loaded by issuing a command of the form

```
RDS (name, type, drive);
```

Note that the file name and type are separated by a comma rather than a period, and the drive requires only a single letter without the customary colon. The drive is an optional argument which defaults to the drive currently logged in.

Each muMATH source file requires a given amount of computer memory to store its function definitions. The unit of storage in muSIMP is the "node" which is described in file MUSDATA.MUS. The pairs of numbers in the dependency diagram above are the approximate number of nodes required to store the corresponding muMATH package in both condensed and uncondensed form. See the next section for the significance of the smaller condensed numbers.

The number of free (i.e. unemployed) nodes is given by the muSIMP command

```
RECLAIM ();
```

When building a system you must ensure that there is adequate storage for the program plus at least 500 additional nodes to store mathematical expressions. This can be accomplished through the use of RECLAIM and reference to the uncondensed number in the dependency diagram corresponding to the program package. Once this has been verified, execute the appropriate RDS load command. This procedure is illustrated by the following dialogue building a muMATH ALGEBRA system:

```
? RECLAIM ();
@ 6360

? RDS (ARITH, MUS);
@ ARITH

? RECLAIM ();
@ 3245

? RDS (ALGEBRA, ARI);
@ ALGEBRA

? RECLAIM ();
@ 1300
```

IV. Save the muMATH-79 System.

In order to avoid repeating the tedious build routine in the future, a memory image file of the muMATH system should be saved immediately after the build using either method A or B described below. Whichever method is used, it is advisable to use a file name representative of the most sophisticated mathematical capabilities loaded. For example, names such as ALGEBRA, TRIG, CALCULUS, EQUATION, etc.

*** WARNING *** Before attempting a save, ensure that a diskette with sufficient free space to store almost an entire memory image is properly mounted on a disk drive.

A. Return control to the DOS by typing CTRL-C. Then generate a COM type file by issuing the DOS SAVE command. Use the decimal number, N, of 256 byte records recorded in step II above. The file size will be (N/4) K bytes.

Although most desirable from a convenience standpoint (see file INTERACT.TXT), this method may NOT work due to limitations of the DOS. The easily recognizable symptoms are either the inability of the DOS to save that large a file, or erratic behavior when the COM file is subsequently executed. If this happens, you will have to build the system again from step II and use method B in the future.

B. Type the muSIMP command of the form

```
SAVE (name, drive);
```

This will save a file called name.SYS on the given drive. Again the drive is optional, defaulting to the current drive. The SYS file will be approximately (N-28)/4 K bytes, where N is the number recorded in step II.

Generation of COM or SYS files is also useful for checkpointing a lengthy dialogue for purposes including:

1. continuation at a later time,

2. preservation of an environment so that uncertain exploratory computations which might endanger the environment can be safely pursued,

3. preservation of a "program" (meaning an environment) produced interactively rather than using a text editor.

Condensed System Generation

Although muMATH is much more compact than any previous general-purpose symbolic math system, it is still a large set of programs for microcomputers. Fortunately, the use of a "condensing" technique to load programs can economize on memory consumption by a factor of almost two. If the control variable named CONDENSE is assigned the value TRUE, then common subexpressions of function definitions are automatically shared as the source files are read in. The infix muSIMP colon operator is used to make the assignment to CONDENSE as described in the muSIMP lesson and documentation files. Thus, to build a condensed ALGEBRA system, the following commands should be issued in place of those in the example of step II above:

```
CONDENSE: TRUE;
RDS (ARITH, MUS);
RDS (ALGEBRA, ARI);
CONDENSE: FALSE;
```

The exhaustive searching makes condensation too slow for interactive use, so that is why it is advisable to set CONDENSE to FALSE just before the save. The above condensed load requires an hour or so, depending on the processor speed. Since it is possible to type the above commands on one line, you are free to take a long break while the condensation takes place.

Condensation can be regarded as an optional sort of "compilation" stage, and one of the principal reasons for generating COM or SYS files is to preserve this investment of time. However, for simplicity, we suggest not worrying about producing condensed COM or SYS files until it becomes impossible to otherwise fit all of the desired files into memory simultaneously.

Many of the muMATH files contain optional sections identified by conspicuous comments. To save space, appropriately pruned versions of these files can be created using a text editor. For example, a user with only 32 kilobytes may have to delete portions of the arithmetic and algebra packages in order to do algebra comfortably, even with condensation. As with the set of muMATH files, each file is internally organized "bottom up", with the most expendable and highest-level features late in the file. Consequently, the files can be truncated between almost any two commands, without risk of invoking undefined functions or uninitialized control variables.

INITIATING A muMATH COM FILE

Initiating muMATH is easiest if someone has already saved a COMmand file having a memory image including all of the muMATH or other programs which you wish to use at that time. The name of such a file is of course up to whoever creates it, but in general the name assigned is of the highest-level math package loaded. Thus ALGEBRA.COM would be the name of a command file containing ARITH.MUS and ALGEBRA.ARL.

For example when using a CP/M (tm) type operating system with file ALGEBRA.COM on the current drive, one merely enters the following system-level command terminated by a carriage return:

```
ALGEBRA
```

After a minute or so of loading from the diskette, the response should be a message of the form:

```
muSIMP-79  (Version month/day/year)  SAVE: size
Copyright (c) 1979 by The SOFT WAREHOUSE  phone
?
```

where appropriate numbers appear as the entries "month", "day", "year", "size", and "phone". You are now free to enter mathematical expressions as described below.

INITIATING A muMATH SYS FILE

Unfortunately for reasons described in file READIST.TXT, it may be impossible to construct a muMATH COMmand file. However, the following means of initiating muMATH is always possible and quite easy provided someone has saved a SYS-type file containing a memory image of the muMATH packages which are needed at that time. Here too the name of such a file is up to whoever creates it, but in all probability the same naming convention describe above for COM files was used. Therefore ALGEBRA.SYS would have had the same source files loaded in as the file ALGEBRA.COM would have.

For example, if MUSIMP79.COM is on the current drive and ALGEBRA.SYS is on disk drive B, the appropriate operating system load command would be as follows:

```
MUSIMP79  B:ALGEBRA
```

About half a minute after the muSIMP logon message appears, the muMATH system should respond with the "? " prompt characters. Now you can begin your interactive dialogue with muMATH.

THE INTERACTION CYCLE

muSIMP prompts the user with a question mark indicating readiness to accept a command entered from the terminal. The user then types an expression followed by a semicolon and a carriage return. First muSIMP parses the expression and converts it into an internal representation. After printing an "@" to herald the "@nswer", the expression is evaluated, and then a space is printed to indicated the evaluation phase is complete. Finally the result is deparsed and printed in mathematical notation. This interaction cycle is repeated indefinitely until a CTRL-C is typed (i.e. a "C" typed while depressing the CTRL key).

For example, here is a segment of a trivial muSIMP dialogue:

```
? 5;
@ 5

? 2 + 2;
@ 4

? JOHN = MARY;
@ FALSE

? MEMBER (APPLE, '(GRAPE, APPLE, PLUM));
@ TRUE
```

CORRECTING TYPOGRAPHICAL ERRORS

Since muSIMP uses the operating system's console I/O routines, all the line-editing features of that system are inherited by muSIMP. Backspacing is usually accomplished by typing either a CTRL-H, or a RUBout, or a DELete key. Some systems echo the deleted character; whereas, others erase the character from the screen and backspace the cursor. Entire lines can be deleted or flushed by typing a CTRL-U or a CTRL-X. As a note of caution: there is no way to modify a line once a carriage return has been typed. If this happens, the entire expression can be flushed by typing a semicolon.

INTERRUPTING EVALUATION

An evaluation in progress can usually be aborted by typing a CTRL-Z, ESCape, or ALTmode. An options available message will then be displayed. The usual choice is typing another CTRL-Z, ESCape, or ALTmode which allows you to enter expressions as before. The other alternatives are fully explained in file ERRORS.TXT. As a last resort, the computer can of course be RESET and a "cold start" performed to reload the operating system.

```
% RDS () $
```

How to Backup the Master Diskettes Supplied by the Soft Warehouse

1. The following information is provided for those who are not throughly familar with their computer's disk operating system. Since there are many different operating systems and computer configurations, it is necessarily a general guide which should be supplemented by study of the documentation supplied with the disk operating system.

2. Obtain an appropriate number of blank new, high-quality diskettes suitable for your drive.

3. Become thoroughly familiar with the terminal, computer, disk drives, and operating system. Most cases of accidental erasure of the master diskettes or other irrecoverable errors are committed in the first few moments by eager users, inexperienced with the system on which they are installing the new software. In particular, practice initializing a diskette, then generating a disk operating system on it. Use the largest version of the system in terms of the space available to user programs. Finally transfer to the new diskette files from a spare, write-protected diskette.

4. Due to wear and inadequate industry-wide manufacturing standards, there are slight or not-so-slight mechanical and electrical differences between various nominally compatible drives and diskettes. Consequently, we suggest that if you have two or more drives, you first try placing the new diskette in the drive on which it will be used most often. The Soft Warehouse diskette can then be tried on each of the other drives, trading with the new diskette if none of the other drives are successful.

5. For two or more drives, some operating systems provide a convenient COPY DISK command which automates most of the copying protocol otherwise necessary. Alternatively, the PIP (Peripheral Interchange Program) or XFER (File Transfer) commands of most operating systems usually permit copying all files from drive A to drive B by a command such as

```
PIP A:=B:*.*
```

or

```
XFER A:=B:*.*
```

6. Copying diskettes is much more laborious on systems having only one drive. Generally, it involves repetitively reading a portion into main memory from the old disk, switching disks, then writing the portion onto the new disk, then switching back to the old disk. The process generally involves using a resident bootstrap monitor in read-only memory, or using a "DDT"-like "Dynamic Debugging Tool" program. Moreover, the process may pad out the otherwise only partially filled last page or record of a file with arbitrary garbage which is harmless in a command file but annoying in a text file. Thus, text files transferred this way may require some text editing to clean-up.

7. For some operating systems it may be necessary to remove the write protect tape from the old diskette temporarily, even though it is only to be read from. (Perhaps this is true only if the old diskette is in the "principal" drive.) Moreover, it may be awkward to copy a diskette without first removing the write protect tape in order to copy the operating system onto that diskette, especially if there is only one drive.

8. We use costly highest-quality diskettes, and we endeavor to record them on the most precisely adjusted drives available. Consequently, if you cannot read our diskettes after several attempts and after carefully restudying our directions and those provided for the hardware and operating system, then

a) Carefully check whether or not you correctly specified all of the details on the order form and purchased the proper type of blank diskettes.

b) Use an alignment test diskette or have your drive checked professionally.

c) Get help from an experienced professional or friend.

Using The Interactive Lessons Files

This file explains how to use the interactive muSIMP and muMATH lesson files. muMATH and the lessons are designed to serve a broad range of math levels from arithmetic through calculus, and to serve a broad range of programming backgrounds from none to professional programmer. How is this scope possible? Read on:

The sequence of lesson files CLES1, CLES2, etc. explains how to use muMATH as an arithmetic or symbolic calculator, for successively more sophisticated mathematical operations. The sequence of lessons PLES1, PLES2, etc. explains how to write programs in muSIMP, in order to enhance the suite of built-in operations or for any other purpose.

The calculator sequence is ordered according to the most common sequence in which the corresponding math subjects are taught. It is intended that a user proceed in this sequence only as far as their math background, before optionally beginning the programming sequence. Due to slight variations in math curricula sequences, some users may prefer to skip certain calculator lessons in the middle of the sequence as well as at the end.

muMATH has such a rich set of built-in capabilities that many users will be content to postpone study of the programming sequence indefinitely. However, many users evenntually will want to proceed to the programming sequence, perhaps for one or more of the following reasons:

1. to enhance the built-in muMATH capabilities,

2. to understand how the underlying muMATH algorithms work,

3. to learn computer programming,

4. to use muSIMP for some other application.

In order to make the programming sequence most useful to users of all mathematical backgrounds, the sequence begins with muSIMP examples which are non-mathematical, or arithmetic at most. Most general programming techniques and their realization in muSIMP are independent of higher-level math. Thus, only the last lessons in this sequence deal with muMATH specifically, explaining how to extend it, alter it, and even replace it with alternative symbolic math systems.

There are three ways to experience the lessons. For most people, the best way is to execute them interactively, trying out examples at the opportunities provided; the second best way is to read the printed record of a dialogue produced by someone else executing the lessons; and the third best way is to read the files containing the original lessons, which contain only one side of an intended dialogue.

As indicated in file FILES.TXT, the first lesson is file CLES1.ARI. Consequently, to commence the lesson you must first initiate a muMATH system containing at least file ARITH.MUS. How to do this is explained in file INTERACT.TXT. Then, you simply issue the muSIMP command

```
RDS (CLES1, ARI, drive);
```

where "drive" is the name of the drive on which the CLES1.ARI is mounted. The lesson will tell you what to do from then on. If the interactive lesson for any reason becomes hopelessly confusing, you can always cease the lesson and simply read it. Also, it may help if you take the lesson along with a companion, because your possible confusions may be disjoint.

Have fun!

```
% RDS () $
```

%File: CLES1.ARI (c) 10/30/79 The Soft Warehouse %

LINELENGTH (78)$ #ECHO: ECHO$ ECHO: TRUE$

% If this lesson is being displayed too fast, it can be temporarily stopped by typing a CTRL-S (i.e. typing the letter "S" while depressing the CTRL key). Then type it again when you are ready to resume.

If you have not yet read files LESSONS.TXT and INTERACT.TXT, it is advisable to abort this lesson and read those files first. To abort the lesson, enter an ESCape or a CTRL-Z character followed by a CTRL-C.

In muMATH a "comment" is a percent sign followed by any number of other characters terminated by a matching percent sign. Thus, this explanation is a comment which has not yet been terminated. Comments do not cause computation; they are merely used to explain programs and examples to human readers. Here is an example of an actual computation%

```
1/2 + 1/6 ;
```
% Note how muMATH uses exact rational arithmetic, reducing fractions to lowest terms.

In muMATH, arithmetic expressions can be formed in the usual manner, using parentheses together with the operators "+", "-", "*", "/", and "^" respectively for addition, subtraction or negation, multiplication, division, and raising to a power. For example: %

```
(3*4 - 5) ^ 2 ;
```
% On some terminals, "^" looks like an upward-pointing arrow; on others it looks like a shallow upside-down letter V; and some terminals may employ an utterly different looking character which you may have to determine by experimentation.

The reason for using ^ and * is that standard terminals do not provide superscripts or centered dots or special multiplication crosses distinct from the letter X.

To prevent certain ambiguities, multiplication cannot be implied by mere juxtaposition. One of the most frequent mistakes of beginners is to omit asterisks.

Later, in order to give you an opportunity to try some examples, we will "assign" the value FALSE to the variable named RDS. When you are ready to resume the lesson, type the "assignment"

```
RDS: TRUE ;
```

including the semicolon and carriage return. This revises the value of the variable named RDS to the value TRUE. We will explain assignment in more detail later.

Don't forget that you can use local editing to correct mistypings on the current line. For example, on many operating systems, the key marked RUBout or DELete cancels the last character typed on the line,

and typing a CTRL-U cancels the current line. There is no way to modify a line after striking the RETurn key, but an expression can always be flushed by typing a final line containing a "grammatical" or "syntax" error such as "(;".

Now we are going to turn control over to you by setting RDS to FALSE. Try some examples of your own similar to the above. Also we suggest that you make a few intentional errors in order to become familiar with how they are treated. For example, try

5 7; 5+ /7; 5/0; and 0/0;

Have fun!: % RDS: FALSE ;

% The value resulting from the last input expression is automatically saved as the value of a variable named #ANS, which can be used in the next expression. For example: %

3 ;#ANS ^ #ANS ;#ANS ^ #ANS;

% As this example illustrates, muMATH can treat very large numbers exactly and quickly. In fact, muMATH can accomodate numbers up to about 611 digits. To partially appreciate how large this is, compute the distance in feet or in meters to the star Alpha Centauri, which is 4 light years away, then use #ANS to compute the distance in inches or in centimeters without starting all over. (In case you forgot, the speed of light is 186,000 miles/second or 300,000,000 meters/second.) %

RDS: FALSE ;

% Our answers are about 123,883,499,520,000,000 feet or 1,486,601,994,240,000,000 inches or 37,843,200,000,000,000 meters or 3,784,320,000,000,000,000 centimeters. Another dramatic comparison with 10^611 is that there are thought to be about 10^72 electrons in the entire universe. (Whoever counted them must be exhausted!)

Often one performs an intermediate computation or a trivial assignment for which there is no need to display the result. When this is the case, the display of the result can be suppressed by using a dollar sign rather than a semicolon as a terminator. For example, type

RDS: TRUE $

and note the difference from when you previously typed RDS:TRUE ; %

RDS: FALSE $

% It is often convenient to save values longer than #ANS saves them, for use beyond the next input expression. The colon ASSIGNMENT operator provides a means of doing so. The name on the left side of the assignment operator is BOUND or SET to the value of the expression on its right. This value is saved as the value of the name until the name is bound subsequently to some other value. The name can be used as a variable in subsequent expressions, as we have used #ANS, in which case the name contributes its value to the expression. For example: %

RATE: 55 $ TIME: 2 $ DISTANCE: RATE * TIME ;

% Alphabetic characters include the letters A through Z, both upper and lower case, and the character "#". Note that the upper and lower case version of a letter are entirely distinct. Names can be any sequence of alphabetic characters or digits, provided the first

character is alphabetic. Thus X, #9, and ABC3 are valid names. Make an assignment of 3600 to a variable named SECPERHOUR, then use this variable to help compute the number of seconds in 1 day and 1 week: %
RDS: FALSE $
% Congratulations on completing CLES1.ARI. To execute the next lesson, merely enter the muMATH command

```
RDS (CLES2, ARI, drive);
```

where drive is the name of the drive on which that lesson is mounted. Alternatively, it may be advisable to repeat this lesson, perhaps another day, if this lesson was not perfectly clear. The use of any computer program tends to become much clearer the second time.

In order to experience the decisive learning reinforcement afforded by meaningful personal examples that are not arbitrarily contrived, we urge you to bring to subsequent lessons appropriate examples from textbooks, tables, articles, or elsewhere. Also, you are encouraged to experiment further with the techniques learned in this lesson: %

```
ECHO: #ECHO $
RDS () $
```

%File: CLES2.ARI (c) 10/30/79 The Soft Warehouse %

LINELENGTH (78)$ #ECHO: ECHO$ ECHO: TRUE$

% This file is the second of a sequence of interactive lessons about the muMATH-79 system for computer symbolic math. This lesson presumes that the muMATH files through ARITH.MUS have been loaded.

For positive integer N, the "postfix" factorial operator named "!" returns the product of the first N successive integers, and 0! returns 1. For example, 3! yields 6, which is 1*2*3. Use this operator to determine the product of the first 100 integers: %
RDS: FALSE $
% The number base used for input and output is initially ten, but the RADIX function can be used to change it to any base from two through thirty-six. For example, to see what thirty looks like in base two: %

THIRTY: 30 $ RADIX (2) ; THIRTY ;
% As you can see, the radix function returns the previous base, which is, of course, displayed in the new number base. This inforation helps to get back to a previous base. In base two, eight is written as 1000, so to see what thirty looks like in base eight: %

RADIX (1000) ; THIRTY ;
% In base eight, sixteen is written as 20, so to see what thirty looks like in base sixteen: %

RADIX (20) ; THIRTY ;
% As you can see, the letters A, B, ... are used to represent the digits ten, eleven, ... for bases exceeding ten. Now can you guess why we limit the base to thirty six?

In input expressions, integers beginning with a letter as the most significant digit must begin with a leading zero so as not to be interpreted as a name. For example, in base sixteen, ten is the letter-digit A, so to return to base ten: %

RADIX (0A) ;
% Why don't you now see what ninety-nine raised to the ninety-nine power looks like in base two and in base thirty-six, then return to base ten: % RDS: FALSE $
% As you may have discovered, it is easy to become confused and have a hard time returning to base ten. Two is represented as 2 in any base exceeding 1, so a foolproof way to get from any base to any other is to first get to base two, then express the desired new base in base two. For example: %

RADIX (2) ; RADIX (1010) ;
% Now we are guaranteeably in base ten, no matter how badly you got lost.

Now consider irrational arithmetic: Did you know that

$$(5 + 2*6^{(1/2)})^{(1/2)} - 2^{(1/2)} - (3/2)^{(1/2)}$$

can be simplified to 0, provided we make certain reasonable choices of branches for the square roots? In general, simplification of arithmetic expressions containing fractional powers is quite difficult, but muMATH makes a valiant attempt. For example: %

```
4 ^ (1/2) ;  12 ^ (1/2) ;  1000 ^ (1/2) ;
```
% Try simplifying the square roots of increasingly large integers to gain a feel for how the computation time increases with the complexity of the input and answer: % RDS: FALSE $
% An input of the form (m/n)^(p/q) is treated in the usual manner as (m^(1/q))^p / (n^(1/q))^p . For example: %

```
(4/9) ^ (3/2) ;
```
% For geometrically similar people, surface area increases as the 2/3 power of the mass. Veronica wears a 1 square-meter bikini, and she is 50,653 grams, whereas her look-alike mother is 132,651 grams. Use muMATH to determine the area of her mother's similar bikini: % RDS: FALSE $
% 4^(1/2) could simplify to either -2 or +2, but muMATH picks the positive real branch if one exists. Otherwise, muMATH picks the negative real branch if one exists, as illustrated by the example: %

```
(-8) ^ (1/3) ;
```
% What if no real branch exits? Then muMATH uses the unbound variable named #I to represent the IMAGINARY number (-1)^(1/2), and expresses the answer in terms of #I, using the branch having smallest positive argument. For example: %

```
(-4) ^ (1/2) ;
```
% Decent simplification of expressions containing imaginary numbers, as described in lesson CLES4.ALG, requires that file ALGEBRA.ARI be loaded. Meanwhile if you believe in imaginary numbers and you can't contain your curiosity, why don't you experiment with them to see what muMATH knows about them: % RDS: FALSE $
% As with manual computation, picking a branch of a multiply-branched function is hazardous, so answers thereby obtained should be verified by substitution into the original problem or by physical reasoning. For this reason, there is a CONTROL VARIABLE named PBRCH, initially TRUE, which suppresses Picking a BRanCH if FALSE. For example: %

```
PBRCH: FALSE $  4 ^ (1/2);
```
% Users having a conservative temperament might prefer to do most of their computation with PBRCH FALSE.

This brings us to the end of CLES2.ARI. Though arithmetic, some of the features illustrated in this lesson may be foreign to you, because sometimes they are taught during algebra rather than before. Thus, if you have any algebra background whatsoever, we urge you to proceed to lesson CLES3.ALG even if some of CLES2.ARI was intimidating. Naturally, as implied by its type, file CLES3.ALG requires a muMATH system containing files through ALGEBRA.ARI.

If you decide not to proceed to algebra, but want to learn how to program using muSIMP, then proceed to lesson PLES1.ARI. %

```
ECHO: #ECHO$  PBRCH: TRUE$  RDS () $
```

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: FALSE$
NUMNUM: DENNUM: 6$  DENDEN: 2$  NUMDEN: PWREXPD: 0$  PBRCH: TRUE$
X: 'X$  ECHO: TRUE$
```

% This file is the third of a sequence of interactive lessons about muMATH-79. This lesson presumes that the muMATH files through ALGEBRA.ARI have been loaded and that the user has studied the arithmetic lessons CLES1.ARI and CLES2.ARI.

An UNBOUND VARIABLE is one to which no value has been assigned. Mathematicians call such variables INDETERMINATES. You may have already inadvertently discovered that if you use an unbound variable in an expression, muMATH treats the variable as a legitimate algebraic unknown. Moreover, muMATH attempts to simplify expressions containing such unbound variables by collecting similar terms and similar factors, etc. For example: %

```
2*X - X^2/X ;
```

% See if muMATH automatically simplifies the expressions 0+Y, Y+0, 0*Y, Y*0, 1*Y, Y*1, Y^1, 1^Y, and 2*(X+Y) - 2*X. %

```
RDS: FALSE $
```

% Sometimes it is desirable to change a bound variable back to unbound status. This can be done by using the single-quote prefix operator, ', which looks like an apostrophe on many terminals. For example: %

```
EG: X + 5;  EG: 'EG;  EG + 2;
```

% Try assigning the value M*C^2 to E, then change E back to unbound status: % RDS: FALSE $

% You may have noticed that some of the more drastic transformations, such as expanding products or integer powers of sums, are not automatic. The reason is that such transformations are not always advantageous. They may make an expression much larger and less comprehensible. However, they may be necessary in order to permit cancellations which make an expression smaller and more comprehensible. Accordingly, there are a few control variables whose values specify whether or not such transformations are performed. For example, the variable controlling expansion of integer powers of sums is called PWREXPD. This variable is conservatively initialized to zero, so that integer powers of sums are not expanded. For example: %

```
EG: (X+1)^2 - (X^2-2*X-1) ;
```

% Clearly this is an instance where expansion is desirable. When PWREXPD is a positive integer multiple of 2, then positive integer powers of sums are expanded, so let's try it: %

```
PWREXPD: 2 $  EG;
```

% Nothing happened!

The reason is that muMATH does not automatically reevaluate previously evaluated expressions just because we change a control value. Not only would this be rather time consuming, but the ability to form

expressions from other expressions constructed under different control settings provides a valuable flexibility for constructing partially expanded expressions.

On the other hand, it is often desirable to reevaluate expressions under the influence of new control settings, and the built-in EVAL function enables this: %

```
EVAL (EG) ;
%    Now that PWREXPD is 2, see how  (X+Y)^2 - (X-Y)^2 simplifies:  %
RDS: FALSE $
%    In muMATH-79, denominators are represented internally as negative
powers, and negative integer powers of sums are expanded if PWREXPD is a
positive integer multiple of 3.  For example:  %

PWREXPD: 3 $  1 / (X+1)^2 ;
%    What happens if  1 / ((X+1)^2 - X)^2  is evaluated under the influ-
ence of PWREXPD being 3?  For a little surprise, try it.% RDS: FALSE $
%    Even though  (X+1)^2 is WITHIN a negative power, it is itself a
positive power, so how about trying again with PWREXPD being 2*3: %
RDS: FALSE $
%    Now, we would like to suggest a little experiment for you:  The
size limitation on algebraic expressions depends primarily upon the
amount of unemployed memory available for storing names, numbers, and
program or algebraic structure.  Memory for the structural use is
measured in units called NODES, which happen to correspond to 4 bytes in
muSIMP-79 on microcomputers.  Node-space tends to be the limiting
resource for algebraic expressions, and we can always determine the
number of unemployed nodes by typing the command: %

RECLAIM ();
%    Numbers and nodes which are no longer a part of any value that we
can retrieve are automatically recycled intermittently, but the RECLAIM
function forces this "garbage collection" process.  The collection takes
on the order of a second, depending on memory size and processor speed;
and these slight pauses are sometimes noticable in the middle of a
printout or a trivial computation.  On a computer with front panel
lights, the collections are also usually recognizable by the change in
light patterns.
```

Naturally, if we load an extravagant number of muMATH files into a single muMATH dialogue or if we save a number of relatively large expressions as the values of variables, then there will be relatively little unemployed space for our next computation. Not only does this limit the size of the next expression, but computation time also increases dramatically as space becomes scarce, because relatively more time becomes devoted to increasingly frequent collections. The moral of the story is: don't unnecessarily load too many muMATH files or retain numerous expressions as the values of variables.

Now, for the experiment: In order to gain an appreciation for how computation time depends on the size of the input expression, answer, and unemployed storage, try timing each computation in the following sequence, until it appears that your space or patience is nearly exhausted:

```
    EG:(1+X)^2;  RECLAIM();  EG:EG^2;  RECLAIM();  EG:EG^2; ... %
RDS: FALSE $
```

% These polynomials are called "dense", because there are no missing terms less than the maximum degree in each unbound variable. In contrast "sparse" polynomials are missing a large percentage of the possible terms less than the maximum degrees. If you are still in an experimental mood, you may wish to try the following analogous sequence which produces extremely sparse results:

```
    RECLAIM(); (A+B)^2;  RECLAIM();  (A+B+C)^2;  RECLAIM(); ... %
RDS: FALSE $
```

% Distribution of sums over sums is another transformation which can dramatically increase expression size but is sometimes necessary to permit cancellations. For example, this transformation is clearly desirable in the expression: %

```
EG: X^2 - 1 - (X+1)*(X-1) ;
```

% When the control variable named NUMNUM is a positive integer multiple of 2, then integers in NUMerators are distributed over sums in NUMerators. Similarly when the variable is a positive integer multiple of 3, then monomials in numerators are distributed over sums in numerators, whereas when the variable is a positive integer multiple of 5, then sums in numerators are distributed over sums in numerators.

The reason for using the successive primes 2, 3, and 5, is that it provides a convenient way to independently control the three types of distribution using one easily remembered control variable name.

The initial value of NUMNUM is 6, because numeric and monomial distribution are recoverable (as we shall see), because neither distribution dramatically increases expression size, and because a lack of these distributions often prevents annoyingly obvious cancellations. For instance the expression `2*(X+1) - 2*X` will not simplify unless NUMNUM is a positive multiple of 2. Similarly `X+1 - (X+1)` will not simplify to 0, since the expression is represented internally as `X+1 + -1*(X+1)`, which requires the -1 to be distributed over the sum.

Thus, to return to our example, %

```
EG;  NUMNUM: 5 * NUMNUM;  EVAL(EG) ;
```

% To witness the great variety of possible expansions, we set %

```
NUMNUM: 0 $  EG:  4 * X^3 * (1+X) * (1-X);
```

% Now, successively EVAL EG with NUMNUM being 2, 3, 5, 6, 10, 15, and 30: % RDS: FALSE $

% In interpreting these results, it is important to recall that negations are represented internally as a product with the integer coefficient -1, so NUMNUM must be a positive multiple of 2 to distribute negations over sums.

If positive values of NUMNUM cause expansion in numerators, how do we request factoring in numerators?

Negative values of NUMNUM cause factoring of numerators. Moreover, the specific negative values cause factoring of the type which reverses the corresponding expansion. For example: %

```
X: 'X $  Y: 'Y $  NUMNUM: -2 $  EG: 10*X^2*Y + 15*X^3;
NUMNUM: 3*NUMNUM;  EVAL(EG);
%    What about negative multiples of 5?  Sorry folks, that's hard for
computers as well as humans.  However, we are working on it for future
releases.  Meanwhile, try out our semifactoring on the example
     3*X*Y^3/7 - 15*X*Y^2/14 + 9*X^4*Y^2/7  %  RDS: FALSE $
%    As you may have guessed, there is a flag named DENDEN which
controls expansion and factoring among negative powers in a manner
entirely analogous to NUMNUM.  Use it together with NUMNUM to expand the
denominator then semifactor the denominator of the expression
     X^2/((X-Y)*(X+Y) + Y^2 + X^2*Y)  %  RDS: FALSE $
%    You may have wondered why we chose the names NUMNUM and DENDEN.
The reason is that there is another closely related control variable
named DENNUM, which controls the distribution of various kinds of
denominator factors over the terms of corresponding numerator factors:
```

> A positive multiple of 2 causes integer denominator factors to be distributed; a positive multiple of 3 causes monomial factors to be distributed; and a positive multiple of 5 causes sum factors to be distributed. For example: %

```
Y: 'Y $  DENDEN: NUMNUM: 0 $  EG: (5 + 3*X^2)*(Y+1)/(15*X*(4+X));
DENNUM: 2 $  EVAL(EG);
DENNUM: 3*DENNUM;  EVAL(EG);
DENNUM: 5*DENNUM;  EVAL(EG);
%    Positive setting of DENNUM and NUMNUM are particularly useful for
work with truncated series or partial fraction expansions.  For example,
see if you can put the expression (6 + 6*X + 3*X^2 + X^3)/6 into a more
attractive form:  %  RDS: FALSE $
%    What about negative values of DENNUM?
```

A little reflection confirms that forming a common denominator reverses the effect of distributing a denominator. Thus, expressions are put over a common integer denominator when DENNUM is a negative integer multiple of 2, expressions are put over a common monomial denominator when DENNUM is a negative integer multiple of 3, and expressions are put over a common sum denominator when DENNUM is a negative integer multiple of 5. For example: %

```
X: 'X $  DENNUM: DENDEN: 0 $  EG:  1 + X/3 + (1+X)/X + (1-X)/(1+X);
DENNUM: -2 $  EG: EVAL(EG);
DENNUM: 3*DENNUM;  EG: EVAL(EG);
DENNUM: 5*DENNUM;  EG: EVAL(EG);
%    Try fully simplifying the expression  X^4/(X^3+X^2) + 1/(X+1) - 1
by expanding over a common denominator, then factoring:  %  RDS: FALSE $
%    As with NUMNUM and DENDEN, the initial setting of DENNUM is 6,
which causes distribution of numeric and monomial denominator factors
over numerator sums.  This tends to give attractive results for
polynomials or series with rational-number coefficients, but the
relatively costly common-denominator operation may be necessary for
problems involving ratios of polynomials.
```

You have now been exposed to the four most important algebraic control variables in muMATH. Together with EVAL, the various

combinations of settings of these variables give rather fine control over the form of algebraic expressions. muMATH cannot read the user's mind, so it is important for the user to thoroughly master the use of these variables in order to achieve the desired effects.

Here are the most frequently useful combinations of settings for these three variables:

PWREXPD: 0; NUMNUM: DENDEN: DENNUM: 6; These initial values are usually good for general-purpose work, when the user wants to view some results before doing anything drastic or potentially quite time consuming.

PWREXPD: 6; NUMNUM: DENDEN: 30; DENNUM: -30; These settings yield a fully expanded numerator over a fully expanded common denominator. This form gives the maximum chance for combination of similar terms. Moreover, a rational function equivalent to 0 is guaranteed to simplify to 0. However, valuable factoring information may be irrecoverably lost.

PWREXPD: 0; NUMNUM: DENDEN: -6; DENNUM: -30; These settings yield a semifactored numerator over a semi-factored common denominator. This form gives the maximum chance for cancellation of factors between a numerator and denominator. However, the factoring is done incrementally, term by term, so it may be necessary to first expand over a common denominator so that all cancellable terms have an opportunity to cancel before attempting factorization.

PWREXPD: 2; NUMNUM: 30; DENDEN: -6; DENNUM: -30; These settings are a good compromise between the advantages of expansion and factoring. Semi-factoring is done in the denominator where it is usually most important, but there is a maximum opportunity for combination of similar terms in the numerator.

PWREXPD: 6; NUMNUM: DENDEN: DENNUM: 30; These settings are good for series expansions or partial fractions, because each term is fully expanded over its own denominator.

Again, we can't overemphasize the importance of mastering the use of these four control variables. They are your primary tool for imposing your will on the simplification process, and any lack of understanding of their proper use will ultimately lead to frustration. Accordingly, why don't you try the above and various other combinations on examples of your own choosing, until the usage becomes second nature: %
RDS: FALSE $
% Congratulations on completing CLES3.ALG. If the mathematical level was uncomfortably high, proceed to lesson PLES1.ARI. Otherwise proceed to CLES4.ALG. In either event, it is advisable to initiate a fresh muMATH environment, because our experiments have altered control values and made assignments which could interfere with those lessons in nefarious ways. %

ECHO: #ECHOS
RDS () $

%File: CLES4.ALG (c) 10/30/79 The Soft Warehouse %

LINELENGTH (78)$ #ECHO: ECHO$ ECHO: TRUE$

% This is the fourth of a sequence of muMATH calculator-mode lessons.

There are some other algebraic control variables besides PWREXPD, NUMNUM, DENDEN, and DENNUM; and they are occasionally crucial for achieving a desired effect. One of these, named NUMDEN, provides the logical completion of the latter three, by controlling the distribution of factors in numerators over the terms of denominator sums. NUMDEN is initially 0, but integer numerators are distributed over denominator sums when NUMDEN is a positive integer multiple of 2, monomial numerators are distributed over denominator sums when NUMDEN is a positive integer multiple of 3, and numerator sums are distributed over denominator sums when NUMDEN is a positive integer multiple of 5. For example: %

```
NUMNUM: DENDEN: DENNUM: 0 $  NUMDEN: 30 $
X / (X^3 + X + 1) / (Y + 1) ;  EG: (X+Y) / (1+X+Y) / (Y+1) ;
```

% Isn't that intriguing? It yields a sort of "continued-fraction" representation. Now for the reverse direction, which performs a denesting of denominators which is less drastic than a single common denominator: %

```
NUMDEN: -6 $  Z + 1 / (1/X + 1/Y) / (1+Y) ;
```

% See if you can devise examples exhibiting dramatic simplifications arising from either direction for this novel transformation. The fact that it so naturally complements NUMNUM, DENDEN, and DENNUM suggests that it must be useful for something! % RDS: FALSE $

% Another control variable named BASEXP controls distribution of a BASe over terms in an EXPonent which is a sum, or controls the reverse process which is collection of similar factors. As might be expected, integer bases are distributed over exponent sums when BASEXP is a positive integer multiple of 2, monomial bases are distributed over exponent sums when BASEXP is a positive integer multiple of 3, and base sums are distributed over exponent sums when BASEXP is a positive integer multiple of 5. Morever, the corresponding negative values cause collection of similar factors having the corresponding types of bases. BASEXP is initially -30. However, distribution (followed perhaps by collection) is sometimes necessary to let some of the terms in an exponent sum combine with the base. For example: %

```
EG: 2^(2+X) / 4 ;  BASEXP: 2 ;  EVAL (EG) ;
```

% See if you can devise an example which requires evaluating an expression first with sufficiently positive BASEXP, then reevaluating with sufficiently negative BASEXP, or vice-versa: % RDS: FALSE $

% Another control variable named EXPBAS controls the distribution of EXPonents over BASes which are PRODUCTS. Integer exponents are distributed over base products when EXPBAS is a positive integer multiple of 2, monomial exponents are distributed over base products when EXPBAS is a positive integer multiple of 3, and exponent sums are distributed over base products when EXPBAS is a positive integer multiple of 5. Naturally, the corresponding negative multiples request

collection of bases which have similar exponents of the indicated type. The initial value is 30, and here are some examples where distribution permits net simplification: %

```
(X^(1/2) * Y) ^ 2 ;  (X*Y)^2 - X^2*Y^2 ;  (4*X^2*Y) ^ (1/2) ;
```

% However, the user should beware that as with manual computation, distribution of noninteger exponents is not always valid. Consequently, conservative users may prefer to generally operate with EXPBAS being 2. Moreover, distribution of exponents tends to make expressions more bulky when no cancellations occur. For example %

```
(X * Y * Z) ^ (1/2) ;
```

% In fact, there are instances where negative settings of EXPBAS are necessary to acheive a desired result. For example: %

```
EG: 2^X * 3^X  + (1+X)^(1/2) * (1-X)^(1/2) - (1-X^2)^(1/2) ;
EXPBAS: -6 ;  NUMNUM: 30 ;  EVAL (EG) ;
```

% See if you can devise an example which requires evaluating an expression first with sufficiently positive EXPBAS, then reevaluating with sufficiently negative EXPBAS, or vice-versa, in order to simplify acceptably: % RDS: FALSE $

% The variable named PBRCH, already discussed in conjunction with fractional powers of numbers, also controls transformations of the form u^v^w --> u^(v*w). PBRCH is initially TRUE, but when PBRCH is FALSE, the transformation occurs only for integer w. Otherwise the transformation occurs for any w. The user should be aware that in some circumstances the selected branch is an inappropriate one, so that it may sometimes be necessary to set PBRCH to FALSE. See if you can devise such an instance: % RDS: FALSE $

% Now, try the examples 0^X and X^0, to see what happens: %

```
RDS: FALSE $
```

% The reason that 0^X is not automatically simplified to 0 is that 0^X is undefined for nonpositive values of X, so the transformation could lead to invalid results. Of course, sometimes users know that the exponent is positive, or they are willing to assume it is positive and verify the result afterwards. Consequently, there is a control variable named ZEROBAS, initially FALSE, which permits the transformation when nonFALSE.

Why then do we automatically simplify X^0 to 1 even though X could perhaps take on the value 0, giving the undefined form 0^0? Well, we also have a control variable for that, named ZEROEXP of course, but we initialized it to TRUE because:

1. If we are thinking of polynomials in X rather than any one specific value of X, then we are free to regard the polynomial X^0 as being formally equivalent to 1.

2. One cannot do effective simplification of rational functions without this widely accepted transformation.

3. Since 1 is the limit of X^0 as X approaches 0 from either side of the real axis, 1 is a reasonable interpretation even for 0^0.

Nevertheless, there is room for disagreement, and anyone who wishes is free to run with ZEROEXP FALSE. Why don't you try it, using some rational expression examples, in order to see how you feel about this issue? % RDS: FALSE $
% It is easy to forget the current control-variable settings, and it is even easy to forget the existence of certain control-variables, so we have provided a handy-dandy function named FLAGS which returns the empty name "" after printing a display of all the flags and their values: %

```
FLAGS ();
```

% If you ever get frustrated because you can't get an answer close to the desired form, try this command. It may reveal some inappropriate settings or remind you of some alternatives you forgot, or reveal the existence of potentially relevant flags of which you were unaware.

Often a dialogue proceeds best under some control settings which are suitable for the majority of the computations, with an occasional need for an evaluation under different control settings. Each such exception could involve new assignments to several control variables, followed by an evaluation then assignments to restore the variables to their usual values. This process can become tedious, and baffling effects can result from inadvertently forgetting to restore a control variable to its usual value. Consequently, as a convenience, we have provided some functions which, for the most commonly desired sets of "drastic" control values, establishes these values, reevaluates its argument, then allows the control variables to revert to their former values before returning the reevaluated argument.

One of these functions is called EXPAND, because it requests full expansion with fully distributed denominators, bases, and exponents. More specifically, it uses the following settings:

```
PWREXPD: 6; NUMDEN: 0; NUMNUM: DENDEN: DENNUM: BASEXP: EXPBAS: 30;
```

To see its effect, try EXPAND (((1+X)/(1-X))^2); % RDS: FALSE $
% Another one of these convenience functions is called EXPD, and it fully expands over a common denominator. Thus the internal control settings are the same as for EXPAND, except that DENNUM: -30. Try

```
EXPD (1/(X+1) + (X+1)^2);  % RDS: FALSE $
```

% Finally, there is a convenience function named FCTR, and it semi-factors over a common denominator. It evaluates its argument under the following control-variable settings:

```
NUMNUM: DENDEN: -6;  DENNUM: BASEXP: EXPBAS: -30;  PWREXPD: NUMDEN: 0;
```

Since semi-factoring is done termwise, it may be necessary to use EXPD before applying FCTR to an expression, in order to get the desired result. See if you can devise an instance where this is true: %
RDS: FALSE $
% This brings us to the end of lesson CLES4.ALG. The next lesson is CLES5.ALG, but as before, it is advisable to start a fresh muMATH to avoid conflicts with bindings established in the current lesson. %

```
ECHO: #ECHO $
RDS () $
```

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: TRUE$
```

% It is often desired to extract parts of an expression. Particularly frequent is a need to extract the numerator or denominator of an expression. Accordingly, there are built-in SELECTOR functions named NUM and DEN for this purpose: %

```
DENNUM: 0 $  EG: (1+X) / X ;  NUM (EG) ;  DEN (EG) ;
NUM (1 + EG);  DEN (1 + EG);
```

% As the last two examples illustrate, NUM and DEN do not force a common denominator or any other transformation before selection, so the denominator is always 1 when the expression is a sum or when the expression is a product having no negative powers. Try out NUM and DEN on a few examples of your own to gain some experience: % RDS: FALSE $

% The Programming-mode lessons will explain how to completely dismantle an expression to get at any desired part, such as a specific term, coefficient, base, or exponent.

muMATH represents the imaginary number (-1)^(1/2) as #I, and muMATH does appropriate simplification of integer powers of #I. For example: %

```
#I ^ 7 ;  EXPAND ((3 + #I) * (1 + 2*#I)) ;  EXPAND ((X + #I*Y) ^ 3) ;
```

% Try it, you'll like it! % RDS: FALSE $

% The definition of the operator "^" in file ALGEBRA.ARI also implements two higher-level transformations which we mention here only in passing:

muMATH represents the base of the natural logarithms as #E and the ratio of the circumference to the diameter of a circle as #PI. Using these, muMATH performs the simplification

$$\#E \text{ ^ } (n * \#I * \#PI / 2) \longrightarrow \#I\text{^}n,$$

where n is any integer constant, after which the power of #I is reduced appropriately. Also, if a control variable called TRGEXPD is a negative multiple of 7, then complex exponentials are converted to trigonometric equivalents. (The opposite transformation for sines and cosines to complex exponentials for TRGEXPD = 7, is implemented by file TRGPOS.ALG.) If your mathematical background includes these facts, you might wish to experience them here. Otherwise you can safely ignore this digression: % RDS: FALSE $

% You may have wondered whether or not an assignment to a variable, say X, automatically updates the value of a bound variable, say EG, which was previously assigned an expression containing X. Let's see: %

```
X: 5 $  Y: 'Y $  EG: X + Y ;  X: 3 ;  EG;  EVAL (EG) ;
```

% Apparently the answer is "no", at least if X is bound when the assignment to EG is made. This should not be surprising, because after contributing its value to the expression X + Y, all traces of the name X are absent from this expression. However, suppose that we do a similar calculation wherein X is initially unbound: %

```
X: 'X $  EG: X + Y;  X: 3;  EG;
% As when we change control variables, previously evaluated
expressions are not automatically reevaluated when we bind an unbound
varible therein.  However, we can always use EVAL to force such a
reevaluation:  %

EVAL (EG) ;
% Since we did not assign the result to EG, reevaluation of EG after a
different assignment to X still has an effect:  %

X: 7 $  EG: EVAL (EG);
% Since this time we did assign the result to EG, further changes to X
can have no effect on EG regardless of evaluation:  %

X: 9 $  EG: EVAL (EG) ;
% If these examples are not entirely clear, you had better take the
time to experimentally learn the principles by trying some examples of
your own: %  RDS: FALSE $
% It is often desired to reevaluate an expression under the influence
of a temporary local assignment to one of the variables therein without
disturbing either the existing value of the variable or else its unbound
status.  The built-in EVSUB function provides a convenient method of
accomplishing this effect.  EVSUB returns a reevaluated copy of its
first argument, wherein every instance of its second argument is
replaced by its third argument.  For example:  %

NUMNUM: 6 $  M: 'M $  C: 'C $  V: 'V $  EG: M*C^2 + M*V^2/2 $
EVSUB (EG, M, 5);  EVSUB (EG, M, M1+M2);  M;
% Play around with EVSUB for awhile until you are absolutely sure that
you understand the difference between substitution and assignment:  %
RDS: FALSE $
% You may have discovered that EVSUB also permits substitution for
arbitrary subexpressions as its second argument.  For example:  %

M: 'M $  C: 'C $  E: 'E $  EVSUB (M*C^2 + 7, M*C^2, E);
% To keep the algebra package small, we have not endowed EVSUB with
any sophistication about finding algebraically IMPLICIT instances of its
second argument in its first.  See if you can find examples where EVSUB
does not do a substitution that you would like it to do:  %  RDS: FALSE $
% Here is an example where a desired substitution doesn't fully occur:%

NUMNUM: 6 $  C: 'C $  S: 'S $  EVSUB (1 - 2*S^2 + S^4,  S^2,  1 - C^2);
% The reason we did not get the desired simplification to C^4 is that
if the second argument is a power, it matches only the same power in the
first argument.  We can usually circumvent such problems by instead
using an equivalent substitution wherein the second argument is a name
rather than a power.  For example:  %

PWREXPD: 2 $  EVSUB (1 - 2*S^2 + S^4,  S,  (1-C^2) ^ (1/2));
% Here is a somewhat different example wherein a desired substitution
does not occur:  %

EVSUB (2*C*S, C*S, C2);
% The reason is that if the second argument is a product, it matches
only the same COMPLETE product in the first argument.  Again, the remedy
```

is to use an equivalent substitution wherein the second argument is a name. For example: $

```
EVSUB (2*C*S, C, C2/S);
```

$ Here is a final example for which a desired substitution does not occur: $

```
EVSUB (C^2 + S^2 - 1 + C + S,  C^2 + S^2,  1);
```

$ Similarly to products, if the second argument is a sum, it matches only the same COMPLETE sum in the first argument. As before, we could circumvent the difficulty by making an equivalent substitution of (1-C^2) ^ (1/2) for S, or (1-S^2) ^ (1/2) for C, but that would leave an ugly square root in the answer. If our goal is to delete the subexpression C^2 + S^2 - 1, then we can use to our advantage the fact that powers must match exactly for a substitution to take place: $

```
EVSUB (C^2 + S^2 - 1 + C + S,  C^2,  1 - S^2) ;
```

$ See now if you can use such techniques to get your examples to work:
$ RDS: FALSE $
$ This brings us to the end of the calculator-mode lessons. There are, of course, higher-level math packages in muMATH, but the fact is that from a usage standpoint, we have already covered the hardest part, which is understanding evaluation, substitution, and the ramifications of the various algebraic control variables. You will find that if you know the relevant math, use of the higher-level packages is quite straightforward, easily learned from studying the corresponding DOC files.

We suggest that before commencing the Programming-mode lessons, you explore calculator-mode usage of the higher-level packages as far as your math background permits. Math curriculum sequences differ, but probably most users will be most comfortable trying the higher-level packaes in the approximate order EQN, SOLVE, ARRAY, MATRIX, LOG, TRGNEG, TRGPOS, DIF, INT and INTMORE. Since space becomes increasingly scarce as higher-level packages are loaded, you may have to reread file READIST.TXT to learn how to CONDENSE and SAVE if you haven't already.

Now for some parting advice about getting the most out of computer symbolic math:

First, storage and time consumption tends to grow dramatically with the number of variables in the input expressions, even if the ultimate result is fortuitously compact. For example, the number of terms in the expanded form of

$$(X1 + X2 + \ldots + XM) \hat{} N$$

grows outrageously with M and N. Consequently, it is important to make every effort to avoid needlessly introducing extra variables for generality's sake. Mathematical and physical problems are often stated using more variables than are strictly necessary, so it is also important to exploit every opportunity to reduce the number of variables from the original problem. Here are some general techniques for doing this:

1. If members of a set of variables can be made to occur only together as instances of a certain subexpression, consider replacing the subexpression with a single variable. For example:

   a) If K, X, and X0 can be made to occur only as instances of the subexpression K*(X-X0), then consider replacing this subexpression with a variable named perhaps KDX.

   b) Similarly, perhaps a combination such as M*C^2 could be replaced with E, or RHO*V^2/L could be replaced with RE.

   These are respectively instances of absorbing an offset together with a proportionality coefficient, renaming a physically-meaningful subexpression, and grouping quantities into dimensionless quantities. Most engineering and science libraries have books describing a more systematic technique called DIMENSIONAL ANALYSIS, and an article in the Journal of Computational Physics (June 1977) explains how computer algebra can automate the process.

2. Even when a variable cannot be eliminated, the complexity of expressions may be reduced if the variable can be made to occur only as instances of a subexpression. For example:

   a) If only even powers of a variable X occur, consider replacing X^2 with a variable named perhaps XSQ.

   b) If X only occurs as instances of 2^X, 2^(2*X), 2^(3*X),..., consider replacing 2^X with a variable named perhaps TWOTOX, yielding mere integer powers of that variable.

Some other advice is to avoid fractional powers and denominators as much as possible. They don't simplify well, they consume a lot of space, and they tend to be hard to decipher when printed one-dimensionally. Often a change in variable can eliminate a fractional power or a denominator.

Sometimes, even when a problem cannot be solved in its full generality, solving a few special cases enables one to infer a general solution which can perhaps then be verified by substitution or by induction. Alternatively, perhaps the original problem can be simplified by neglecting some lower-order contributions, in order to get an analytic solution which will at least convey some qualitative information about the solution to the original problem.

Sometimes only part of a problem or perhaps even none can be solved symbolically, and the rest must be solved numerically. If so, the attempt at an analytic solution at least allows one to proceed with an approximate numerical solution having more confidence that a concise analytical solution has not been overlooked. %

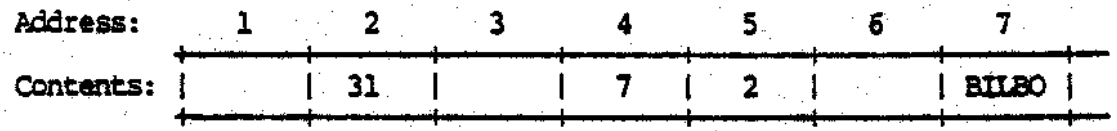
ECHO: #ECHO $ RDS () $

```
%File:  PLES1.TRA  (c)          11/01/79          The Soft Warehouse %

LINELENGTH (78) $  #ECHO: ECHO $   ECHO: FALSE $
#CONDENSE: CONDENSE $   CONDENSE: FALSE $   RFFIRST: 'RFFIRST $

MOVD (PRINT, #PRINT) $
FUNCTION PRINT (EX1),
  WHEN ATOM (EX1), #PRINT (EX1) EXIT,
  #PRINT (LPAR),  PRINT (FIRST (EX1)),  #PRINT (" . "),
  PRINT (REST (EX1)),  #PRINT (RPAR),  EX1,
ENDFUN $

MOVD (PRINTLINE, #PRINTLINE) $
FUNCTION PRINTLINE (EX1),
  PRINT (EX1), NEWLINE (), EX1,
ENDFUN $

ECHO: TRUE $

%  This is the first of a sequence of interactive lessons about muSIMP
programming.  It presumes that you have read files READ1ST.TXT and
LESSONS.TXT, and executed at least one of the calculator mode lessons.
It also presumes that you have loaded packages through TRACE.MUS.

    muSIMP supplies a number of built-in functions and operators.  The
calculator-mode lessons introduced a few of these, such as RDS, RECLAIM,
+, *, etc.  These progamming-mode lessons introduce more built-in
functions and operators, but more important, the lessons reveal how to
supplement the built-in functions and operators with definitions of your
own, thus extending muSIMP itself.

    It is important to realize that, until the last programming-mode
lessons, we will not deal with muMATH.  Instead we deal first with
muSIMP, the language in which muMATH is written.  The illustrative
examples for these first few lessons are utterly different from muMATH,
because we want to suggest a few of the many other applications for
which muSIMP is especially well suited, and because we want these
lessons to be comprehensible regardless of math training level.

    Data is what programs operate upon.  The most primitive UNSTRUCTURED
muSIMP data are integers and names, collectively called ATOMS to suggest
their indivisibility by ordinary means.  Some programs must distinguish
these two types of atoms, so there are two corresponding RECOGNIZER
functions: %

INTEGER (X76#) ;
NAME (X76#) ;
EG: -3271 $
INTEGER (EG) ;
NAME (EG) ;
%  Do you suppose that "137", " ", "", and "X + 3", with the quotation
marks included, are integers, names, or invalid?  Find out for yourself%
RDS: FALSE $
%  Data can be stored in the computer's memory.  The  location at which
```

a data item is stored is called its ADDRESS. An address is analogous to a street address on the outside of a mailbox. The data stored there is analogous to mail inside the mailbox. As with a row of mailboxes, the contents of computer memory can change over time.

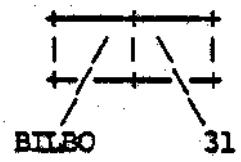
There are useful programs which deal only with unstructured data, but the most interesting applications involve aggregates of primitive data elements. One way to make an aggregate of 2 data elements is to use a structural data element called a NODE, which stores the addresses of the 2 data elements. Thus, a node is "data" consisting of addresses of 2 other data items.

For example, suppose that we wish to represent the aggregate consisting of the name BILBO and his age 31. We could store the name BILBO beginning at location 7, the number 31 beginning at location 2, and the node beginning at location 4. Then, begining at location 4, there would be stored the addresses 7 and 2, as illustrated in the following diagram:



Is that clear?

The specific placement of data within memory is managed automatically, so all we are concerned about is the specific name and number values and the connectivity of the aggregates. Thus, for our purposes it is best to suppress the irrelevant distracting detail associated with the specific addresses. The following diagram is one helpful way to portray only what we are concerned about:



This imagery suggests the word "pointers" for the addresses stored in nodes.

If you have seen one bisected box you have seen them all, so to reduce the clutter and thus emphasize the essential features, we henceforth represent such nodes by a mere vertex in our diagrams, giving schematics such as

Although most muSIMP programs use such aggregates internally, many muSIMP programs are designed to read and print data in whatever specialized notation is most suitable for the application. For example, muMATH uses operator and functional notation. Suppose however that we

want to specify such aggregates directly in input and output. How can we do it? If we have a nice graphics terminal, then then we conveniently could use diagrams such as the above. Most of us do not have nice graphics terminals, so we must use another external representation. For this purpose muSIMP uses a representation consisting of the first data item, followed by the second data item, separated by a dot and spaces, all enclosed in a pair of matching parentheses. For example:

```
(BILBO . 31)
```

We call this representation of a node a DOTTED PAIR. However, this is a diferent use of parentheses and periods from how they are otherwise used in muSIMP input, so we must preceed the dotted pair by the single-quote prefix operator to indicate to the parser that we are using dotted-pair notation rather than the usual operator or functional notation:

```
'(BILBO . 31)
```

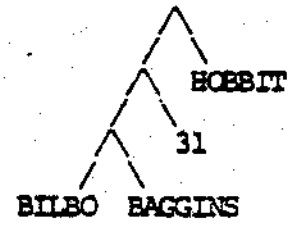
Moreover, we must use an ampersand rather than a semicolon as the expression-terminator in order to inform the driver to print the expression as a dotted pair rather than attempt to print it using operator and functional notation. We say "attempt" because not all dotted pairs are appropriate for operator or functional printing, as we will explain in the last lessons. Here then is an example of dotted-pair input and printing: %

```
'(78 . TROMBONES) &
% Try a few of your own, and note what happens when you forget the
single-quote or use a semicolon rather than an ampersand: %
RDS: FALSE $
% What about when we want to represent an aggregate of more than two
atomic data elements? For example, what if we want to include BILBO's
last name, BAGGINS? Well, we can let one of the pointers of a node
point to another node, whose first pointer points to BILBO and whose
other pointer points to BAGGINS. For example:
```

We can input this as a dotted pair nested within a dotted pair: %

```
'((BILBO . BAGGINS) . 31) &
% Note that we only quote the outermost dotted pair.
```

Now suppose that we want to also include BILBO's species, structured as follows:



How would you input that?
Remember, your input must be terminated by an ampersand.
% RDS: FALSE $
% We would input it as: %

EG: '(((BILBO . BAGGINS) . 31) . HOBBIT) &

% An alternative structure for this information is the one corresponding to the input
'((BILBO . BAGGINS) . (31 . HOBBIT)).
On a piece of scratch paper, sketch the corresponding diagram, then hold it close to my face so I can check it.

% RDS: FALSE $
% My eyes must be getting bad, I couldn't see it. Oh well...

Since either element of a dotted pair can be a dotted pair, they can be used to represent arbitrary "binary tree structures". Moreover, although perhaps unprintable using pure dotted-pair notation, linked networks of binary nodes can be used to represent any data-structure whatsoever.

In order to do anything interesting with data aggregates, a program must be able to extract their parts. Accordingly, there are a pair of SELECTOR functions namd FIRST and REST which respectively return the left and right pointers in a node. For example: %

```
REST (EG) &
FIRST (EG) &
FIRST (FIRST (EG)) &
REST (FIRST (EG)) &
```

% See if you can extract BILBO and BAGGINS from EG, using nested compositions of FIRST and/or REST: % RDS: FALSE $
% Our answers are: %

```
FIRST (FIRST (FIRST (EG))) &
REST (FIRST (FIRST (EG))) &
```

% Deeply nested function invocations become difficult to type and read, so let's define our first muSIMP function named FFFIRST, so that FFFIRST (EG) could be used as shorthand for the first of the above two examples and for any analogous example thereafter: %

```
FUNCTION FFFIRST (U),
  FIRST (FIRST (FIRST (U)))
ENDFUN &
```

% If you are not using a hard-copy terminal, jot down this function definition and all subsequent ones for reference later in the lesson.

Despite the word ENDFUN, the fun has just begun: Now that FFFIRST is defined, we can apply it at any subsequent time during the dialogue. For example: %

```
FFFIRST (EG) &
FFFIRST ('(((BIG . MAC) . CATSUP) . (FRENCH . FRIES))) &
```

% Using the definition of FFFIRST as a model, define a function named RFFIRST which extracts the REST of the FIRST of the FIRST of its argument, then test RFFIRST on EG: % RDS: FALSE $

% Our solution is: %

```
FUNCTION RFFIRST (FOO),
  REST (FIRST (FIRST (FOO))),
ENDFUN &
RFFIRST (EG) &
```

% The name FOO in the definition is called a PARAMETER, whereas EG where the function is applied is an example of an ARGUMENT. We can use any name for a parameter — even a name which has been bound to a value or even the same name as an argument. The name is merely used as a "dummy variable" to help indicate what to do to an argument when the function is subsequently applied. A function definition is like a recipe. It is filed away, without actually being EXECUTED until applied to actual arguments.

As another simple example, since an atom is defined as being either a name or an integer, it is convenient to have a recognizer function for atoms, so that we do not have to test separately for names and atoms when we do not care which type of atom is involved. We could define this recognizer as follows:

```
FUNCTION ATOM (U),
  NAME (U)  OR  NUMBER (U)
ENDFUN &
```

Actually, ATOM is already built-into muSIMP, but the example provides a good opportunity to introduce the built-in infix OR operator, which returns FALSE if both of its operands are FALSE, returning TRUE otherwise. Try out ATOM on the examples -5, X and EG % RDS: FALSE $

% Analogous to OR, there is a built-in infix AND operator which returns FALSE if either operand is FALSE, returning TRUE otherwise. There is also a built-in prefix NOT operator which returns TRUE if its operand is FALSE, returning FALSE otherwise. Knowing this, see if you can define a recognizer named NODE, which returns TRUE if its argument is a node, returning FALSE otherwise: % RDS: FALSE $

% In programming there is rarely, if ever, one unique solution, but ours is: %

```
FUNCTION NODE (U),
  NOT ATOM (U)
ENDFUN &
NODE (EG) &
NODE (5) &
```

% So much for trivial exercises. Now let's write a function which

counts the number of atoms in its argument. We will count each instance of each atom, even if some atoms occur more than once.

At first this may seem like a formidable task, because a tree can be arbitrarily branched. How can we anticipate ahead of time all of these possibilities. Well, let's procrastinate by disposing of the most trivial cases even though we can't yet see the whole solution: If the argument is an atom, then there is exactly 1 atom in it.

So much for trivial cases. We haven't yet solved the whole problem, but it builds our self-confidence to make progress, so that is a good psychological reason for first disposing of the easy cases. Also, with the easy cases out of the way, we can turn our full intellectual powers on the harder cases, unfettered by any distractions to trivial loose ends.

We are left with the case where we know we have a node. Perhaps we could somehow subdivide the problem into smaller cases?

Let's see ... Nodes have a FIRST part and a REST part, so perhaps that provides the natural subdivision. Hmmm ...

If we knew the number of atoms for the left part and the number for the right part, clearly the number for the whole aggregate is merely their sum. But how can we find out the number of atoms in these parts? Why not RECURSIVELY use the very function we are defining to determine these two contributions!

It may sound like cheating to refer to the function we are defining from with the definition itself, but remembering that the definition is not actually APPLIED until sometime after its definition is complete, perhaps it will work. We are working in a highly interactive environment, so the quickest way to resolve questions about muSIMP is to try it and see! Here then is a formal muSIMP function definition corresponding the the above informal English "algorithm": %

```
FUNCTION #ATOMS (U),
  WHEN ATOM (U), 1  EXIT,
  #ATOMS (FIRST(U))  +  #ATOMS (REST(U))
ENDFUN &
```

% Here we introduce 2 new concepts: The BODY of a function definition can consist of a sequence of one or more expressions separated by commas. A CONDITIONAL-EXIT is an expression consisting of a sequence of one or more expressions nested between the matching pair of words WHEN and EXIT. When a function definition is APPLIED, the expressions in its body are evaluated sequentially, until perhaps a conditional exit causes an exit from the procedure or until the delimiter named ENDFUN is reached. For a conditional exit, the first expression after the word WHEN is evaluated. If the value is FALSE, then evaluation proceeds to the point immediately following the matching delimiter named EXIT. Otherwise, evaluation proceeds sequentially through the remaining expressions in the conditional exit, if any, exactly as if the body of the conditional exit replaced that of the function. The value of a conditional exit is that of the last expression evaluated therein, and the value returned by a function is that of the last expression

evaluated therein when the function is applied.

Thus, #ATOMS immediately returns the value 1 whenever the argument is an atom, and otherwise the function breaks the problems into two parts which are necessarily smaller, hence closer to being atoms. Let's test it, starting with trivial cases first: %

```
#ATOMS (FOO) &
#ATOMS (5) &
EG &
#ATOMS (EG) &
```

% It looks promising, but it is still perhaps mysterious how muSIMP and #ATOMS keep track of all of these recursive function invocations. Since the trace package is supposedly loaded, to trace the execution of #ATOMS, we merely issue the command: %

```
TRACE (#ATOMS) &
```

% Now every time #ATOMS is entered, it prints its name and argument values, whereas every time it is exited, it prints its name followed by an equal sign, followed by the returned value. Moreover, the trace is indented in a manner which allows corresponding entries and exits to be visually associated. Watch: %

```
#ATOMS (FOO) &
EG &
#ATOMS (EG) &
```

% Try a few examples of your own, until these new ideas begin to gel:
% RDS: FALSE $

```
UNTRACE (#ATOMS) &
#ATOMS (FOO) &
```

% Here is a function which counts only the number of integers in its argument: %

```
FUNCTION #INTEGERS (U),
  WHEN INTEGER (U), 1 EXIT,
  WHEN NAME (U), 0 EXIT,
  #INTEGERS (FIRST(U)) + #INTEGERS (REST(U))
ENDFUN $
EG &
#INTEGERS (EG) ;
```

% Now, using it as a model, try writing a function named #NAMES, which returns the number of names in its argument. If your first syntactically accepted attempt fails any test, try using TRACE to reveal the reason why: % RDS: FALSE $
% Our solution is ...

On second thought, we won't give you our solution. Consequently, if you were lazy and didn't try, you had better try now, because the examples will get steadily harder now. % RDS: FALSE $
% The HEIGHT of an atom is 1, and the HEIGHT of a node is 1 more than the maximum of the two heights of its FIRST and REST parts. Accordingly, let's first write a function named MAX, which returns the maximum of its two integer arguments. There is a built-in infix integer comparator named ">", so here is a hint:

```
FUNCTION MAX (INT1, INT2),
  WHEN INT1 > INT2,  ...  EXIT,
  ...
ENDFUN $
```

Enter such a definition, with appropriate substitutions for the missing portions, then test your function to make sure it works correctly: %

```
RDS: FALSE $
```

% Now, with the help of our friend MAX, see if you can write a function named HEIGHT, which returns the height of its argument: %

```
RDS: FALSE $
```

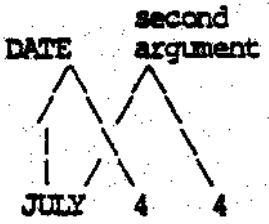
% Our solution is: %

```
FUNCTION HEIGHT (U),
  WHEN ATOM (U), 1  EXIT,
  1 + MAX (HEIGHT(FIRST(U)), HEIGHT(REST(U)))
ENDFUN $
```

% This brings us to the end of the first programming-mode lessons. It may be a good idea to review this lesson before proceeding to lesson PLES2.TRA. %

```
ECHO: #ECHO $
MOVD (#PRINT, PRINT) $
MOVD (#PRINTLINE, PRINTLINE) $
CONDENSE: #CONDENSE $
RDS () $
```

```
% File:  PLES2.TRA  (c)          11/01/79          The Soft Warehouse %

LINELENGTH (78) $  #ECHO: ECHO $  ECHO: FALSE $
#CONDENSE: CONDENSE $   CONDENSE: FALSE $

MOVD (PRINT, #PRINT) $
FUNCTION PRINT (EX1),
  WHEN ATOM (EX1),  #PRINT (EX1)  EXIT,
  #PRINT (LPAR),  PRINT (FIRST(EX1)),  #PRINT (" . "),
  PRINT (REST(EX1)),  #PRINT (RPAR),  EX1,
ENDFUN $

MOVD (PRINTLINE, #PRINTLINE) $
FUNCTION PRINTLINE (EX1),
  PRINT (EX1),  NEWLINE (),  EX1,
ENDFUN $

ECHO: TRUE $
%   This is the second of a sequence of muSIMP programming lessons.

    EQ is a primitive muSIMP Comparator function which returns TRUE if
its two arguments are the same address or equal integers, returning
FALSE otherwise:  %

FIVE: 5 $  EQ (5, FIVE) ;
%   Names are stored uniquely, so two occurences of a name must involve
the same address:  %

ACTOR: 'BOGART ;  EQ (ACTOR, 'BOGART) ;
%   Here is an example of two different references to the same physical
node:  %

DATE: '(JULY . 4) &  FOO:  DATE $  EQ (FOO, DATE) ;
%  However, watch this:  %

EQ (DATE, '(JULY . 4)) ;
%   What happened?  The two aggregates are DUPLICATES, but since they
were independently formed they do not start with the same node.  In
fact, only the name JULY is shared among them, as shown below:
```



Clearly it is desirable to have a more comprehensive equality comparator which also returns TRUE for aggregates which are duplicates in the sense of printing similarly. Let's write such a function, called DUP. Following the general advice given in PLES1, let's first dispose of the trivial cases:

If either argument is an atom, then they are duplicates if and only if they are EQ.

Otherwise, they are both nodes, which is the nontrivial case. Now, let's employ our "divide-and-conquer" strategem, using FIRST and REST as the partitioning. Two nodes refer to duplicate aggregates if and only if the FIRST parts are duplicates and the REST parts are duplicates. Moreover, that can be tested with our beloved recursion, using DUP itself!

See if you can write a corresponding function named DUP: %
RDS: FALSE $
% There are many possible variants, but here is one of the most compact: %

```
FUNCTION DUP (U, V),
  WHEN ATOM (U), EQ (U, V) EXIT,
  WHEN ATOM (V), FALSE EXIT,
  WHEN DUP (FIRST(U), FIRST(V)), DUP (REST(U), REST(V)) EXIT,
ENDFUN $
```

% An interesting challenge for your spare time is to see how many different but reasonable ways this function can be written.

Actually, there already is a built-in infix operator named "=", which is equivalent to DUP: %

```
DATE: '(JULY . 4) $
DATE = '(JULY . 4) ;
```

% Do you feel DUPed to learn that an exercise duplicated an existing facility?

It is crucial to understand exactly what the existing facilities do, and the best way to learn that is to understand how they work by creating them independently.

Here is a good exercise: See if you can write a comparator function named SAMESHAPE, which returns TRUE if its two arguments are similar in the sense of having nodes and atoms at similar places. For example,

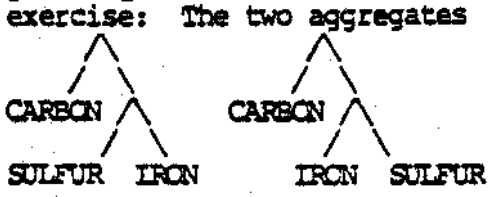
SAMESHAPE ('((KINGS . ROOK) . 5), '((QUEENS . 3) . PAWN))

is TRUE: % RDS: FALSE $
% This is one of those instances where we will not give the answer.

Now, using the infix operator named "=", see if you can write a function named CONTAINS which returns TRUE if its first argument is a duplicate of its second argument or contains a duplicate of its second argument. For example,

((JULY . 4) . (1931 . FRIDAY))

contains (1931 . FRIDAY). It is at least as hard as DUP, so take your time and don't give up easily. % RDS: FALSE $
% Here is a harder exercise: The two aggregates



are ISOMERS because they are either the same atom or at every level either the left branches are isomers and the right branches are isomers, or the left branch of one is an isomer of the right branch of the other and vice-versa. Write a corresponding comparator function named ISOMERS. (It's similar to DUP, with a twist.) % RDS: FALSE $
% Our answer is: %

```
FUNCTION ISOMERS (U, V),
  WHEN ATOM (U), EQ (U, V) EXIT,
  WHEN ATOM (V), FALSE EXIT,
  ISOMERS (FIRST(U), FIRST(V)) AND ISOMERS (REST(U), REST(V))
    OR ISOMERS (FIRST(U), REST(V)) AND ISOMERS (REST(U), FIRST(V))
ENDFUN $
```

% Because of all the combinations which might have to be checked, the execution time for this function can grow quite quickly with depth. Try tracing a few examples of moderate depth: % RDS: FALSE $
% So far our functions have merely dismantled or analyzed aggregates given to them as arguments. None of our examples have constructed new aggregates. The dot of course results in aggregates, but this occurs as the dot is read. Moreover, since the single quote necessarily preceeding an outermost dotted pair prevents evaluation, bound variables in a dotted pair contribute merely their names rather than their values. For example: %

```
EG: 7 $  '(EG . 3) &
```

% What we want is a function which evaluates its two arguments in the usual way, then returns a node whose two pointers point to those values. There is such a function, named ADJOIN: %

```
ADJOIN (EG, 3) &
```

% A dotted pair within a function definition is a static entity, frozen at the time the function is defined. In contrast, a reference to ADJOIN within a function definition is dynamic. The node creation is done afresh, with the current values of its arguments every time that part of the function is applied. As an example of the use of ADJOIN, let's write a function named SKELETON, which constructs a new tree which is structurally similar to its argument but has the name of length zero, "", wherever its argument has an atom. Thus, when printed, the new aggregate will display the skeletal structure of the aggregate without visually-discernable atoms. For example,
SKELETON ('((HALLOWEEN . GHOSTS) . WITCHES)) & will yield ((.) .)

OK, let's recite the litany: What comes first?

TRIVIAL CASES.

So, if the argument is an atom we return what?

"".

Otherwise we have a node, which is the most general case. However, nodes have a FIRST and a REST, so can we somehow recurse, using SKELETON on these parts, then combine them?

Yes, as follows: %

```
FUNCTION SKELETON (U),
  WHEN ATOM (U), "" EXIT,
  ADJOIN (SKELETON (FIRST(U)), SKELETON (REST(U)))
ENDFUN $
SKELETON ('((MOO . GOO) . (GUY . PAN))) &
```

% Easy. Yes?

Now it is your turn. Write a function named TREEREV, which produces a copy of its argument in which every left and right branch are interchanged at every level. For example,

```
          TREEREV ('((MOO . GOO) . (GUY . (PAN . CAKE)))) &
```

should yield

```
                (((CAKE . PAN) . GUY) . (GOO . MOO))
```

% RDS: FALSE $

% If you didn't get the following solution, you may groan when you see how easy it is: %

```
FUNCTION TREEREV (U),
  WHEN ATOM (U), U EXIT,
  ADJOIN (TREEREV (REST(U)), TREEREV (FIRST(U)))
ENDFUN &
TREEREV ('(("Isn't" . that) . easy)) &
```

% Here is a somewhat harder exercise: Write a function named SUBST, which returns a copy of its first argument wherein every instance of its second argument is replaced by its third argument. For example, if

```
PHRASE:
  '(((THIS . (GOSH . DARN)) . CAR) . (IS . ((GOSH . DARN) . BAD))) $
```

then SUBST (PHRASE, '(GOSH . DARN), '(expletive . deleted)) yields

```
(((THIS . (expletive . deleted)) . CAR)
   . (IS . ((expletive . deleted) . BAD)))  %          RDS: FALSE $
```

% That's all folks.

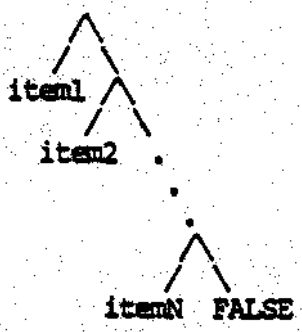
The next lesson deals with a special form of tree called a list. Many people find lists more to their liking, and perhaps you will too.%

```
ECHO: FALSE$

MOVD (#PRINT, PRINT) $  MOVD (#PRINTLINE, PRINTLINE) $
CONDENSE: #CONDENSE $  ECHO: #ECHO $  RDS () $
```

```
LINELENGTH (78) $
#ECHO: ECHO $  #CONDENSE: CONDENSE $  CONDENSE: FALSE $  ECHO: TRUE $
```

% This is the third of a sequence of interactive lessons about muSIMP programming.

Often, it is most natural to represent a data aggregate as a sequence or LIST of items rather than as a general binary tree. For example, such a sequence is quite natural for the elements of a vector or of a set. We can represent such a sequence in terms of nodes by having all of the FIRST cells point to the data elements, using the REST cells to link the sequence together. The last linkage node can have a REST which is FALSE to indicate that there are no further linkage nodes:



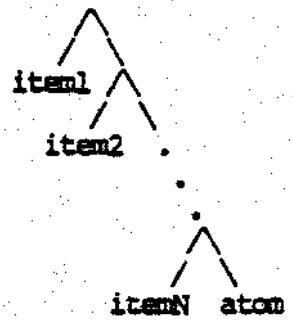
Viewed at a 45 degree rotation, this diagram is analogous to a clothes line with the successive data elements suspended from it, thus more clearly suggesting a sequence. The simple regularity of the structure permits correspondingly simple function definitions for processing such structures. Moreover, the linear structure suggests an external printed representation which is far more readable than dotted pairs. In response to an ampersand terminator, muSIMP prints the above aggregate in the more natural LIST notation:

```
(item1, item2, ..., itemN)
```

rather than the equivalent dot notation

```
(item1 . (item2 .  ... (itemN . FALSE) ... ))
```

Conversely, the reader accepts list notation as an alternative input form to dot notation. Naturally, any of the items in a list can themselves be either lists or more general dotted pairs. The printer uses list notation as much as possible. Thus, a structure of the form



where "atom" is not the atom FALSE, is printed in a mixed notation as

(item1, item2, ... (itemN . atom))

Similarly, the reader can appropriately read such mixed notation.

Thus the last item in a list is implicitly dotted with FALSE, and a blank between two items is equivalent to " . (", together with a matching ")" adjacent to the next right parenthesis. You may wonder why you never noticed such printing conventions during lessons PLES1 and PLES2. The reason is that we purposely redefined the printer for those lessons so that it did not use the list-abbreviation convention.

It is important to fully understand the connection between dotted pairs and lists, so take 5 minutes or so to type in some lists, nested lists, nested dotted pairs, and mixtures, noting carefully how they print. % RDS: FALSE $
% Did your examples include: %

```
'() &
```

% Is that surprising?

Since FALSE is used to signal the end of the list, FALSE and the empty list must be equivalent.

Clearly the trivial terminal case in processing lists will involve an equality test against FALSE. Since this test is so common, there is a corresponding built-in recognizer defined as follows:

```
FUNCTION EMPTY (LIS),
  EQ (LIS, '())
ENDFUN;
```

Using EMPTY, see if you can define a function named #ITEMS, which returns the number of (top-level) items in its list argument. For example, #ITEMS ('(FROG, (FRUIT . BAT), NEWT)) should yield 3. Here is an incomplete solution. All you have to do is enter it with the portions marked "..." appropriately filled.

```
FUNCTION #ITEMS (LIS),
  WHEN EMPTY (LIS), ...  EXIT,
  1 + #ITEMS ( ... )
ENDFUN $                    % RDS: FALSE $
```

% Actually, there is already a built-in function called LENGTH, which returns the length of a list. It is somewhat more general in that it returns the number of characters necessary for printing when given an atom.

Note that with lists it is typical to recur only on the REST of the list, whereas with general binary trees it is typical to recur on both the FIRST and the REST.

So far, the examples and exercises have been relatively isolated ones. Now we will focus on writing a collection of functions which together provide a significant applications package:

A list provides a natural representation for a set. For example, (MANGO, (CHOCOLATE . FUDGE), (ALFALFA, SPROUTS)) can represent a set of three foods. Using this representation, let's write functions which test set membership and form unions, intersections, etc.

First, write a function named ISIN, which returns TRUE if its first argument is in the list which is its second argument, returning FALSE otherwise: % RDS: FALSE $
% Our solution is: %

```
FUNCTION ISIN (U, LIS),
  WHEN EMPTY (LIS), FALSE  EXIT,
  WHEN U = FIRST (LIS), EXIT,
  ISIN (U, REST(LIS))
ENDFUN $
ISIN ('FROG, '(SALAMANDER  NEWT  TOAD)) ;
```

% Actually, there is already a built-in version of ISIN called MEMBER.

A set contains no duplicates, so we really should have a recognizer function named ISSET, which returns TRUE if its list argument contains no duplicates, returning FALSE otherwise. Try to write such a function:
% RDS: FALSE $
% Here is a hint, in case you gave up:

```
FUNCTION SET (LIS),
  WHEN ... EXIT,
  WHEN MEMBER (FIRST(LIS), ... ), FALSE  EXIT,
  SET ( ... )
ENDFUN;                          %       RDS: FALSE $
```

% In case it isn't clear by now, a rule of this game is that you are free (and encouraged) to use any functions we have already discussed, whether they are built-in, previous examples, or previous exercises. That is one reason it is adviseable for you to actually do the exercises.

Now write a function named SUBSET, which returns TRUE if the set which is its first argument is a subset of that which is its second argument. (Remember that every set is a subset of itself and the empty set is a subset of every set.) % RDS: FALSE $
% Here is a hint, in case you gave up or had a less compact solution:

```
FUNCTION SUBSET (SET1, SET2),
  WHEN ... EXIT,
  WHEN MEMBER (FIRST(SET1), ...), SUBSET( ...) EXIT
ENDFUN;                          %      RDS: FALSE $
%   Two sets are equal if and only if they contain the same elements.
However, the elements need not occur in the same order.  Write a
corresponding comparator function named EQSET:  %       RDS: FALSE $
%   Ah yes, a hint perhaps?:

FUNCTION EQSET (SET1, SET2),
   ...
ENDFUN;          %       RDS: FALSE $
%   Do you think that's not much of a hint?
```

Well, the body of the function really can be written with one modest line, so try harder: % RDS: FALSE $

```
%   Remember the rules of the game:  You are encouraged to use any
function discussed previously:  %

FUNCTION EQSET (SET1, SET2),
  SUBSET (SET1, SET2)  AND  SUBSET (SET2, SET1)
ENDFUN;
%   Our examples so far have merely analyzed sets.  We can use ADJOIN to
construct lists, just as we used ADJOIN to construct binary trees.  As
an example of this, write a function named MAKESET, which returns a copy
of its list argument, except without duplicates if there are any:
%   RDS:  FALSE $
%   If you need a hint, here is one, but it is all you will get:

FUNCTION MAKESET (LIS)
  WHEN ...,  '()  EXIT,
  WHEN MEMBER ( ... ), ...  EXIT,
  ADJOIN ( ... )
ENDFUN;          %       RDS: FALSE $
%   Let's see if your solution works correctly:  %

MAKESET ('(FROG, FROG, FROG)) &
%   If there is a duplicate in the answer, then back to the computer
terminal:  %  RDS: FALSE $
%    (It helps to think of nasty test cases BEFORE you start
programming).
```

Now for the crowning glory of our set package: The UNION of two sets is defined as the set of all elements which are in either (perhaps both) sets. Give it a try: % RDS: FALSE $

```
%   A hint perhaps?  Well, the function body can be written in 3 lines,
each of which begins just like the corresponding line in our hint for
MAKESET.  %  RDS: FALSE $
%   Here is our solution: %

FUNCTION UNION (SET1, SET2),
  WHEN EMPTY (SET1),  SET2  EXIT,
  WHEN MEMBER (FIRST(SET1), SET2),  UNION (REST(SET1), SET2)  EXIT,
  ADJOIN (FIRST(SET1), UNION (REST(SET1), SET2))
ENDFUN $
```

% The intersection of two sets is the set of all elements which are in both sets. Using our definition of UNION as inspiration, write a corresponding function for the intersection: % RDS: FALSE $
% So far, our set algebra package has been developed in a so-called BOTTOM-UP maner, with the most primitive functions defined first, and with the more sophisticated functions defined in terms of them. The opposite approach is TOP-DOWN, where we define the most comprehensive functions in terms of more primitive ones, then we define those more primitive ones in terms of still more primitive ones, until no undefined functions remain.

As an example of the top-down attitude, let's write a SYMMETRIC DIFFERENCE function for our set-algebra package. The symmetric difference of two sets is the set of all elements which are in exactly one of the two sets. This is in contrast to the ordinary diference of two sets, which is all of the elements that are in the first set but not the second. However, if an ordinary difference function was available, we could write the symmetric difference as the union of the ordinary difference between set1 and set2, with the ordinary difference between set2 and set1. We have already written UNION, but an ordinary set difference is not yet available. Nevertheless, let's bravely proceed to write the symmetric difference in terms of the ordinary difference, then we will worry about how to write the latter:
%

```
FUNCTION SYMDIF (SET1, SET2),
  UNION (ORDDIF (SET1, SET2),  ORDDIF (SET2, SET1))
ENDFUN $
```

% Now you try to write ORDDIF. It may help you to know that it can be written very similarly to UNION: % RDS: FALSE $
% Some programmers are initially uncomfortable with the top-down approach because it makes them nervous to refer to undefined functions: there are obvious loose ends during the writing process. However, it is not necessary to understand how an auxiliary function can be written before daring to refer to it. All that is necessary is that the duty relegated to the auxiliary function be somehow more elementary than the overall duty performed by the function which refers to it.

There are necessarily loose ends during the writing of a program in any sequential order. With the bottom-up approach, the loose ends are neither written nor referred to until lower-level functions have been written. Unfortunately, as such hidden loose ends are revealed they often make apparent the need to completely reorganize and rewrite all subordinate functions into a more suitable organization. In contrast, the obvious loose ends during a top-down development provide invaluable clues about how to organize the remaining functions. Moreover, any subsequent changes tend to be easier, because communication between the functions is more localized, more independent, and more hierarchial. For example, we know that in the definition of SYMDIF we are taking the union of two DISJOINT sets, because from the definition of ORDDIF it is clear that ORDDIF (SET1, SET2) and ORDDIF (SET2, SET1) cannot have elements in common. Hence it would be more efficient merely to append the second ordinary set difference to the first ordinary set difference, or vice-versa. Unfortunately, ADJOIN does not accomplish the desired

effect.

For example, ADJOIN ('(5, 9), '(3, 7)) yields ((5, 9), 3, 7) rather than the desired (5, 9, 3, 7). What we must do is ADJOIN 9 to (3, 7), then adjoin 5 to that result. See if you can generalize this process into a function named APPEND, which returns a list consisting of the list which is its first argument appended onto the beginning of the list which is its second argument:% RDS: FALSE $
% How about: %

```
FUNCTION APPEND (LIS1, LIS2),
  WHEN EMPTY (LIS1), LIS2 EXIT,
  ADJOIN (FIRST(LIS1), APPEND (REST(LIS1), LIS2))
ENDFUN $
```

% You may not be getting tired, but my circuits are weary, so let's bring this lesson to a close. %

```
ECHO: #ECHO $          CONDENSE: #CONDENSE $

RDS () $
```

```
LINELENGTH (78) $
#CONDENSE: CONDENSE $  CONDENSE: FALSE $  #ECHO: ECHO $  ECHO: TRUE $
```

% This is the fourth is a series of muSIMP programming lessons.

Often within a function definition it is desired to form a list of values DYNAMICALLY. For example, suppose that we wish to form a list of the VALUES of the variables FIRSTNAME, LASTNAME, and MAILADDRESS. It will not do to use '(FIRSTNAME, LASTNAME, MAILADDRESS), because the quote prevents evaluation of the variables.

We can accomplish the desired effect by writing

```
ADJOIN (FIRSTNAME, ADJOIN (LASTNAME, ADJOIN (MAILADDRESS, '()))).
```

However, this rather unreadable construct is tedious to write. Fortunately, muSIMP provides a convenient function named LIST for this purpose: We can accomplish the desired effect by merely writing

```
LIST (FIRSTNAME, LASTNAME, MAILADDRESS).
```

Unlike most functions, LIST uses any number of arguments. As specific examples: %

```
FIRSTNAME: 'JOHN &
LASTNAME: 'DOE &
MAILADDRESS: 'TIMBUKTU &
```

% Now, compare using a quote with using LIST: % RDS: FALSE $
% Reversing a list is an occasional need, and it is somewhat tricky to write a function for this. The following partial definition reveals that our friends APPEND and LIST can help:

```
FUNCTION REVLIST (LIS),
  WHEN  ...  EXIT,
  APPEND ( ... ,  LIST (FIRST (LIS)))
ENDFUN $
```

See if you can successfully complete this definition. Naturally, you also have to reenter APPEND if a correct version is not around from the previous lesson. (Remember also to jot down all function definitions if you are not using a hard-copy terminal.) % RDS:FALSE $
% A well-written APPEND necessarily requires execution time which is approximately proportional to the length of its first argument. The REVLIS function outlined above invokes APPEND n times if n is the length of its original argument, and the average length of the argument to APPEND is n/2. Thus, the time is approximately proportional to n*(n/2), which is proportional to n^2.

Fortunately, an important technique called a COLLECTION VARIABLE permits list reversal in time proportional to n, yielding tremendous time savings for long lists: %

```
FUNCTION REVLIS (LIS, ANS),
  WHEN EMPTY (LIS),  ANS  EXIT,
  REVLIS (REST(LIS),  ADJOIN (FIRST(LIS), ANS))
ENDFUN $
TRACE (REVLIS) ;
REVLIS ('(1, 2, 3)) &
```

% A collection variable accumulates the answer during successive recursive invocations. Then, the resulting value is passed back through successive levels as the returned answer.

As is illustrated here, we can invoke a function with fewer arguments than there are parameters. When this is done, the extra parameters are initialized to FALSE, and they are available for use as LOCAL VARIABLES within the function body. Quite often, as in this example, the initial value of FALSE is exactly what we want, because it also represents the empty list. (When we want some other initial value, either the user can supply it, or the function can supply it to an auxiliary function which does the recursion.)

Of course, if a user of REVLIS supplies a second argument, then the function returns the reversed first argument appended onto the second argument, which is also occasionally useful.

What if the user supplies more arguments than there are parameters? The extra arguments are evaluated, but ignored. This is also occasionally convenient.

The style of programming exemplified so far is the so-called "applicative" style popularized by the influential Turing lecture of J Backus, published in the August 1978 issue of the Communications of the ACM: The emphasis is on expressions, functional composition, and recursion.

muSIMP also supports the alternative "Von Neumann" style emphasizing loops, assignments, and other side-effects. To illustrate this style, here is an alternative definition of REVLIS which introduces the LOOP construct: %

```
FUNCTION REVLIS (LIS, ANS),
  LOOP
    WHEN EMPTY (LIS),  ANS  EXIT,
    ANS: ADJOIN (FIRST(LIS), ANS),
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
```

% muSIMP has a primitively defined function named REVERSE, which has an equivalent machine language definition.

An iterative loop is an expression consisting of the keyword LOOP, followed by a sequence of one or more expressions separated by commas, followed by the matching delimiter named ENDLOOP. The body of a loop is evaluated similarly to a function body, except:

1. When evaluation reaches the delimiter named ENDLOOP, evaluation proceeds back to the first expression in the loop.

2. When evaluation reaches an EXIT within the loop, evaluation proceeds to the point immediately following ENDLOOP, and the value of the loop is that of the last expression evaluated therein.

There can be any number of conditional exits anywhere in a loop. Ordinarily there is at least one exit unless the user plans to have the loop repeat indefinitely until perhaps interrupted by typing ESCape, ALTmode or CTRL-Z. (This interrupt can succeed only if the loop invokes at least one function which is not built-into muSIMP.)

Now consider the following sequence: %

```
L1: '(THE ORIGINAL ) $
L2: '(TAIL) $
REVLIS (L1, L2) &
```

% The above definition of REVLIS makes assignments to its parameters LIS and ANS. For this example, the final assignments are LIS: '() and ANS: '(ORIGINAL, THE, TAIL). So, what do you guess are the corresponding current values for L1 and L2? See for yourself: %

```
RDS: FALSE $
```

% The assignments to parameters LIS and ANS have no effect on arguments L1 and L2! This "call-by-value" mechanism permits function definitions to freely utilize their parameters without fear of damaging the values of user's argument variables outside. Thus, ordinary function parameters are never employed for passing information back to the user. If we wish to return more than one piece of information, the most well-disciplined way to do so is to return an aggregate of the pieces as the returned value. However, another way is to make assignments within the function body to variables which are not among its parameters — so-called "fluid" or "global" variables.

As is often the case for iteration versus recursion in muSIMP, the iterative LOOP version of REVLIS is slightly faster than the recursive collection-variable version, but the latter is more compact. When there is such a trade-off between speed and compactness, a good strategy is to program for speed in the crucial few most-frequently invoked functions, and program for compactness elsewhere.

However, looping does have another advantage when it is applicable: Recursion entails a "stack" of information which grows with the depth of recursion. Consequently, even though the space allocated to the stack is quite generous, excessively deep recursion can abort a computation by exhausting this space.

For practice with loops, use one to write a nonrecursive recognizer named ISSET, which returns TRUE if its list argument contains no duplicate elements, returning FALSE otherwise. (Compare your definition with the recursive version in lesson PLES3.) % RDS: FALSE $

% Here is our solution: %

```
FUNCTION ISSET (LIS),
  LOOP
    WHEN EMPTY (LIS), EXIT,
    WHEN MEMBER (FIRST(LIS), REST(LIS)),  FALSE  EXIT,
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
```

% Another good exercise adapted from PLES3 is to use a loop to write a nonrecursive function named SUBSET, which returns TRUE if its first argument is a subset of its second argument, returning FALSE otherwise:

```
%   RDS: FALSE $
```

% A BLOCK is another control construct which is sometimes convenient, particularly in conjuction with the Von Neumann style. As an illustration of its use, the following iterative version of the MAKESET function from PLES3 returns a set composed of the unique elements in the list which is its first argument: %

```
FUNCTION MAKESET (LIS, ANS),
  LOOP
    WHEN EMPTY (LIS),  ANS  EXIT,
    BLOCK
      WHEN MEMBER (FIRST(LIS), ANS),  EXIT,
      ANS:  ADJOIN (FIRST(LIS), ANS)
    ENDBLOCK,
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
MAKESET ('(FROG, FROG, FROG, TERMITE)) &
```

% When evaluation reaches an EXIT, it proceeds to the point following the next ENDBLOCK, ENDLOOP, or ENDFUN delimiter — whichever is nearest. Thus, BLOCK provides a means for alternative evaluation paths which rejoin within the same function body or loop body, without causing an exit from that body. The first expression in a block must be a conditional-exit (anything else can be moved outside anyway), but since there can be any number of other conditional exits or other expressions within the block, the block provides a very general structured control mechanism. For example, the CASE-statement and IF-THEN-ELSE construct of some other languages are essentially special cases of a block.

You may not have noticed, but the loop version of MAKESET has the effect of reversing the order of the set elements. Using ADJOIN in a loop generally has this effect, which is why it is so suitable for REVERSE. With sets, incidental list reversal is perhaps acceptable, but for most applications of lists it is not. We could of course use a preliminary or final invocation of REVERSE so that the final list would emerge in the original order, but that would relinquish the speed advantage of the loop approach, while further increasing its greater bulk. Thus, recursion is usually preferable to loops when ADJOIN is involved. For example, recursion is used almost exclusively to implement muMATH, because its symbolic expressions are represented as ordered lists.

Loops are also less applicable to general tree structures than to lists, but it is often possible to loop on the REST pointer while recursing on the first pointer, or vice-versa, particularly if ADJOIN is

not involved. For example, compare the following semi-recursive definition of #ATOMS with the fully-recursive one in FLESL: %

```
FUNCTION #ATOMS (U, N),
  N: 1,
  LOOP
    WHEN ATOM (U), N EXIT,
    N: N + #ATOMS (FIRST(U)),
    U: REST (U)
  ENDLOOP
ENDFUN $
#ATOMS ('((3 . FOO), BAZ)) ;
```

% If the answer surprises you, don't forget the FALSE which BAZ is implicitly dotted with.

See if you can similarly write a semi-recursive function named DUP which does what the infix operator named "=" does: % RDS: FALSE $
% Those of you with previous exposure to only Von Neumann style programming undoubtedly feel more at home now. The reason we postponed revealing these features until now is that we wanted to force the use of applicative programming long enough for you to appreciate it too. Naturally, one should employ whichever style is best suited for each application, so it is worthwhile to become equally conversant with both styles.

Thus endeth the sermon. %

```
ECHO: #ECHO $          CONDENSE: #CONDENSE $
RDS () $
```

```
LINELENGTH (78) $
#CONDENSE: CONDENSE $  CONDENSE: FALSE $  #ECHO: ECHO $  ECHO: TRUE $
```

% This is the fifth in a sequence of muSIMP programming lessons.

In the previous lesson our original version of REVERSE, called REVLIS, required time proportional to n^2, where n is the length of the first argument. We then showed how a collection variable or a loop could yield a much faster technique using time proportional only to n. Now, let's consider the speed of some of the other set functions that we defined:

Whether iterative or recursive, MEMBER can require a number of equality comparisons equal to the length of its second argument. Whether defined iteratively or recursively, SUBSET, EQSET, UNION, and INTERSECTION all require a membership test for each element of one argument in the list which is the other argument. Thus, these definitions can all consume computation time which grows as the product of the lengths of the two arguments. By similar reasoning, the one-argument functions ISSET and MAKESET are seen to require time proportional to the square of the length of their argument. Data-base applications and others can involve thousands of set operations on sets having thousands of elements, so it is worthwhile to seek methods for which the computation time grows more slowly with set size.

In muSIMP, every name has an associated PROPERTY LIST which is immediately accessible in an amount of time that is independent of the total number of names in use. Provided the elements of the sets are all names, this permits techniques for the above set operations requiring time proportional merely to the length of the one set or to the sum of the lengths of the two sets.

A property list is a list of dotted pairs. The first of each dotted pair is a atom called the KEY or INDICATOR, and the rest of each dotted pair is an expression called the associated INFORMATION. For example, in a meteorological data-base application, the name HONOLULU might have the property list

```
((RAIN . 2), (HUMIDITY . 40), (TEMPERATURE, 58, 96))
```

The function used in the form GET (name, key) returns the information which is dotted with the value of "key" on the property list of the value of "name", returning FALSE if no such key occurred on the property list.

A command of the form REMPROP (name, key) has the side effect of deleting from the property list of "name" the first dotted pair beginning with the value of "key", if any. REMPROP returns FALSE if no such indicator occurs on that property list, returning TRUE otherwise.

A command of the form PUT (name, key, information) causes a command of the form REMPROP (name, key) to be executed, after which the value of "key" dotted with the value of "information" is put on the

property list of the value of "name". PUT returns the value of "information".

Using property lists, the basic technique for accomplishing our various operations on two sets of names is:

1. For each name in one of the two sets of names, store TRUE under the key SEEN.

2. For each name in the other set, check to determine whether or not the name has already been seen, and act accordingly.

3. For each name in the first set, remove the property SEEN so that we won't invalidate subsequent set operations which utilize any of the same elements.

A simpler variant of this idea is applicable to the one-argument functions named ISSET and MAKESET.

As an example, here is UNION defined using this technique together with the applicative style: %

```
FUNCTION UNION (SET1, SET2),
  MARK (SET1),
  UNMARK (SET1, UNIONAUX (SET2))   ENDFUN $
FUNCTION MARK (SET1),
  WHEN EMPTY (SET1), EXIT,
  PUT (FIRST(SET1), 'SEEN, TRUE),
  MARK (REST (SET1))   ENDFUN $
FUNCTION UNIONAUX (SET2),
  WHEN EMPTY (SET2), SET1  EXIT,
  WHEN GET (FIRST(SET2), 'SEEN), UNIONAUX (REST(SET2))  EXIT,
  ADJOIN (FIRST(SET2), UNIONAUX(REST(SET2)))   ENDFUN $
FUNCTION UNMARK (SET1, ANS),
  WHEN EMPTY (SET1), ANS  EXIT,
  REMPROP (FIRST(SET1), 'SEEN),
  UNMARK (REST(SET1), ANS)   ENDFUN $
```

% Each time any function is invoked, the outside values of its parameter names, if any, are "stacked" away to be restored later, just prior to return from that invocation. If a function refers to a variable which is not among its parameters, then the most recent value of the variable on the stack is used. Thus, when UNIONAUX is invoked from within UNION, SET1 in the definition of UNIONAUX refers to the argument value associated with that parameter of UNION. This treatment is called "dynamic binding", and a reference such as to SET1 in UNIONAUX is called a "fluid reference". We could have avoided this by making SET1 be an argument and a parameter to UNIONAUX, but that would have made the program slightly slower and more bulky. However, fluid variables make programs much harder to debug and maintain, especially if assignments are made to them in functions other than the ones which establish them. Consequently, we recommend generally avoiding fluid variables. The only reason we used one here is to introduce the concept to issue this advice.

Values assigned at the top-level of muSIMP, outside all function definitions, are called GLOBAL values. Examples are the initial values of muSIMP control variables such as RDS, ECHO and CONDENSE, or of muMATH control variables such as PBRCH or PWREXPD. Reference to a global value from within a function definition is not quite as confusing as reference to a fluid value, and it is indeed onerous to creat numerous long lists of parameters in order to pass such environmental control values through a long sequence of function definitions for use deep within.

However, here too it is at the very least considered bad programming style to unnecessarily modify such global values from within a function without restoring the values before exiting from the function. In fact it is generally bad manners for any program file to modify global values if the modification is merely incidental to the central purpose. That is why these lessons carefully save the prevailing values of the control variables named ECHO and CONDENSE, then restore these values just prior to the end of the file. (It is truely annoying to have someone else's program litter your environment unnecessarily.)

The property-list technique for set operations is one which we think is more naturally implemented using the Von Neumann programming style. Try to write such a version of UNION: % RDS: FALSE $
% Now, using either style, write an INTERSECTION function using the property-list technique: % RDS: FALSE $
% One does not usually take the FIRST or REST of an atom intentionally, but they do in fact have well-defined values: The FIRST of an atom is its value, and the REST of an atom is its property list. For example: %

```
WEATHER: 'FOUL $
FIRST (WEATHER) ;
PUT ('WEATHER, 'TEMPERATURE, -3) &
PUT ('WEATHER, 'WIND, '((NORTH . WEST), 30)) &
REST ('WEATHER) &
```

% This is true of integer atoms too, though it is usually pointless to put anything on the property list of an integer, because integers are not stored uniquely: %

```
FIRST (7);
REST (7);
NINE: 9 $
PUT (NINE, 'TESTING, '(1, 2, 3)) &
GET (NINE, 'TESTING) &
GET (9, 'TESTING) &
```

% Since all nodes and atoms have a FIRST and a REST which are either nodes or atoms, misuse of these selectors can't accidently give access to the machine language, stack, print names, or anywhere else which could inadvertently compromise the integrity of muSIMP. Thus, inadvertent omission of a termination test in functions which follow chains of pointers is likely to be revealed by stack exhaustion in the case of recursion, and by an infinite loop in the case of iteration.

It is common practice to use EMPTY to test for the end condition as a function proceeds down a list. If such a function is inadvertently given a non-list (i.e. a Non-FALSE atom or a structure whose final REST cell points to a Non-FALSE atom), the function will use the FIRST cell of that atom (i.e. its Value cell) as an element of the list and the REST cell of the atom (i.e. its Property List cell) as the REST of the list. Generally the Property List is a well defined list so the EMPTY test will ultimately cause termination with no ill affects.

We prefer to have non-list arguments give more predictable results confined to the argument. Thus, our internal implementations of MEMBER, REVERSE, and any other functions ordinarily applied to lists use ATOM rather than EMPTY as the termination test. This is slightly faster too, so you may wish to generally avoid EMPTY in favor of ATOM. Alternatively, you can redefine EMPTY to print and return an error message when given a nonFALSE atom: %

```
FUNCTION EMPTY (LIS),
  WHEN ATOM (LIS),
    WHEN EQ (LIS, FALSE), EXIT,
    PRINT ("*** Warning: EMPTY given nonlist ") EXIT
ENDFUN $
EMPTY (5) $
```

% This is our first example illustrating the fact that conditional exits can be nested arbitrarily deep. The same is true of loops or blocks. This example also illustrates the PRINT function, which prints its one argument the same way that expressions terminated with an ampersand are printed. There is an analogous function named PRTMATH which prints its one argument the same way that expressions terminated with a semicolon are printed.

When functions are called with fewer actual arguments than the function has formal arguments, the remaining formal arguments are assigned the value FALSE. This provides a convenient mechanism for automatically inserting default values for these extra arguments. When an argument evaluates to FALSE, the function can assign the appropriate default value. For example, if the user omits the drive as the third argument of RDS, that function uses the currently logged in drive (i.e. the drive indicated by the last operating system prompt given before entering muSIMP).

There are instances where it is desirable to permit a function to have an arbitrary number of arguments. This is accomplished by making the formal parameter list of a function definition be an atom or non-list rather than a list. The arguments are passed to the function as a single list of argument values, from which the function can extract the values. For example, it is convenient to have a function named MAX which returns the largest of one or more argument values. We can implement this as follows: %

```
FUNCTION MAX ARGLIS,
  MAXAUX (FIRST(ARGLIS), REST(ARGLIS))
ENDFUN $
FUNCTION MAXAUX (BIGGEST, UNTRIED),
  WHEN EMPTY (UNTRIED), BIGGEST EXIT,
```

```
  WHEN BIGGEST > FIRST(UNTRIED), MAXAUX (BIGGEST, REST(UNTRIED)) EXIT,
  MAXAUX (FIRST(UNTRIED), REST(UNTRIED))
ENDFUN $
MAX (7) ;
MAX (3, 8, -2) ;
```
% This collection of arguments into a list is called NOSPREAD, to distinguish from the SPREAD brand of peanut butter.

More generally, muMATH permits a combination of the 2 techniques: If a parameter-list is a dotted-pair of two names or a list whose last element is a dotted pair of two names, then the last parameter name accumulates a list of any excess arguments beyond those spread to the other parameter names. Thus, we can simplify our definition of MAX to: %

```
FUNCTION MAX (FRST . OTHERS),
  MAXAUX (FRST, OTHERS)
ENDFUN $
```
% Would you like to try this technique? Appropriate candidates include MIN, UNION, and INTERSECTION. % RDS: FALSE $
% Now, suppose that for some reason we already have a list of integers such as %

```
NUMBLIS: '(18, 3, 7, 91, 12, 2) $
```
% and we want to find their maximum. The expression MAX (NUMBLIS) will not work, because MAX is designed for numeric arguments, not for a list of numbers. We could of course extract the elements and feed them individually to MAX, but this is awkward, especially if we are referring to MAX inside a function and we do not know ahead of time how many integers are in NUMBLIS. Fortunately there is a convenient function named APPLY, which applies the function whose name is the value of its first argument to the argument list which is the value of its second argument. Consequently, we need merely write %

```
APPLY ('MAX, NUMBLIS) &
```
% APPLY works on either SPREAD or NOSPREAD functions. Why don't you try out a few examples: % RDS: FALSE $
% A function written in muSIMP-79 is stored internally as a nested list, and the function named GETD returns a pointer to this list. Consequently, to see what the internal representation of UNION looks like: %

```
GETD (UNION) &
```
% GETD returns TRUE if the definition is in machine language, and GETD returns FALSE if there is no function definition for its argument. Those who are curious may wish to use this function to experimentally determine the correspondence between the external and internal forms of a function definition. This can be useful for revealing bugs arising from misconceptions about how the parser regards certain constructs. All we want to point out here is that since function definitions are represented as lists, muSIMP functions can easily operate upon other muSIMP functions. This makes it easy to write muSIMP programs which service other muSIMP programs. Examples include muSIMP-oriented editors, cross-reference programs, debuggers, verifiers, statistics-gatherers, pretty-printers, file comparators, and compilers. The

internal representation also makes it possible for functions to modify each other dynamically, as they execute. The implications for artifical intelligence are intriguing to contemplate.

REMD is a related command which clears any function definition existing under the name which is the value of its argument. For example, %

```
REMD (UNION) &
GETD (UNION) &
```

% One good use of REMD is to free space occupied by functions which are no longer needed immediately, in order to provide enough space for a more urgent need. For example, suppose that in muMATH a problem requires the SOLVE package followed by the MATRIX package, but there is not room enough for both packages to coexist in the amount of memory present on the machine. Then, after using the SOLVE package but before reading in the MATRIX package we could remove function definitions for SOLVE by commands such as

```
REMD (SOLVE) $  REMD (SOLEXP) $  ...
```

Less typing would be involved if we defined a command named MULTIREMD, which for an argument which is a list of names, successively applies REMD to each name. In this example it is the side efects rather than the returned value which is of interest, so MULTIREMD can return whatever is the least trouble. MULTIREMD is trivial to write, using either recursion or iteration, because the same "program schema" occurs so often: Walk down a list, successively applying a function of one argument to each element of the list, then return anything. This observation leads to the following idea: Let's write a function which, given the name of any function of 1 argument, together with a list, successively applies the function to the elements, then returns anything convenient: %

```
FUNCTION MAP (FUNNAME, LIS),
  LOOP
    WHEN EMPTY (LIS),  EXIT,
    APPLY (FUNNAME, FIRST(LIS)),
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
```

% Then, for example, we could write

```
MAP ('REMD, '(SOLVE, SOLEXP, ... )) $
```

What we have done is to separate the general-purpose control-sequence from the specific tasks which can use it. This division of labor accomplishes two useful things:

1. Program space savings can accrue for each use of MAP with a different function, beyond the first, because essentially duplicate control sequences are avoided.

2. Once the meaning of MAP becomes familiar, the program is more readable, because MAP ('REMD, '(SOLVE, SOLEXP, ...)) is

then instantly understood to mean REMD all of SOLVE, SOLEXP, etc.. In contrast, the altermative form MULTIREMD ('(SOLVE, SOLEXP, ...)) requires the user to check the definition of MULTIREMD to be sure the purpose is correctly understood.

Another frequent need is to walk down a list, applying a function of one argument to each element, but return the list of results. Write a mapping function of this kind, called MAPLIST since it returns a list. Then, try
MAPLIST ('-, '(3, 8, 14)), and MAPLIST ('NOT, '(TRUE, FALSE, MAYBE))
%RDS: FALSE $
% MAP and MAPLIST are the most widely applicable mapping functions, but if you grow to like mapping functions you may develop a large suite of them. For example:

1. For functions of two arguments you could have a map function of the form MAP2 (function name, list1, list2) or MAP2 (function name, list of pairs). Since much of muMATH is stored on property lists, this could be used to apply REMPROP appropriately to help delete high-level muMATH packages in order to make space. (Here is an idea: for each muMATH file, write a corresponding file of type DEL, which has an appropriate command of the form MAP ('REMD, ...), together with one of the form MAP2 ('REMPROP, ...). Then, to delete the SOLVE package from memory, one merely issues the command RDS (SOLVE, DEL, drive).)

2. For functions of two arguments you could have a mapping function used in the form MAP2LIST (function name, list1, list2) or MAP2LIST (function name, list of pairs), which is like MAP2 but returns a list of results.

3. For general trees you could have a mapping function called TREEMAP which applies a function to the atoms in a tree, and there could be a similar one called TREEMAPTREE which is similar but returns a tree. %

ECHO: #ECHO $ CONDENSE: #CONDENSE $

RDS () $

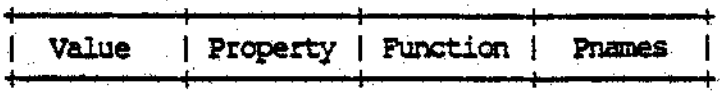
muSIMP-79 Primitive Data Structures

The Soft Warehouse 11/26/79

I. DATA STRUCTURES.

muSIMP data is comprised of names, numbers, and nodes. Each type is recognizable and consists of a fixed number of "pointer" cells containing memory addresses. The cells can either point to other objects or to special-purpose entities outside the pointer space of objects. However, all three types have a FIRST cell and a REST cell. Moreover, these FIRST/REST cell pairs can only point to other objects within the pointer space. This eliminates the need for time-consuming run-time type-checks in the crucial selector functions which fetch these pointers.

A. Names



A name is a recognizable, structured object consisting of four pointer cells. Names are uniquely stored so that duplicate names cannot coexist in storage. Here are the uses of the four cells:

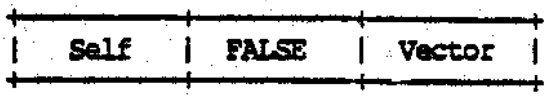
1. The FIRST or value cell contains a pointer to the name's current value which is used by the evaluation functions. The value of a name is initialized to a self-reference of the name; however, it is modified by the assignment functions and when the name is used as a formal parameter in a function definition.

2. The REST or property list cell contains a pointer to the name's property list which is used by the property functions. Elements of this list are indicators dotted with the corresponding values. Property lists are initially set to the empty list.

3. The Function cell contains a pointer to the name's function definition if any. The contents of this cell can't be accessed except as function applications, and the contents can't be modified except by means of the function definition primitives. When a new name is first created, its function cell is initialized to the undefined-function trap routine.

4. The Pname cell contains a pointer to the name's ASCII print-name string, which can be of arbitrary length. Access to this cell is restricted to the I/O and sub-atomic primitives. Print names are defined when a name is first used, and they cannot be modified or expunged.

B. Numbers



A number is a recognizable, structured object consisting of three pointer cells. Numbers are not uniquely stored, so duplicate numbers might coexist in storage. The cells are used as follows:

1. The FIRST cell contains a pointer to itself.

2. The REST cell is initialized to FALSE.

3. The Number Vector cell contains a pointer to the actual number, which consists of a signed vector of up to 254 bytes. Thus, the magnitude of numbers is limited to 256^254, which is approximately 10^611.

C. Nodes

Binary trees are the primary data structure in muSIMP. Internally they are implemented as a network of cell pairs called nodes. Each node consist of a FIRST cell and a REST cell. As mentioned earlier, the node's cells can only point to other bonified muSIMP data objects; either a name, a number, or a node. Nodes are often called "dotted-pairs", because of their linearized external notation produced by PRINT or accepted by READLIST: The notation

(X . Y)

represents a node whose FIRST cell points to the object X, and whose REST cell points to the object Y. Although the dot notation is more general, it is often more convenient to think of data as a linear list than as a deeply nested binary tree. For this purpose, lists are recursively defined as follows:

1. The empty list is denoted by the name FALSE.

2. If Y is a list and X an object, then (X . Y) is a list.

A list of objects is printed by the function PRINT as a sequence of its elements separated by commas and delimited by parenthesis. The function READLIST recognizes this notation for input. For example, if Y is the list (Y1, Y2, ..., Yn) then the dotted pair (X . Y) is printed as

(X, Y1, Y2, ..., Yn)

Conversely, the input of the form (X, Y1, Y2, ..., Yn) is recognized as (X . Y) by the READLIST function.

II. MEMORY MANAGEMENT.

Dynamic, transparent memory management gives muSIMP much of its inherent power. Ideally, at any given time during the execution of a program, all of the memory not actually required to decribe the state of the machine should be available for any subsequent program use. This is approximated in muSIMP-79 by first partitioning the available resources into the various data-spaces and then recycling storage within each of these spaces as required. Normal stack operations continuously reclaim the stack space; whereas, an automatically invoked garbage collector reclaims the remaining spaces.

A. Initial Data-space Partition

During the initialization phase of muSIMP, the amount of read/write memory available to the interpreter is first computed. Memory is then partitioned into four distinct data-spaces using the following proportions:

4/32	Atom Space	Name and number pointer cells.
3/32	Vector Space	Print-name strings and number vectors.
23/32	Node Space	Node cell pairs.
2/32	Stack Space	Control/value stack.

Based on our experience, these proportions provide a reasonable balance between the spaces for most applications.

B. Garbage Collection

New data structures are generally constructed during the execution of a muSIMP program, while others are implicitly discarded as they become un-referenced. When the construction process uses up all available resources, a garbage collector routine is called to reclaim the storage space vacated by discarded data structures, so that the user program can continue. In muSIMP-79 the exhaustion of resources in either the atom, vector, or node spaces will cause collection to occur. Those data structures accessible by means of chaining through pointer cells beginning either from a name cell or from a value stack entry are marked. Then during the second pass all the unmarked nodes and numbers are collected for re-use, while simultaneously removing the mark on the accessible nodes.

Although garbage collection is automatic, it is not entirely invisible to the user since it periodically causes a pause in the execution of a program. About 1.5 seconds is required for the collection process in a 48K byte muSIMP-79 system using a 2MHz CPU clock. Normally this is of no concern to the programmer; however, it should be considered in the design of real time systems. A phenomenon know as thrashing occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small amount of nodes. This can be resolved by increasing the computer's memory size or decreasing the amount of program and data storage requirements.

III. ERROR And INTERRUPT TRAPS

If there is a reasonable interpretation for a construct, muSIMP generally uses it. Consequently, error traps are induced only by situations for which there is no satisfactory recovery. Examples are the exhaustion of available data space or disk I/O errors. On the other hand, a software interrupt is caused by an interrupt character (i.e. an ESC, ALT, or Ctrl-Z) received from the terminal, which can be sent at any time. When a particular trap occurs, the appropriate diagnostic and the following "options" message are sent to the terminal:

EXECUTIVE: ESC, ALT, Ctrl-Z; RESTART: RUB, DEL; SYSTEM: Ctrl-C?

The user may then type one of the appropriate alternative option characters. The "EXECUTIVE" option is the least drastic since it merely causes control to return to the muSIMP executive driver loop, without changing function definitions, property values, or name values, from what they were just prior to the interrupt. The second option destroys all non-primitive muSIMP functions, property values, and name values, then restarts muSIMP afresh. Finally, the "SYSTEM" option terminates muSIMP, and returns control to the operating sytem.

A. Data Space Overflow

As discussed in Section II, there are four distinct data spaces in muSIMP to accommodate the various data types. Normally, automatically invoked garbage collections will provide sufficient space in each area to continuously satisfy the demands of user programs. However, in the event all of the available resources in an area become exhausted, an error trap will occur and one of the following diagnostic messages will be displayed on the terminal:

NODE Space Exhausted
ATOM Space Exhausted
VECTOR Space Exhausted
STACK Overflow

B. Disk File I/O Errors

Disk errors may be caused by insufficient disk space, attempts to read past the end-of-file, or hardware malfunctions. The read and write disk error diagnostics are respectively:

End of File or READ Error
No Disk Space or WRITE Error

C. Undefined Numerical Operations

If the second argument to any of the functions QUOTIENT, MOD, or DIVIDE is 0, a zero-divide trap occurs with the following diagnostic:

ZERO Divide Error

D. Input Syntax Error

The only syntax error trap caused by the function READ is when a closing right parenthesis is not found when using the 'dot' notation. The diagnostic is:

Input Syntax Error

Function PARSE can produce syntax error traps together with diagnostics of the following forms:

*** SYNTAX ERROR: expression USED AS NAME,

*** SYNTAX ERROR: expression USED AS PREFIX OPERATOR,

*** SYNTAX ERROR: expression USED AS INFIX OPERATOR,

*** SYNTAX ERROR: delimiter NOT FOUND,

where "expression" is the apparent offending portion of the input, and where "delimiter" is an apparently missing right delimiter such as a right parenthesis, ENDFUN, ENDSUB, EXIT, ENDLOOP, or ENDBLOCK. In any event, the remainder of the input from the point of confusion through the next terminator, such as ";", "$", or "&", is output to the terminal to help indicate the probable neighborhood of the cause. Examples which provoke the above four types are respectively:

```
5 (X);              {perhaps 5*(X) was intended?}

X Y;                {perhaps X*Y  was intended?}

X*/Y;               {perhaps X/Y was intended?}

WHEN ATOM(X, EXIT       {perhaps WHEN ATOM(X), EXIT was
                         intended}.
```

IV. PRIMITIVELY DEFINED FUNCTIONS.

The muSIMP (Structured IMPlementation) Language is a high level computer language ideally suited for symbolic and semi-numerical processing. Currently, it is implemented by means of a bootstrap file MUSMORE.MUS which is automatically loaded prior to using the language. For an interactive introduction to the features available in muSIMP, the tutorial lesson files, beginning with PLES1.TRA, may be executed. See the description on how to take the programming lessons in LESSONS.TXT.

Every language must be described in terms of some language, which must be described in terms of some language, etc. Thus it is clear that at some point we must appeal to assumed inborn or culturally acquired understanding. This unnecessary sequence of "buck passing" can be avoided by using a somewhat circular description of muSIMP. In other words muSIMP-79 can be described in terms of muSIMP supplemented with English where necessary. Use of such a description requires some prerequisite knowledge of muSIMP gained by other means, just as use of an English dictionary requires some prerequisite knowledge of English.

After one has initially learned the basics of muSIMP from the lessons, this type of reference manual has the advantage of being compact while requiring mastery of no auxiliary notations. In addition it provides excellent, nontrivial examples of structured programs written in muSIMP.

The following is a description of all of the primitively defined user-level functions, operators, control constructs, and control variables in muSIMP-79. For descriptive purposes only, we introduce some fictitious functions which are unavailable to the user. They are indicated by being unnumbered and are also unindexed. Lower-case type is employed where English is used rather than legitimate muSIMP program constructs.

A. Selector Functions

1. FUNCTION FIRST (X),
   the contents of the FIRST cell of X,
   ENDFUN;

   Interpretations:
   a. The first item of a list X,
   b. The left element of a dotted-pair X,
   c. The value of an atom X.

2. FUNCTION REST (X),
   the contents of the REST cell of X,
   ENDFUN;

   Interpretations:
   a. The tail of a list X,
   b. The right element of a dotted-pair X,
   c. The property list of an atom X.

3. FUNCTION SECOND (X),
   FIRST (REST (X)),
   ENDFUN;

4. FUNCTION RREST (X),
   REST (REST (X)),
   ENDFUN;

5. FUNCTION THIRD (X),
   FIRST (REST (REST (X))),
   ENDFUN;

6. FUNCTION RRREST (X),
   REST (REST (REST (X))),
   ENDFUN;

B. Constructor Functions

1. FUNCTION ADJOIN (X, Y),
   a new cell-pair whose FIRST cell is X and whose REST cell is Y,
   ENDFUN;

   Interpretations:
   a. A list whose first element is X and whose tail is Y,
   b. A dotted-pair whose left element is X and whose right element is Y.

2. SUBROUTINE LIST (X1, X2, ..., Xn),
   WHEN n = 0, FALSE EXIT,
   ADJOIN (EVAL (X1), LIST (X2, X3, ..., Xn)),
   ENDSUB;

   Interpretation: The list (X1, X2, ..., Xn).

3. FUNCTION REVERSE (X, Y),
   WHEN ATOM (X), Y EXIT,
   REVERSE (REST (X), ADJOIN (FIRST (X), Y)),
   ENDFUN;

   Interpretation: The reverse of the list X. If a second argument Y is given, the reversed list is appended to the beginning of the object Y.

4. FUNCTION OBLIST (),
   a list of the current built-in and user-introduced names,
   ENDFUN;

   Interpretation: The object (name) list.

C. Modifier Functions

1. FUNCTION REPLACEF (X, Y),
   FIRST cell of X: Y,
   X,
ENDFUN;

Interpretations:
   a. Replace the first element of a list X by Y,
   b. Replace the left element of a dotted-pair X by Y,
   c. Replace the value of an atom X by Y.

2. FUNCTION REPLACER (X, Y),
   REST cell of X: Y,
   X,
ENDFUN;

Interpretations:
   a. Replace the tail of a list X by Y,
   b. Replace the right element of a dotted-pair X by Y,
   c. Replace the property list of an atom X by Y.

3. FUNCTION CONCATEN (X, Y),
   WHEN ATOM (X), Y EXIT,
   WHEN ATOM (REST (X)), REPLACER (X, Y) EXIT,
   CONCATEN (REST (X), Y),
   X,
ENDFUN;

Interpretation: Concatenate, without adjoining, the list Y onto the right end of the list X.

D. Recognizer Functions

1. FUNCTION NAME (X),
   WHEN X is a name, EXIT,
   ENDFUN;

   Interpretation: Recognize objects which are names.

2. FUNCTION INTEGER (X),
   WHEN X is an integer, EXIT,
   ENDFUN;

   Interpretation: Recognize objects which are integers.

3. FUNCTION ATOM (X),
   NAME (X) OR INTEGER (X),
   ENDFUN;

   Interpretation: Recognize objects which are atoms.

4. FUNCTION EMPTY (X),
   X = FALSE,
   ENDFUN;

   Interpretation: Recognize the empty list.

5. FUNCTION POSITIVE (X),
   X > 0,
   ENDFUN;

   Interpretation: Recognize positive integers.

6. FUNCTION NEGATIVE (X),
   X < 0,
   ENDFUN;

   Interpretation: Recognize negative integers.

7. FUNCTION ZERO (X),
   X = 0,
   ENDFUN;

   Interpretation: Recognize zero.

Note: All recognizers return TRUE or FALSE.

E. Comparator Functions and Operators

1.
```
FUNCTION EQ (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),  X = Y  EXIT,
    WHEN X and Y point to the same object,  EXIT,
ENDFUN;
```

Interpretation: The identity comparison of X and Y.

2.
```
PROPERTY RBP, =, 80;
PROPERTY LBP, =, 80;

FUNCTION = (X, Y),
    WHEN ATOM (X),  EQ (X, Y)  EXIT,
    WHEN ATOM (Y),  FALSE  EXIT,
    WHEN FIRST(X) = FIRST(Y),  REST(X) = REST(Y)  EXIT,
ENDFUN;
```

Interpretation: The infix equality operator, =, treats X and Y as being equal if and only if they have isomorphic tree structures with identical atomic terminal nodes.

3.
```
FUNCTION ORDERP (X, Y),
    WHEN the address of the object X is less than the
        address of the object Y,  EXIT,
ENDFUN;
```

Interpretation: A generic ordering function for system names based on their order of introduction.

4.
```
PROPERTY RBP, >, 80;
PROPERTY LBP, >, 80;

FUNCTION > (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),  X > Y  EXIT,
ENDFUN;
```

5.
```
PROPERTY RBP, <, 80;
PROPERTY LBP, <, 80;

FUNCTION < (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),  X < Y  EXIT,
ENDFUN;
```

Note: All comparators return TRUE or FALSE.

F. Logical Operators

1. PROPERTY RBP, NOT, 70;

```
FUNCTION NOT (X),
    X = FALSE,
ENDFUN;
```

Interpretation: NOT is a prefix operator with right binding power 70.

2.
```
PROPERTY RBP, AND, 60;
PROPERTY LBP, AND, 60;

SUBROUTINE AND (X1, X2, ..., Xn),
    WHEN n = 0,  EXIT,
    WHEN NOT EVAL (X1),  FALSE  EXIT,
    AND (X2, X3, ..., Xn),
ENDSUB;
```

Interpretation: AND is a logical infix operator with a left and right binding power of 60.

3.
```
PROPERTY RBP, OR, 50;
PROPERTY LBP, OR, 50;

SUBROUTINE OR (X1, X2, ..., Xn),
    WHEN n = 0,  FALSE  EXIT,
    WHEN EVAL (X1),  EXIT,
    OR (X2, X3, ..., Xn),
ENDSUB;
```

Interpretation: OR is a logical infix operator with a left and right binding power of 50.

Note: All logical operators return TRUE or FALSE, and any nonFALSE logical operand has the same effect as TRUE.

G. Assignment Functions

1. FUNCTION ASSIGN (X, Y),
   FIRST cell of X: Y,
   Y,
ENDFUN;

Interpretation: Set the value of the atom X to Y, and return Y.

2. PROPERTY RBP, :, 20;
PROPERTY LBP, :, 180;

SUBROUTINE : (X, Y),
   ASSIGN (X, EVAL (Y)),
ENDSUB;

Interpretation: Set the value of 'X to Y, and return Y. ":" is an infix operator with left binding power 180 and right binding power 20. Evaluation of this form returns the value of the expression, after achieving the side effect of assigning the value of the expression to the name.

Note: Assignments to non-names are allowable, having an effect similar to REPLACEF, but returning a different pointer.

H. Property Functions

1.
```
FUNCTION ATSOC (X, Y),
    WHEN ATOM (Y), Y EXIT,
    WHEN ATOM (FIRST (Y)), ATSOC (X, REST (Y)) EXIT,
    WHEN EQ (FIRST (FIRST (Y)), X), FIRST (Y) EXIT,
    ATSOC (X, REST (Y)),
ENDFUN;
```

Interpretation: The first non-atomic object on the "asSOCiation" list Y whose ATomic FIRST cell is X.

2.
```
FUNCTION GET (X, Y),
    X: ATSOC (Y, REST (X)),
    WHEN ATOM (X), FALSE EXIT,
    REST (X),
ENDFUN;
```

Interpretation: The property value associated with the indicator Y on the property list of X.

3.
```
FUNCTION PUT (X, Y, Z),
    WHEN EMPTY (GET (X, Y)),
        REPLACER (X, ADJOIN (ADJOIN (Y, Z), REST (X))),
        Z EXIT,
    REPLACER (ATSOC (Y, REST (X)), Z),
    Z,
ENDFUN;
```

Interpretation: Place on the property list of the atom X under the indicator Y the property value Z, destroying any previous value under the same indicator.

4.
```
FUNCTION REMPROP (X, Y),
    WHEN ATOM (REST (X)), REST (X) EXIT,
    WHEN EQ (FIRST (SECOND (X)), Y),
        Y: REST (SECOND (X)),
        REPLACER (X, RREST (X)),
        Y EXIT,
    REMPROP (REST (X), Y),
ENDFUN;
```

Interpretation: Remove from the property list of X the property value associated with the indicator Y.

5. PROPERTY name, atom, value
read two names X and Y, then parse an expression Z.
then without evaluating any of them, place Z on the
property list of X, under key Y.
then return Z;

PROPERTY is a data-base construct which returns a list of name and atom after accomplishing the side effect of storing the value on the property list of the first operand, under the key which is the second operand. Any previous value on that property list under the same key is deleted, with a corresponding warning message. The three operands of PROPERTY are automatically quoted, so that, for example, they can be unquoted operators.

I. Definition Functions

1. FUNCTION GETD (X),
   WHEN NOT (NAME (X)), FALSE EXIT,
   WHEN X is not a defined function, FALSE EXIT,
   WHEN the function cell of X points to a machine
   language function, EXIT,
   the object pointed to by the function cell of X,
   ENDFUN;

   Interpretation: The definition of the function named X.

2. FUNCTION PUTD (X, Y),
   WHEN NOT (NAME (X)), FALSE EXIT,
   function cell of X: Y,
   Y,
   ENDFUN;

   Interpretation: Place a pointer to the definition Y in
   the function cell of X.

3. FUNCTION MOVD (X, Y),
   WHEN NOT (NAME (X)) OR NOT (NAME (Y)), FALSE EXIT,
   function cell of Y: function cell of X,
   GETD (Y),
   ENDFUN;

   Interpretation: Copy the definition of the function
   named X to Y.

4. FUNCTION REMD (X),
   WHEN NOT (NAME (X)), FALSE EXIT,
   function cell of X: undefined,
   GETD (old definition of X),
   ENDFUN;

   Interpretation: Remove the function definition from X.

5. PROPERTY PREFIX, FUNCTION,
   parse a function definition, then use PUTD to put it
   in the function cell of the function name,
   then return the function name.

   Interpretation: FUNCTION is the leading keyword of a
   control construct which has the general form:

```
FUNCTION name parameters,
  task1,
  task2,
  ...
  taskn
ENDFUN
```

The name can be omitted when there is no need to refer to it, such as when a nonrecursive function is stored on a property list for use by APPLY. "parameters" can be an arbitrary symbolic expression. When "parameters" is any name, except "FALSE", the function may be subsequently called with an arbitrary number of arguments and passed to the function as a list assigned to the argument. If "parameters" is not a name, the first argument in a call to the function is assigned to the FIRST of "parameters" and the REST of the arguments is assigned to the REST of the "parameters" in an identical manner.

Task evaluation within the function is performed successively until either the end of the tasks is reached or a non-FALSE predicate is evaluated. In the latter case evaluation proceeds as before except down the predicates task list. In either case, the value of the function is the value of the last task evaluated.

J. Sub-atomic Functions

1. FUNCTION COMPRESS (X),
    WHEN ATOM (X), "" EXIT,
    WHEN NAME (FIRST (X)),
        concatenate the print name of FIRST (X) onto the
        beginning of COMPRESS (REST (X)) then return the
        corresponding muSIMP name,
    COMPRESS (REST (X)),
ENDFUN;

Interpretation: The atom whose print name is the
packed version of the names in the list X.

2. FUNCTION EXPLODE (X),
    WHEN NAME (X),
        a list of names whose print names correspond
        to the characters in the print name of X, EXIT,
ENDFUN;

Interpretation: A list of the characters, in order,
in the print name of X.

3. FUNCTION LENGTH (X),
    WHEN NAME (X),
        WHEN EMPTY (X), 0 EXIT,
        the number of characters in the print name of X EXIT,
    WHEN INTEGER (X),
        the number of bytes in the vector of X EXIT,
    1 + LENGTH (REST (X)),
ENDFUN;

Interpretations:
a. The number of characters in the name X,
b. The number of bytes in the number X,
c. The number of top-level items in the list X.

K. Numerical Functions

1.
```
FUNCTION MINUS (X),
    WHEN INTEGER (X),
        -X  EXIT,
ENDFUN;
```

2.
```
FUNCTION PLUS (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X + Y  EXIT,
ENDFUN;
```

3.
```
FUNCTION DIFFERENCE (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X - Y  EXIT,
ENDFUN;
```

4.
```
FUNCTION TIMES (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X * Y  EXIT,
ENDFUN;
```

5.
```
FUNCTION QUOTIENT (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        WHEN Y = 0,  zero-divide error-trap  EXIT,
        WHEN POSITIVE (Y),  floor (X/Y)  EXIT,
        ceiling (X/Y)  EXIT,
ENDFUN;
```

Note: The integer quotient which is consistent with MOD being a periodic nonnegative remainder.

6.
```
FUNCTION MOD (X, Y),
    X - (Y * QUOTIENT(X,Y)),
ENDFUN;
```

7.
```
FUNCTION DIVIDE (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        ADJOIN (QUOTIENT (X, Y), MOD (X, Y))  EXIT,
ENDFUN;
```

Note: All muSIMP-79 numerical functions return FALSE if either of their arguments is non-numeric.

```
 8.  FUNCTION + (X, Y),
         PLUS (X, Y),
     ENDFUN;

     PROPERTY +, PREFIX, PARSE (SCAN, 130);

     PROPERTY +, RBP, 100;
     PROPERTY +, LBP, 100;


 9.  FUNCTION - (X, Y),
         WHEN EMPTY (Y), MINUS (EX1) EXIT,
         DIFFERENCE (X, Y),
     ENDFUN;

     PROPERTY PREFIX, -, LIST ('-, PARSE (SCAN, 130));

     PROPERTY RBP, -, 100;
     PROPERTY LBP, -, 100;


10.  FUNCTION * (X, Y),
         TIMES (X, Y),
     ENDFUN;

     PROPERTY RBP, *, 120;
     PROPERTY LBP, *, 120;


11.  FUNCTION / (X, Y),
         QUOTIENT (X, Y),
     ENDFUN;

     PROPERTY RBP, /, 120;
     PROPERTY LBP, /, 120;
```

L. Reader Functions

1. FUNCTION READCHAR (),
   read one character from the current input file and return the corresponding muSIMP atom. Integer atoms are returned only if the character is a decimal digit less than the current base.
   ENDFUN;

2. FUNCTION SCAN (),
   read one atom from the current input file and return the corresponding muSIMP atom. Atoms are delimited by either separator or break characters, however, the latter also are returned as atoms themselves.
   END;

   Separator characters: space, carriage return, line feed, and tab (control-I).

   Break characters: ! $ & ' ( ) * + , - . / @ : ; < = > ? [\ ] ^ _ ` { | } ~

3. FUNCTION READ (),
   read from the current input file one complete expression written in dotted-pair and/or list notation, then return the corresponding generated object. (Atoms are delimited by either separator or break characters, but the latter also are returned as atoms themselves).
   ENDFUN;

   Separator characters: space, comma, carriage return, line feed, and tab (control-I).

   Break characters: .) (

   Note: Extra right parentheses and dots are ignored.

4. FUNCTION PARSE (EX1, RBP, EX2),
   from the current input file, read the complete expression including the already-read token EX1, where RBP is the right binding power of the operator to the left of EX1, if any, then return the resulting unevaluated object. (EX2 is a local variable which accumulates the parsed representation.)
   ENDFUN;

Note: When two operators compete for an operand between them, the operator with higher binding power towards the operand acquires the operand. In the case of a tie, the operator on the left acquires the operand.

```
5.  FUNCTION SYNTAX X,
        print " *** SYNTAX ERROR:  ", then print each element
        in the list of arguments, X, then read up through the
        next terminator, echoing the input beginning on a new
        line, then return FALSE,
    ENDFUN;


6.  FUNCTION MATCH (DELIM),
        % Uses the fluid variable SCAN set by SCAN () %
        WHEN SCAN = DELIM,  SCAN (),  FALSE  EXIT,
        WHEN SCAN = comma,  SCAN (),  MATCH (DELIM)  EXIT,
        WHEN DELIMITER (),  SYNTAX (DELIM, "NOT FOUND")  EXIT,
        ADJOIN (PARSE(SCAN,0), MATCH(DELIM)),
    ENDFUN;


7.  DELIMITER: '(EXIT, ENDFUN, ENDLOOP, ENDBLOCK,
        right parenthesis, comma) &

    FUNCTION DELIMITER (),
        TERMINATOR ()  OR  MEMBER (SCAN, DELIMITER),
    ENDFUN;


8.  FUNCTION TERMINATOR (),
        SCAN = ';  OR  SCAN = '$  OR  SCAN = '&,
    ENDFUN;


10. ECHO: FALSE;

    FUNCTION ECHO (),
        NOT RDS  OR ECHO,
    ENDFUN;
```

Interpretation: ECHO () is a function which returns TRUE if input is being echoed to the terminal.

```
11. RDS: FALSE;                    % Device ReaD Select %

    FUNCTION RDS (X, Y, Z),
        WHEN EMPTY (X),  RDS: FALSE  EXIT,
        WHEN NAME (X) AND NAME (Y),
            WHEN EMPTY (Z),
                if there exists a file named X.Y on the
                currently logged disk drive, then open
                that file and RDS: X,  EXIT,
            WHEN NAME (Z),
```

```
                    if there exists a file named X.Y on drive Z,
                    then open that file and RDS: X,  EXIT  EXIT,
ENDFUN;
```

Notes:

1. Normally control of the current input file is done through the use of the function RDS as described above. However, after a file has been opened and made current, control can be returned to the console without closing the input file, simply by setting the value of RDS to FALSE. A subsequent non-FALSE assignment to RDS will then return control to the point in the opened disk file at which reading was suspended.

2. If the console is the current input file and all the characters have been read from the current line, the operating system's line-edit routine is called for further input. Thus the system's normal line-edit features will be used until a carriage return is typed, at which point muSIMP will regain control.

3. If a disk file is the current input file and the EOF (end-of-file) character is read, an error message is sent to the console, the console is made the current input file, and an error-options trap occurs.

4. If a disk file is being read and the value of the name ECHO is non-FALSE, the characters being read are also echoed to the current output file.

5. Comments in an input source file must be delimited by matching percent signs. The text of the comment will then be ignored by the functions SCAN and READ, except to possiblly echo the comment as described in note 4 above.

6. Special characters such as the comment, separator, and break characters can be read in as names or parts of names by means of quoted strings. Such strings are delimited by double quote marks. The double quote can be included within the string by using two adjacent double quotes for each desired internal double quote.

7. As an added programming convenience the muSIMP name SCAN is always set to the most recently read atom.

8. Lower-case letters are legitimate and distinct from their upper-case counterparts. The only exception to this is file names and types given to the functions RDS and WRS. They are always converted to upper case in order to eliminate conflicts with the operating system's file naming convention.

M. Printer Functions

```
1.  FUNCTION PRINT (X),
        WHEN NAME (X),
            output the print name of X to the current
            output file,  EXIT,
        WHEN INTEGER (X),
            output to the current output file the digits of X
            expressed in the current base, preceeded by a
            minus sign if X is negative,  EXIT,
        PRINT (LPAR),
        PRINLIST (X),
        X,
    ENDFUN;

    FUNCTION PRINLIST (X),
        PRINT (FIRST (X)),
        WHEN EMPTY (REST (X)),  PRINT (RPAR)  EXIT,
        PRINT (" "),
        WHEN ATOM (REST (X)),
            PRINT (". "),
            PRINT (REST (X)),
            PRINT (RPAR)  EXIT,
        PRINLIST (REST (X)),
    ENDFUN;
```

Interpretation: Print the standard list notation of the object X to the current output file.

```
2.  FUNCTION NEWLINE (),
        output a carriage return and line feed to the
        current output file, then return FALSE,
```

Interpretation: Terminate the current output line.

```
3.  FUNCTION PRINTLINE (X),
        PRINT (X),
        NEWLINE (),
        X,
    ENDFUN;
```

Interpretation: Print the expression X, terminate the last line and return X.

```
4.  FUNCTION SPACES (X),
        WHEN  X > 0  AND  X < 256,
            PRINT (" "),
            SPACES (X-1) EXIT,
        return the current cursor position,
    ENDFUN;
```

Interpretation: Output X spaces to the current output file and return the resulting cursor position.

5. FUNCTION PRIMATH (EX1, RBP, LBP, PRTSPACE),
    Taking account of declared binding powers and any special print rules on the property list of PRIMATH, print a deparsed representation of EX1, assuming
    a) the operator to its left, if any, has right binding power RBP, the operator to its right, if any, has left binding power LBP,
    b) appropriate spaces are to be printed if PRTSPACE is nonFALSE.

```
ENDFUN;
```

Interpretation: PRIMATH (expr, RBP, LBP) is a function which prints expr in standard mathematical form and surrounds it within parenthesis if the leading operator in expr has a left binding power less than or equal to RBP, or a right binding power less than LBP.

6. WRS: FALSE;

```
FUNCTION WRS (X, Y, Z),                    (Write Select)
    WHEN NOT EMPTY (WRS),
        write out the final record of WRS and
        close the file,
        WRS: FALSE,
        WRS (X, Y, Z)  EXIT,
    WHEN EMPTY (X),  WRS: FALSE  EXIT,
    WHEN NAME (X) AND NAME (Y),
        WHEN EMPTY (Z),
            on the currently logged disk drive,
            delete any previous file named X.Y and
            make a new directory entry for X.Y,
            WRS: X  EXIT,
        WHEN NAME (Z),
            on drive Z, delete any previous file
            named X.Y and make a new directory entry
            for X.Y,
            WRS: X  EXIT  EXIT,
ENDFUN;
```

7.
```
FUNCTION LINELENGTH (X),
    WHEN  X > 11  AND  X < 256,
        set maximum line-length to X,
        return the previous line-length  EXIT,
    return the current line-length,
ENDFUN;
```

Interpretation: Set the length at which output lines will automatically be terminated. The line-length is initially set to 72.

8. FUNCTION RADIX (X),
    WHEN X > 1 AND X < 37,
        set base to X,
        return the old base EXIT,
    return the current radix base,
ENDFUN;
Interpretation: Set the base in which numbers are expressed for both input and output. The base is initially set to ten.

Notes:

1. Normally control of the current output file is done through the use of the function WRS as described above. However, after a file has been opened for output, output can be directed to the console without closing the disk file by simply setting the value of WRS to FALSE. A subsequent non-FALSE assignment to WRS will then redirect output to the disk file and append data onto the end of the file.

2. If there is insufficient disk space or a hardware write error prevents correctly writing output onto the disk, an error message is sent to the console, the console is made the current output file, and an error-options trap occurs.

N. Evaluation Functions

1. PROPERTY PREFIX, ', LIST (READLIST (SCAN)) &

```
SUBROUTINE ' (X),
    X,
ENDSUB;
```

Interpretation: Suppress evaluation and return the object X itself.

2.
```
FUNCTION EVAL (X),
    WHEN ATOM (X), FIRST (X) EXIT,
    WHEN NAME (FIRST (X)),
        WHEN UNDEFINED (GETF (FIRST (X))),
            WHEN EQ (FIRST (X), EVAL (FIRST (X))),
                EVLIS (X) EXIT,
            EVAL (ADJOIN(EVAL(FIRST(X)), REST(X))) EXIT,
        WHEN FUNCTIONP (GETF (FIRST (X))),
            APPLY (FIRST (X), EVLIS (REST (X))) EXIT,
        WHEN SUBROUTINEP (GETF (FIRST (X))),
            APPLY (FIRST (X), REST (X)) EXIT,
        EVLIS (X) EXIT,
    WHEN FUNCTIONP (FIRST (X)),
        APPLY (FIRST (X), EVLIS (REST (X))) EXIT,
    WHEN SUBROUTINEP (FIRST (X)),
        APPLY (FIRST (X), REST (X)) EXIT,
    EVLIS (X),
ENDFUN;
```

Interpretation: Evaluate the object X.

```
FUNCTION EVLIS (X),
    WHEN ATOM (X), FALSE EXIT,
    ADJOIN (EVAL (FIRST (X)), EVLIS (REST (X))),
ENDFUN;

FUNCTION GETF (X),
    contents of the function cell of the name X,
ENDFUN;

FUNCTION UNDEFINED (X),
    return FALSE if X is a pointer to the undefined
    function trap, TRUE otherwise,
ENDFUN;
```

Interpretation: The recognizer for undefined functions.

```
FUNCTION FUNCTIONP (X),
    SUBR (X) OR EXPR (X),
ENDFUN;
```

Interpretation: The recognizer for call by value functions.

```
FUNCTION SUBROUTINEP (X),
    FSUBR (X) OR FEXPR (X),
ENDFUN;
```

Interpretation: The recognizer for call by name functions.

```
FUNCTION SUBR (X),
    return TRUE if X is a pointer to a FUNCTION subroutine,
    FALSE otherwise,
ENDFUN;

FUNCTION FSUBR (X),
    return TRUE if X points to a SUBROUTINE subroutine,
    FALSE otherwise,
ENDFUN;

FUNCTION EXPR (X),
    FIRST(X) = 'EXPR,
ENDFUN;

FUNCTION FEXPR (X),
    FIRST(X) = 'FEXPR,
ENDFUN;
```

3.
```
FUNCTION APPLY (X, Y),
    WHEN NAME (X),
        WHEN UNDEFINED (GETF (X)),
            WHEN X = EVAL(X), FALSE EXIT,
            EVAL (ADJOIN (EVAL (X), Y)),
        WHEN SUBR (GETF (X)),
            WHEN ATOM (Y), X (Y, FALSE, FALSE) EXIT,
            WHEN ATOM (REST (Y)),
                X (FIRST (Y), REST (Y), FALSE) EXIT,
            WHEN ATOM (RREST (Y)),
                X (FIRST(X), SECOND(X), RREST(Y)) EXIT,
            X (FIRST (Y), SECOND (Y), THIRD (Y)) EXIT,
        WHEN FSUBR (GETF (X)), X (Y) EXIT,
        WHEN EXPR (GETF (X)) OR FEXPR (GETF (X)),
            BIND (SECOND (GETF (X)), Y),
            Y: EVALBODY (FALSE, RREST (GETF (X))),
            UNBIND (SECOND (GETF (X))),
            Y EXIT EXIT,
    WHEN EXPR (X) OR FEXPR (X),
        BIND (SECOND (X), Y),
        Y: EVALBODY (FALSE, RREST (X)),
        UNBIND (SECOND (X)),
        Y EXIT,
ENDFUN;
```

Interpretation: Apply the function X to the list of arguments Y.

```
FUNCTION EVALBODY (X, Y),
    WHEN ATOM (Y), X EXIT,
    WHEN ATOM (FIRST (Y)) OR ATOM (FIRST (FIRST (Y))),
        EVALBODY (EVAL (FIRST (Y)), REST (Y)) EXIT,
    WHEN ATOM (FIRST (FIRST (FIRST (Y)))),
        X: EVAL (FIRST (FIRST (Y))),
        WHEN NOT X, EVALBODY (X, REST (Y)) EXIT,
        EVALBODY (X, REST (FIRST (Y))) EXIT,
    EVALBODY (EVALBODY (X, FIRST (Y)), REST (Y)),
ENDFUN;

FUNCTION BIND (X, Y),
    WHEN ATOM (Y),
        WHEN ATOM (X),
            WHEN EMPTY (X), FALSE EXIT,
            ARGSTACK: ADJOIN (EVAL (X), ARGSTACK),
            ASSIGN (X, FALSE) EXIT,
        ARGSTACK: ADJOIN (EVAL (FIRST (X)), ARGSTACK),
        ASSIGN (FIRST (X), FALSE),
        BIND (REST (X), Y) EXIT,
    WHEN ATOM (X),
        WHEN EMPTY (X), FALSE EXIT,
        ARGSTACK: ADJOIN (EVAL (X), ARGSTACK),
        ASSIGN (X, Y) EXIT,
    ARGSTACK: ADJOIN (EVAL (FIRST (X)), ARGSTACK),
    ASSIGN (FIRST (X), FIRST (Y)),
    BIND (REST (X), REST (Y)),
ENDFUN;

FUNCTION UNBIND (X),
    WHEN ATOM (X),
        WHEN EMPTY (X), FALSE EXIT,
        ASSIGN (X, FIRST (ARGSTACK)),
        ARGSTACK: REST (ARGSTACK) EXIT,
    UNBIND (REST (X)),
    ASSIGN (FIRST (X), FIRST (ARGSTACK)),
    ARGSTACK: REST (ARGSTACK),
ENDFUN;
```

4.
```
SUBROUTINE COND (X1, X2, ..., Xn),
    EVALCOND (LIST (X1, X2, ..., Xn)),
ENDSUB;

FUNCTION EVALCOND (X, Y),
    WHEN ATOM (X), FALSE EXIT,
    Y: EVAL (FIRST (FIRST (X))),
    WHEN NOT Y, EVALCOND (REST (X)) EXIT,
    EVALBODY (Y, REST (FIRST (X))),
ENDFUN;
```

Interpretation: Successively evaluate the FIRST of X1, X2, ..., Xn until either a non-FALSE value is encountered or all have evaluated to FALSE. In the former case the REST of that argument is evaluated

as a function body (see the interpretation of APPLY for details). In the latter case FALSE is returned by COND.

5. PROPERTY PREFIX, LOOP, ADJOIN ('LOOP, MATCH(ENDLOOP));

```
SUBROUTINE LOOP (X1, X2, ..., Xn),
    EVALLOOP (LIST (X1, X2, ..., Xn),
        LIST (X1, X2, ..., Xn)),
ENDSUB;

FUNCTION EVALLOOP (X, Y, Z),
    WHEN ATOM (Y), EVALLOOP (X, X) EXIT,
    WHEN ATOM (FIRST (Y)) OR ATOM (FIRST (FIRST (Y))),
        EVAL (FIRST (Y)),
        EVALLOOP (X, REST (Y)) EXIT,
    WHEN ATOM (FIRST (FIRST (FIRST (Y)))),
        Z: EVAL (FIRST (FIRST (Y))),
        WHEN NOT Z, EVALLOOP (X, REST (Y)) EXIT,
        EVALBODY (Z, REST (FIRST (Y))) EXIT,
    EVALBODY (FALSE, FIRST (Y)),
    EVALLOOP (X, REST (Y)),
ENDFUN;
```

Interpretation: The LOOP construct evaluates its argument in a manner identical to the evaluation of the clauses in a function body. However, if all the arguments are evaluated without a conditional having been satisfied, evaluation begins again with the first argument.

LOOP is the leading keyword of a control construct having the form:

```
LOOP
  task1,
  task2,
  ...
  taskn
ENDLOOP
```

This construct parses to the internal representation (LOOP task1 task2 ...). Usually at least one of the tasks is a conditional exit. Evaluation repetitively cycles through the sequence of tasks until a conditional exit causes control to proceed directly to the point following the matching delimiter ENDLOOP. The value of a LOOP construct is that of the last task evaluated therein. Since the LOOP construct parses to a function invocation, this construct can be used outside function definitions.

6. PROPERTY PREFIX, WHEN, MATCH (EXIT);

WHEN is the leading keyword of the conditional-exit control construct, which has the general form

WHEN expression1, expression2, ... EXIT

This construct parses to the internal representation

((expression1 expression2 ...)

If expression1 evaluates to FALSE, then evaluation proceeds directly to the point immediately following the matching EXIT. Otherwise, the expressions between expression1 and the matching EXIT, if any, are successively evaluated, and the last evaluated, after which evaluation proceeds to the point immediately following the next delimiter ENDLOOP, ENDBLOCK, ENDFUN, or ENDSUB.

7. PROPERTY PREFIX, BLOCK, MATCH (ENDBLOCK);

BLOCK is the leading keyword for the control construct of the form:

```
BLOCK
  WHEN ... EXIT,
  ...,
ENDBLOCK,
```

As indicated, the first task within a block must be a conditional exit. Since other tasks within the block can also be conditional exits, blocks provide a generalization of the "case" construct of some other languages, which includes the "if-then-else" construct as a special instance. The evaluation of tasks within a block proceeds sequentially unless a conditional exit therein causes evaluation to proceed directly to the point following the matching delimiter ENDBLOCK. The value of a block is that of the last expression evaluated therein.

```
8.  FUNCTION DRIVER (EX1, EX2),
        RDS: EX1,
        WRS: FALSE,
        NEWLINE (),
        NEWLINE (),
        LOOP
            ERR: FALSE,
            BLOCK
                WHEN ECHO (),
                    PRINT ("? "),
                    WHEN NOT RDS,  PRINT ("")  EXIT EXIT,
            ENDBLOCK,
            EX1: FALSE,
```

```
            EX1: PARSE (SCAN(), 0),
            EX2: SCAN,
            BLOCK
                WHEN ECHO (), NEWLINE () EXIT,
            ENDBLOCK,
            BLOCK
                WHEN ERR OR NOT TERMINATOR (),
                    SYNTAX (),
                    NEWLINE () EXIT,
                WHEN EX2 = '$,
                    #ANS: EVAL (EX1),
                    WHEN ECHO (), NEWLINE () EXIT EXIT,
                PRINT (@),
                #ANS: EVAL (EX1),
                PRINT (" "),
                WHEN EX2 = ';,
                    PRIMATH ("ANS, 0, 0, TRUE),
                    NEWLINE (), NEWLINE (), NEWLINE () EXIT,
                PRINTLINE (#ANS),
                NEWLINE (), NEWLINE (),
            ENDBLOCK,
        ENDLOOP,
    ENDFUN;
```

DRIVER is a function which controls the interaction cycle. After establishing the console as the current input and output file by setting RDS and WRS to FALSE respectively, the main read, evaluate, and print driver loop is entered. An expression is first read by PARSE. If the terminator was the character ";", the result is printed in mathematical notation by PRIMATH. If an "&", it is printed in List notation. And if a "$", it is not printed at all. However, in all cases it is assigned to the variable #ANS unless an error occurred during the parse phase in which case an error message is displayed. For some applications, it may be desirable to (perhaps dynamically) replace this driver with another one or to recursively call DRIVER.

P. Storage Functions

1. FUNCTION RECLAIM (),
reclaim all un-referenced nodes by generating a free node list from them, and compact the atom space and vector space. The total resulting number of free nodes is returned.
ENDFUN;

2. CONDENSE: FALSE;

```
FUNCTION CONDENSE (X, Y, Z),
    WHEN ATOM (FIRST (X)),
        WHEN ATOM (REST (X)), FALSE EXIT,
        Z: SUBEXPN (REST (X), Y),
        WHEN EMPTY (Z), CONDENSE (REST (X), Y) EXIT,
        REPLACER (X, Z),
        FALSE EXIT,
    Z: SUBEXPN (FIRST (X), Y),
    WHEN EMPTY (Z),
        CONDENSE (FIRST (X), Y),
        WHEN ATOM (REST (X)), FALSE EXIT,
        Y: ADJOIN (FIRST (X), Y),
        Z: SUBEXPN (REST (X), Y),
        WHEN EMPTY (Z), CONDENSE (REST (X), Y) EXIT,
        REPLACER (X, Z),
        FALSE EXIT,
    REPLACEF (X, Z),
    WHEN ATOM (REST (X)), FALSE EXIT,
    Z: SUBEXPN (REST (X), Y),
    WHEN EMPTY (Z), CONDENSE (REST (X) Y) EXIT,
    REPLACER (X, Z),
    FALSE,
ENDFUN;

FUNCTION SUBEXPN (X, Y, Z),
    WHEN COMPARE (X, Y),
       WHEN X = Y, Y EXIT EXIT,
    Z: SUBEXPN (X, FIRST (Y)),
    WHEN EMPTY (Z), SUBEXPN (X, REST (Y)) EXIT,
    Z,
ENDFUN;

FUNCTION COMPARE (X, Y),
    WHEN ATOM (Y), EXIT,
    WHEN ATOM (X), FALSE EXIT,
    WHEN COMPARE (FIRST (X), FIRST (Y)),
        COMPARE (REST (X), REST (Y)) EXIT EXIT,
ENDFUN;
```

Q. System Functions

1.
```
FUNCTION SAVE (X, Y),
    WHEN NOT EMPTY (WRS),
        write out the final record of WRS and
        close the file,
        WRS: FALSE,
        SAVE (X, Y)  EXIT,
    WHEN NAME (X)  AND  NAME (Y),
        WHEN EMPTY (Y),
            save a binary memory image of the current
            muSIMP system as a file named X of
            type "SYS" on the current drive,
            TRUE  EXIT,
        save a binary memory image of the current muSIMP
        system as a file named X of type "SYS" on
        drive Y,
        TRUE  EXIT
ENDFUN;
```

Note: SYS files occupy about 15 kilobytes less than the memory size for which the operating system is generated.

2.
```
FUNCTION LOAD (X, Y),
    WHEN NAME (X)  AND  NAME (Y),
        WHEN EMPTY (Y),
            load a memory image file named X of type "SYS"
            from the current disk,
            return control to the executive DRIVER loop EXIT,
        load a memory image file named X of type "SYS"
        from drive Y,
        return control to the executive DRIVER loop EXIT,
ENDFUN;
```

Interpretation: Restore the muSIMP environment present at the time of the SAVE.

V. The muSIMP79 PRATT Parser

A. Operators

1. INFIX is a name on whose property list is stored expressions specifying how to parse infix operators for which mere left and right binding powers do not suffice. It is used for the assignment operator ":" since a check is made on its left operand to make sure it is a name. Also the INFIX property is used for "(" to correctly parse function calls written using mathematical notation. The respective operator's left-hand operand is passed to the expression as the fluid name "EX1".

2. PREFIX is a name on whose property list is stored expressions specifying how to parse prefix operators for which mere left and right binding powers do not suffice. The matchfix operators, which include WHEN, LOOP, BLOCK, FUNCTION, SUBROUTINE, PROPERTY, and "(" when used to delimit a functions argument list, are examples of the use of the PREFIX property.

B. Binding Powers

1. LBP is a name on whose property list is the integer left binding powers of infix and postfix operators. When two operators are competing for an operand, the operator with higher binding power toward the operand obtains the operand. In case of a tie, the left operator obtains the operand, so that infix operators with the same left and right binding powers associate left, as is usually desired.

2. RBP is a name on whose property list is the integer right-binding powers of infix and prefix operators, for use as described for LBP.

C. Constants

1. COMMA is a global constant having the value ",". Because of conflicting parse properties associated with its use as a separator character, the name COMMA should be used for the literal ",".

2. LPAR is a global constant having the value "(". Because of conflicting parse properties associated with its use as a separator character, the name LPAR should be used for the literal "(".

3. RPAR is a global constant having the value ")". Because of conflicting parse properties associated with its use as a separator character, the name RPAR should be used for the literal ")".

D. Delimiters

1. DELIMITER is a name which is initialized to the list (EXIT, ENDLOOP, ENDBLOCK, ENDFUN, ENDSUB, RPAR, COMMA). When a matchfix operator is established, adjoining the matching delimiter to this list enables the parser to give more informative diagnostics by recognizing when a delimiter is used out of place. However, it is not necessary to adjoin delimiters to this list, and adjoining a delimiter has the effect of precluding its use out of context for other purposes.

2. DELIMITER () is a predicate which returns FALSE if the current value of the name SCAN is neither a terminator nor on the list named DELIMITER.

3. MATCH (delim) is a function which parses zero or more expressions separated by commas and delimited by the value of its argument. MATCH returns a list of the parsed representations of these expressions.

4. MATCHNOP (expr, delim) is a function used to verify that a matching "delim" was found following the PARSE of an expression within delimiters.

E. Parsing

1. PARSE (expr, rbp) is a function used to read a muMATH expression and convert it to List notation according to various rules established by operators LBP and RBP binding powers and/or PREFIX or INFIX property rules as described earlier.

Errors specific to PARSE include a member of DELIMITER "USED AS AN INDETERMINATE", an infix operator "USED AS AN PREFIX OPERATOR", and a prefix or postfix operator "USED AS AN INFIX OPERATOR".

2. SYNTAX exprs is a function which takes an arbitrary number of arguments. If the value of the Global variable ERR is FALSE, then the message "*** SYNTAX ERROR: " is printed followed by the arguments to SYNTAX separated by spaces. Unless input echoing from a file, the remainder of the expression is printed until a TERMINATOR character is reached. Finally, in order to return control to the console, the control variable RDS (i.e. ReaD Select described in Section IV. L.) is set to FALSE.

muSIMP-79 OPERATOR BINDING POWER TABLE

Category	Operator	LBP	RBP
Ordering	(	200	0
Assigament	:	180	20
Numerical	!	160	0
	^	140	139
	*	120	120
	/	120	120
	+	100	100
	-	100	100
Comparison	=	80	80
	<	80	80
	>	80	80
Logical	NOT	70	70
	AND	60	60
	OR	50	50

Note: When "+" and "-" are used as prefix operators a right binding power of 130 is used instead of 100.

MATRIX Package Documentation

PURPOSE:

File MATRIX.ARR provides the following matrix operations on arrays: transpose, multiplication, division, inverse, and other integer powers. Elementwise operations such as addition are provided by the prerequisite file ARRAY.ARI.

PREREQUISITE FILE: ARRAY.ARI

USAGE:

```
IDMAT (positiveinteger),
array   ,
array1 . array2,
array1 \ array2,
array ^ integer
```

EXAMPLES:

```
IDMAT(2)  -->  {[1],
                [0, 1]},

If  A = {[1, 2],     and  B = {P,     then:
         [0, 3]}              6}

    B` --> [P, 6],

    A` --> [{1,
             2}, {0,
                  3}],

    A` . IDMAT(2)  -->  {[1, 0],
                         [2, 3]},

    A . B  -->  {P+12,
                 18},

    A \ B  -->  {P-4,
                 2},

    A ^ 2  -->  {[1, 8],
                 [0, 9]},

    A ^ -1  -->  {[1, -2/3],
                  [0, 1/3]}
```

REMARKS:

1. The function named IDMAT returns a (left-triangular) identity matrix with the number of rows indicated by its positive integer argument.

2. The postfix operator named `, having a left binding power of 160, the same as "!", requests the transpose of its operand. (This "backward accent" character, ASCII code 60 hex, different from an apostrophe or single quote, is usually found on the same key as the character "@".) The transpose of a scalar is a scalar, the transpose of a row is the column of the transposes of its elements, and the transpose of a column is the row of the transposes of its elements. These rules are recursively employed so that the transpose of a ragged and/or nested matrix is appropriately performed. These rules also convert a column of rows into a row of columns, which does not print attractively. However, multiplication by an appropriate sized identity matrix always yields the attractive column-of-rows form of a matrix.

3. The matrix-product infix operator designated by a period has left and right binding powers 120, the same as for "*". The interpretations are:

```
scalar1 . scalar2  -->  scalar1 * scalar2,

scalar . array  -->  scalar * array,

array . scalar  -->  array * scalar,

row . col  -->  row .col  + row .col  + ...,
                   1  1       2  2

col . row  -->  {[col .row , col .row , ...],
                     1  1     1  2
                 [col .row , col .row , ...],
                    2  1     2  2
                  ...                    ]},

rowA . rowB  --> [[rowA .rowB , rowA .rowB , ...],
                      1  1      1  2
                  [rowA .rowB , rowA .rowB , ...],
                     2  1      2  2
                   ...                       ]],

colA . colB  --> {{colA .colB ,
                       1  1
                   colA .colB ,
                      1  2
                          ...},
                  {colA .colB ,
                      2  1
                   colA .colB ,
                      2  2
                          ...}}
```

Consistent with the interpretation described in ARRAY.DOC, when a row and column are of unequal length, the shorter is treated as

having implied trailing zero elements when forming "row . col" . These interpretations of matrix product are recursively employed so that matrix products of nested and/or ragged arrays are appropriatedly performed.

4. For a matrix A:

```
A ^ 0   —> IDMAT (LENGTH(A)-1),
A ^ 1   —> A,
A ^ -1  —> A inverse,
```

For integer n > 1:

```
A ^ n   —> A . (A ^ (n-1)),
A ^ -n  —> (A ^ -1) . (A ^ (n+1)).
```

When a matrix is singular, raising it to a negative power yields warning messages about divisions by zero, and the offending subexpressions are encapsulated in a question-mark form according to the usual muMATH-79 computational error treatment.

5. When a matrix A is square and nonsingular, then A\B is equivalent to (A ^ -1) . B. However, WE STRONGLY RECOMMEND using A\B unless the inverse is of independent interest or must be used many separate times, because A\B is more efficient and because, provided B is consistent, A\B will yield a parameterized solution even when A is singular. In this case, the parameters are designated by the forms ARB(1), ARB(2), ..., starting with 1 when file MATRIX.ARR or SOLVE.EQN is most recently loaded.

6. Comments in file MATRIX.ARR indicate how to save space by omitting the matrix transpose, division, or power packages.

Trace Package Documentation

PURPOSE:
File TRACE.MUS provides a trace package to help debug programs.

PREQUISITE FILE: MUSIMP79.COM

USAGE:
1. TRACE (name1, name2, ...),
2. UNTRACE (name1, name2, ...).

EXAMPLE:

```
FUNCTION MEMB (EX1, EX2),
  WHEN EMPTY (EX2), FALSE EXIT
  WHEN EX1 = FIRST (EX2), TRUE EXIT
  MEMB (EX1, REST (EX2))
ENDFUN;

TRACE (MEMB);
MEMB ('DOG, '(CAT, COW, DOG, PIG));

MEMB [DOG, (CAT, COW, DOG, PIG)]        %This is computer generated%
  MEMB [DOG, (COW, DOG, PIG)]
    MEMB [DOG, (DOG, PIG)]
    MEMB = TRUE
  MEMB = TRUE
MEMB = TRUE
@ TRUE

UNTRACE (MEMB);
```

REMARKS:

1. The trace of a function during the execution of a program provides an invaluable debugging tool.

2. Whenever a function is called it arguments are first evaluated and then printed following the function name.

3. After the function has been applied it's value is printed following the function name.

4. Indention is used to more easily pair corresponding calls and returns.

5. The function is restored to normal by UNTRACE.

Basic ALGEBRA Package Documentation

PURPOSE:

File ALGEBRA.ARI provides for the basic algebraic simplification of expressions using the elementary operators "+", "-", "*", "/", and "^". Simplifications may be categorized as either automatic or user controlled by means of CONTROL VARIABLES.

AUTOMATIC SIMPLIFICATIONS:

1. Rational arithmetic is used to combine numerical operands. (see ARITH.DOC for a complete description)

2. Identities and zeros are appropriately applied to expressions.
0+X —> X; 1*Y —> Y; 0*Z —> 0;

3. Sums and products are flattened and uniquely ordered to facilitate expression comparisons.
X+(Y+Z) —> X+Y+Z; Z*(Y*X) —> X*Y*Z;

4. Similar terms and products are combined.
3*X + 2*X —> 5*X; X^5 / X^2 —> X^3;

5. Powers of #I (i.e. the square root of -1) are reduced.
#I^7 —> -#I;

CONTROL VARIABLES:

The control variables described in this section enable the muMATH user to have complete control over the rules used to simplify an expression. However, they are rather difficult for the novice to master. Therefore the utility functions EXPAND, EXPD, and FCTR (described below) have been included in muMATH to make it easy to obtain the most common forms of an expression without the need to individually set control variables. We recommend these functions be used until more precise control of the control variables is required.

1. NUMNUM controls the distribution (factoring) of factors in the NUMerator of an expression over (from) a sum in the NUMerator.

Identity: A * (B+C) <—> A*B + A*C

2. DENDEN controls the distribution (factoring) of factors in the DENominator of an expression over (from) a sum in the DENominator.

Identity: $\frac{1}{A} * \frac{1}{B + C} \quad <\!\!\longrightarrow \quad \frac{1}{A*B + A*C}$

3. DENNUM controls the distribution (factoring) of factors in the DENominator of an expression over (from) a sum in the NUMerator.

$$\text{Identity:} \quad \frac{1}{A} * (B+C) \quad <\!\!\longrightarrow \quad \frac{B}{A} + \frac{C}{A}$$

4. NUMDEN controls the distribution (factoring) of factors in the NUMerator of an expression over (from) a sum in the DENominator.

$$\text{Identity:} \quad A * \frac{1}{B+C} \quad <\!\!\longrightarrow \quad \frac{1}{B/A + C/A}$$

5. BASEXP controls the distribution (factoring) of the BASe of an expression over (from) the EXPonent.

$$\text{Identity:} \quad A^{(B+C)} \quad <\!\!\longrightarrow \quad A^B * A^C$$

6. EXPBAS controls the distribution (factoring) of the EXPonent of an expression over (from) the BASe.

$$\text{Identity:} \quad (A*B)^C \quad <\!\!\longrightarrow \quad A^C * B^C$$

7. PWREXPD controls whether or not integer PoWeRs of sums are EXPanDed in numerators and/or denominators.

8. ZEROEXPT controls the use of the following identity which is valid for all A not equal to 0.

$$\text{Identity:} \quad A^0 \quad \longrightarrow \quad 1$$

9. ZEROBASE controls the use of the following identity which is only valid for positive A.

$$\text{Identity:} \quad 0^A \quad \longrightarrow \quad 0$$

USAGE:

1. For the first six of the above control variables, the kinds of factors, bases, or exponents which are distributed or factored from the expression can be precisely controlled by assigning appropriate values to the respective control variable. Positive integer values will cause distribution; whereas, negative values cause factoring. The exact type of expression which will be distributed or factored can be determined from the following table:

Prime	Type	Examples
2	Numerical expressions	4, -1/3, 5/7
3	Other non-sums	X, SIN(Y), Z^3
5	Sums	R+S, X^2-X, LN(X)+Z

Therefore, if a control variable is a multiple of one or more of the above primes, then that type of expression will be distributed or factored in accordance with that control variable's identity transform.

2. For example, since differences are internally represented as sums involving negative coefficients, evaluation of

```
3 * X * (1+X) * (1-X)  -->
        3 * X * (1+X) * (1-X)          if NUMNUM is 0,
        X * (3+3*X) * (1-X)            if NUMNUM is 2,
        3 * (X+X^2) * (1-X)            if NUMNUM is 3,
        3 * X * (1-X^2)                if NUMNUM is 5,
        (3*X+3*X^2) * (1-X)            if NUMNUM is 6,
        X * (3-3*X^2)                  if NUMNUM is 10,
        3 * (X-X^3)                    if NUMNUM is 15,
        3*X - 3*X^3                    if NUMNUM is 30.
```

3. As another example, if DENDEN is 15, then
Y / 3 / X / (1+X) / (1-X) --> Y / (3*(X-X^3)).

4. As another example, if DENNUM is 6, then
(X+3) / 3 / X --> 1/3 + 1/X.

5. When PWREXPD is a positive integer multiple of 2, then multinomial expansion occurs in numerators. When PWREXPD is a positive integer multiple of 3, then multinomial expansion occurs in denominators. Thus, when PWREXPD is 6,

```
(1+X)^3 / (1+X+Y)^2  -->
            (1+3*X+3*X^2+X^3) / (1+2*X+2*Y+2*X*Y+X^2+Y^2).
```

6. The importance of becoming thoroughly familar with the use of PWREXPD, NUMNUM, DENDEN, and DENNUM cannot be over-emphasized! muMATH-79 cannot read a user's mind, so these control variables are the major means of specifying which of the many alternative transformations are desired at each stage in a dialog.

7. The remaining control variables are of less frequent concern, but changing their settings is occasionally crucial to acheiving a desired effect. Since they follow the same general scheme, they are easy to use after the more important control variables have been mastered. For example,

```
(3+X) / (1+X)  -->
        1 / (3/(1+X) + X/(1+X))        if NUMDEN is 5,
        1 / (1/(1/3+X/3) + 1/(1/X+1))  if NUMDEN is 30.
```

Thus, this transformation yields a kind of "continued-fraction" expansion.

8. BASEXP is set in an analogous fasion as follows:

```
2 ^ (1+N)  -->  2 * 2^N
                if BASEXP is a positive integer multiple of 2,
X ^ (1+N)  -->  X * X^N
                if BASEXP is a positive integer multiple of 3,
(A+B) ^ (1+N)  -->  (A+B) * (A+B)^N
                if BASEXP is a positive integer multiple of 5.
```

The opposite of these transformations is more often appropriate, and is accomplished by setting BASEXP to be negative.

UTILITY FUNCTIONS:

1. EVAL (expr) returns the evaluated and simplified expression resulting from expr operated on under the current control variable environment.

2. SUB (expr1, expr2, expr3) returns the expression which results from SUBstituting all occurrences of expr2 by expr3 in expr1.

3. EVSUB (expr1, expr2, expr3) is defined as
EVAL (SUB (expr1, expr2, expr3)).

4. NUM (expr) returns the NUMerator of expr.

5. DEN (expr) returns the DENominator of expr.

6. FLAGS () prints the current value of the system control variable.

7. EXPAND (expr) evaluates expr to yield a fully expanded denominator distributed over the terms of a fully expanded numerator. The following temporary assignments are made:

```
PWREXPD: 6;  NUMDEN: 0;  NUMNUM: DENDEN: DENNUM: BASEXP: EXPBAS: 30;
```

8. EXPD (expr) evaluates expr to yield a fully expanded numerator over a fully expanded denominator. The following temporary assignments are made:

```
PWREXPD: 6; NUMDEN: 0; DENNUM: -30; NUMNUM: DENDEN: BASEXP: EXPBAS: 30;
```

9. FCTR (expr) evaluates expr to yield a semi-factored numerator over a semi-factored denominator. The following temporary assignments are made:

```
PWREXPD: NUMDEN: 0;  NUMNUM: DENDEN: -6;  DENNUM: BASEXP: EXPBAS: -30;
```

CONTROL VARIABLE SUMMARY:

Control Var.	Initial Value	Positive Transformation	Negative Transformation
NUMNUM	6	A*(B+C) ==> A*B + A*C	A*B + A*C ==> A*(B+C)
DENDEN	2	$\frac{1}{A} * \frac{1}{B+C} \Rightarrow \frac{1}{A*B + A*C}$	$\frac{1}{A*B + A*C} \Rightarrow \frac{1}{A} * \frac{1}{B+C}$
DENNUM	6	$\frac{B+C}{A} \Rightarrow \frac{B}{A} + \frac{C}{A}$	$\frac{B}{A} + \frac{C}{A} \Rightarrow \frac{B+C}{A}$
NUMDEN	0	$\frac{A}{B+C} \Rightarrow \frac{1}{B/A + C/A}$	$\frac{1}{B/A + C/A} \Rightarrow \frac{A}{B+C}$
BASEXP	-30	A^(B+C) ==> A^B*A^C	A^B*A^C ==> A^(B+C)
EXPBAS	30	(A*B)^C ==> A^C*B^C	A^C*B^C ==> (A*B)^C
PWREXPD	0	(A+B)^N ==> A^N+...+B^N	$(A+B)^{-N} \Rightarrow \frac{1}{A^N+...+B^N}$

Equation Package Documentation

PURPOSE:

File EQNALG provides a facility whereby equations are treated as expressions which can be assigned, added, multiplied, squared, etc.

PREREQUISITE FILE: ALGEBRA.ARI

USAGE: expression1 == expression2

EXAMPLES:

```
            EQN1:  5*X - 3*X - 7 == 2 + 4;  -->  2*X - 7 == 6
then        EQN1 + (7 == 7);                -->  2*X == 13
then        #ANS/2;                         -->  X == 13/2
```

REMARKS:

1. The two sides of the equation are independently simplified according to the current control settings. However, there is no attempt to automatically shift terms from one side to the other, etc. Moreover, there is no attempt to verify or disprove that the equation is an identity or has a solution.

2. This use of the == sign to indicate equations should not be confused with the use of = within the conditional EXIT construct in muSIMP function definitions. When used in this more active role the result is always either TRUE or FALSE depending upon whether or not the left and right sides have identical (as distinct from equal) values.

3. The left and right binding powers of == are 80, which is the same as for =.

4. As illustrated by the above example, when a non-equation is combined with an equation, the non-equation is independently combined with both sides.

5. Although the above example illustrates how equations can be solved stepwise, file SOLVE_EQN automates this process.

6. Provided file ARRAY.ARI is loaded, sets of simultaneous equations can be represented as an array of equations. For example:

```
[2*X == 6, 4*Y == 8] / 2;  -->  [X == 3, 2*Y == 4].
```

7. As with manual computation, operations such as squaring both sides or clearing non-numeric denominators can enlarge the solution set, so the user should exercise caution and verify candidate solutions generated by such means.

8. If FOO is an equation, then SECOND (FOO) returns the left-hand side of the equation, and THIRD (FOO) returns the right-hand side.

Solve Equation Package Documentation

PURPOSE:
File SOLVE.EQN provides a function for the exact solution of an algebraic equation.

PREREQUISITE FILE: EQN.DOC

USAGE: SOLVE (equation, unknown)

EXAMPLES:

```
SOLVE (X^2 == 4*A,  X);          -->  {X == 2*A^(1/2)
                                       X == -2*A^(1/2)}

SOLVE (LN(ATAN(X-1)) == B,  X);  -->  {X == 1 + TAN(#E^B)}
```

REMARKS:

1. SOLVE returns a column of solutions, where columns are as described in file ARRAY.DOC. The functions FIRST, REST, SECOND, and THIRD can be used to extract individual solutions from a column of solutions. Alternatively, subscripts can be used for this purpose provided file ARRAY.ARI is loaded.

2. Forgetting the second argument of SOLVE is a frequent mistake.

3. As a convenience, when either side of an equation is zero, the == 0 can be omitted.

4. When no solution exists, SOLVE returns the empty column, {}.

5. When degenerate equations have an entire locus of solutions which require parameterization to represent completely, SOLVE introduces the parameters,

ARB(1), ARB(2), ARB(3), ...

Their indexes start at 1 every time SOLVE.EQN or MATRIX.ARR is loaded. The following is an example of the muSIMP-79 solution to a degenerate equation:

SOLVE (X == X, X); --> {X == ARB(1)}.

6. SOLVE expands the difference in two sides of an equation over a common denominator, then multiplies by the denominator to clear it. This multiplication can introduce spurious solutions if a zero of the denominator coincides with one of the numerator. Similarly, this multiplication can suppress a solution associated with an infinity of the denominator. Thus, the returned set should be regarded as candidates for some of the solutions rather than the

complete verified solution set if an equation has a denominator which could be zero or infinite for finite values of the unknown. When these possibilities are present, it is the user's responsibility to verify his solutions by substitution or perhaps by taking limits. It may be helpful in such instances to also use SOLVE to find any zeros of the common denominator in order to see if they coincide with any of those in the numerator.

7. After clearing the denominator, SOLVE attempts moderate factorization, then independently attempts to determine the zeros of each resulting factor. SOLVE recursively employs appropriate formulas for the inverses of the elementary functions and for the zeros of linear, quadratic, and binomial factors. When SOLVE encounters a factor which it cannot treat, it returns a "solution" of the form "factor == 0". Since the factor may be simpler than the original equation, it might serve as a useful point of departure for an approximate numerical solution.

8. A careful study of the source listing for the file SOLVE.EQN reveals how additional inverse functions can be employed.

9. File MATRIX.ARR contains a matrix division operation which can be used to solve simultaneous linear algebraic equations.

Array Package Documentation

PURPOSE:

File ARRAY.ARI provides a facility for establishing generalized arrays, for extracting their components, and for performing elementwise operations between arrays or between arrays and scalars.

PREREQUISITE FILE: ARITH.MUS

USAGE:

1. Formation of a column vector:
   {expression1, expression2, ..., expressionN}.
2. Formation of a row vector:
   [expression1, expression2, ..., expressionM].
3. Extraction of components:
   array rowvector
4. Operations having forms such as:
   array1 operator array2,
   scalar operator array,
   functionname (array).

EXAMPLES:

```
[0, X] + [5, X, Y];       —>  [5, 2*X, Y]
2 * {X, LN(Y)};           —>  {2*X,
                               2*LN(Y)}
SIN ([X, Y]);             —>  [SIN(X), SIN(Y)]
[X, [Y,Z], [W]][2];       —>  [Y,Z]
[X, [Y,Z], [W]][2,1];     —>  Y
[X, [Y,Z], [W]][2][1];    —>  Y
```

REMARKS:

1. Arrays can be nested to any desired depth. The elements of a row or column can be any arbitrary expressions, including perhaps another row or column.

2. Columns are printed starting each element on a new line. Thus, 2-dimensional arrays generally look better as a column of rows than as a row of columns. Higher dimensional arrays generally appear best as a column of rows of rows . . . of rows.

3. When rows or columns of unequal length are combined elementwise by an arithmetic operation, the shorter of the two arrays is treated as having implied zeros corresponding to the extra elements of the longer array. (Consistent with this interpretation, a subscript value larger than the number of explicit elements in a row or column yields a zero as the value of the element.) Thus, upper-triangular, left-triangular, and other "ragged" arrays are efficiently represented.

4. When an array is combined with a scalar, the latter distributes over the elements of the array.

5. Functions of one argument such as SIN, ATAN, etc., which employ the general rule-application function named SIMPU, distribute over the elements of an array.

6. Subscripts can be recursively employed to any level, and they can be symbolic. For example, [Y, Z][2][N] --> Z[N].

7. FIRST(row) --> [, FIRST(column) --> {, SECOND(row or column) --> first element, etc.

8. Comments in file ARRAY.ARI indicate how to save space by omitting the column and/or subscript packages. (Rows together with FIRST, REST, SECOND, etc. are sufficient for many purposes.)

9. File MATRIX.ARR implements matrix operations on arrays, including matrix transpose, multiplication, division, and power, including inverse.

RATIONAL ARITHMETIC Package Documentation

PURPOSE:

File ARITH.MUS can perform exact rational arithmetic operations including sums, differences, products, quotients, and powers up to 611 digits of accuracy in any desired radix base.

PREREQUISITE FILE: MUSIMP79.COM

EXAMPLES:

```
5/9 + 7/12;                 % Exact rational arithmetic %
FOO: (236 - 3*127) * -13;   % Make assignments to variables %
FOO ^ 16;                   % Raise numbers to integer powers %
RADIX (2);                  % The radix base can be set from 2-36 %
FOO;                        % Convert numbers between radix bases %
1011000101 + 111010001;     % Do binary arithmetic %
RADIX (1010);               % To return to base 10 %
GCD (436, 582);             % Compute the GCD of two numbers %
```

CONTROL VARIABLES:

1. PBRCH is a control variable which, when TRUE, permits selection of a branch of a multiply-branched function. For ARITH.MUS, PBRCH nonFALSE permits the simplification

(expr1 ^ expr2) ^ expr3 —> expr1 ^ (expr2 * expr3)

even when expr3 is not an integer.

2. ZEROBAS is a control variable which, when TRUE, permits the simplification 0 ^ expr —> 1 even when expr is nonnumeric.

3. ZEROEXP is a control variable which, when TRUE, permits the simplification expr ^ 0 —> 1 even when expr is nonnumeric.

PRIMITIVLY DEFINED FUNCTIONS:

1. ABS (expr) is a function which returns the absolute value of its argument when the argument is a rational number. Otherwise, the rule-application function SIMPU is invoked, so the unevaluated absolute-value form is returned if no applicable rules are present.

2. ARGEX (expr) is a helper function used by SIMPU and elsewhere to appropriately partition an expression for application of a rule.

3. ARGLIST (expr) is a helper function used to appropriately group the operands of an expression for application of rules to varyary operators such as "+" and "*".

4. BASE (expr) is a selector function which returns the base of an expression of the form base ^ exp; otherwise it returns expr itself.

5. CODIV (expr) is a selector function which returns the codivisor (i.e. the non-numeric factors) of an expression which is a product; 1 if NUMBER (expr); otherwise it returns expr itself.

6. COEFF (expr) is a selector function which returns the coefficient (i.e. the numeric factors) of an expression which is a product; the expr if NUMBER (expr); otherwise it returns 1. Note that in all cases

expr = COEFF (expr) * CODIV (expr)

7. DEN (expr) is a selector function which returns the denominator of its argument, returning 1 when there is none.

8. DENOM (expr) is a recognizer function which returns TRUE iff its argument has the internal form (^ bas exp), with exp being negative or having a negative coefficient.

9. EVSUB (expr, subexpr, replacement) is a function which returns the result of evaluating a copy of its first argument, wherein each syntactic occurrence of its second argument is replaced by the third argument.

10. EXPON (expr) is a selector function which returns the exponent of an expression of the form base ^ exp; otherwise it returns 1. Note that in all cases

expr = BASE (expr) ^ EXPON (expr)

11. GCD (intgr1, intgr2) is a function which returns the positive greatest common divisor of its integer arguments.

12. IDENTITY (expr) returns its argument. This trivial function is used for applying inverses and accomodating conditional exits having atomic conditions.

13. LCM (intgr1, intgr2) is a function which returns the positive least common multiple of its integer arguments.

14. MIN (intgr1, intgr2) is a function which returns the minimum of its two integer arguments.

15. MULTIPLE (intgr1, intgr2) is a function which returns FALSE if its second integer argument is not an integer multiple of its first integer argument.

16. NEGCOEFF (expr) is a recognizer function which returns TRUE iff its argument is negative or has a negative coefficient, returning FALSE otherwise.

17. NEGMULT (intgr1, intgr2) is a predicate which returns TRUE iff its second integer argument is a negative integer multiple of its first integer argument.

18. NUM (expr) is a selector function which returns the numerator of its argument, returning the entire argument when there is no denominator.

19. NUMBER (expr) is a recognizer function which returns TRUE iff its argument is an integer or a rational number.

20. POSMULT (intgr1, intgr2) is a predicate which returns TRUE iff its first integer argument is a positive multiple of its second integer argument.

21. POWER (expr) is a recognizer function which returns TRUE iff its argument is of the form expr1 ^ expr2, returning FALSE otherwise.

22. PRODUCT (expr) is a recognizer function which returns TRUE iff its argument is of the form expr1 * expr2, returning FALSE otherwise. It is important to realize that quotients are represented as products involving negative powers.

23. RECIP (expr) is a recognizer function which returns TRUE iff its argument is a rational number of the form 1/d, returning FALSE otherwise.

24. SIMPU (name, expr) is a function which applies any appropriate established rules for the unary function or operator whose name is the first argument of SIMPU and whose operand is the second argument of SIMPU.

25. SUB (expr, subexpr, replacement) returns a copy of its first argument, wherein every syntactic instance of its second argument is replaced by its third argument. In general this will produce an unsimplified result, so the similar EVSUB function uses SUB, then EVAL.

26. SUM (expr) is a recognizer function which returns TRUE iff its argument is of the form expr1 + expr2, returning FALSE otherwise. It is important to realize that differences are represented as sums involving terms having negative coefficients.

Optional FRACTIONAL POWER Package

PURPOSE:
Provides the facilities for the simplification of fracticnal powers of numbers and complex exponentials.

USAGE:
number ^ (fraction),
#E ^ (intgr * #I * #PI / 2).

EXAMPLES:
(-24) ^ (1/3) -> -2 * 3 ^ (1/3),
(-4) ^ (1/2) -> #I * 2,
#E ^ (3 * #I * #PI / 2) -> - #I.

CONTROL VARIABLE:
PBRCH, which if FALSE, prevents Picking a BRanCH for fractional powers. (e.g. 4 ^ (1/2) will not simplify to 2.)

REMARKS:

1. #E represents the base of the natural logarithms, #I represents the positive square root of minus one (+ (-1)^(1/2)), and #PI represents the ratio of the circumference of a circle to its diameter.

2. Simplification of fractional powers takes place only if the control variable named PBRCH is not FALSE. The positive real branch is selected if one exists. Otherwise, the negative real branch is selected if one exists. Otherwise, the branch with smallest positive argument is selected.

3. As in manual computations, Picking a BRanCH of a fractional power involves an arbitrary choice which can yield invalid results. Thus, the user is cautioned to verify results obtained by such operations.

4. The global variable named PRIMES contains a list of successive primes, beginning with the integer 2. For fractional powers, the radicand is factored into a product of powers of the numbers in PRIMES, perhaps times a residual having no factors in PRIMES. The fractional power is then distributed over this product, with a discrete variant of Newton's method being used to determine if the fractional power of any residual is an integer. Thus, simplification of fractional powers of large integers might be incomplete if PRIMES is not long enough.

5. As in manual computations, reduction of complex exponentials modulo (2 * #PI * #I) is inconsistent with the identity LN(Z*W) = LN(Z) + LN(W). Thus, the user is cautioned to verify results obtained using both transformations together.

Optional FACTORIAL Package

PURPOSE:

Provides the factorial postfix operator "!". The factorial of a non-negative integer is recursively defined as follows:

$$0! = 1,$$
$$N! = N*(N-1)!, \quad \text{for } N > 0.$$

USAGE:

N! where N is a non-negative integer.

EXAMPLE:

5! --> 120.

REMARKS:

1. The left binding power of "!" is 160. Thus -5! parses to -(5!) and 3^5! parses to 3^(5!).

2. When not given a nonnegative integer operand, "!" calls upon the SIMPU rule-application function, thus returning the unevaluated factorial form if no appropriate rules are established.

Logarithm Package Documentation

PURPOSE:
File LOG.ALG provides for logarithmic simplifications.

PREREQUISITE FILE: ALGEBRA.ARI

CONTROL VARIABLES:

1. LOGBAS, which is the default LOGarithm BASe when LOG is given only one argument.

2. PBRCH, which if FALSE prevents Picking a BRanCH of logarithms.

3. LOGEXPD, which controls expansion or collection of logarithms, and base conversion.

USAGE:
LN (expr),
LOG (expr),
LOG (expr, base).

REMARKS:

1. Since the emphasis of muMATH is on exact results, there is no attempt to approximate irrational logarithms.

2. The unbound variable #E represents the base of the natural logarithms.

3. Although all logarithms are stored internally as two argument functions, LN (expr) is used as an abbreviation for LOG (expr, #E) on input and output.

4. LOG (expr) is used as an abbreviation for LOG (expr, LOGBAS) on input and output, where LOGBAS is a control variable initially set to #E.

5. base ^ LOG (expr, base) --> expr.

6. Provided PBRCH is TRUE:

LOG (1, base) --> 0,
LOG (base, base) --> 1,
LOG (base^expr, base) --> expr.

7. Provided LOGEXPD is a positive integer multiple of 2:

LOG (expr,base) —> LOG (expr, #E) / LOG (base, #E)

when base is not #E. When LOGEXPD is a negative integer multiple of 2, the opposite transformation of combining appropriate ratios of logarithms occurs.

8. Provided LOGEXPD is a positive integer multiple of 3:

LOG (expr^exp, base) —> exp * LOG (expr, base).

A negative integer multiple of 3 causes the opposite transformation.

9. Provided LOGEXPD is a positive integer multiple of 5:

LOG (expr1*expr2, base) —> LOG (expr1, base) + LOG (expr2, base),
LOG (expr1/expr2, base) —> LOG (expr1, base) - LOG (expr2, base).

A negative integer multiple of 5 causes the opposite collection transformation.

Positive TRIGONOMETRIC Simplification Package Documentation

PURPOSE: File TRGPOS.ALG provides the following trigonometric transformations:

1. exploitation of symmetry to simplify trig arguments
2. replacement of other trig functions by sines and cosines
3. replacement of integer powers of sines and cosines by linear combinations of sines and cosines of multiple angles
4. replacement of products of sines and cosines by linear combinations of sines and cosines of angle sums
5. replacement of integer powers of sines by those of cosines or vice-versa

PREREQUISITE FILES: ALGEBRA.ARI

(Note: Loading TRGNEG.ALG after TRGPOS.ALG preserves the full capabilities of both files. Loading TRGPOS.ALG after TRGNEG.ALG destroys the angle-reduction capabilities of the latter, thus saving some space.)

CONTROL VARIABLES:

1. TRGEXPD controls replacement of trig functions by sines and cosines and replacement of powers and products of sines and cosines by linear combinations. Only positive values of TRGEXPD are significant when TRGPOS.ALG is loaded without TRGNEG.ALG.

2. TRGSQ controls the conversion of integer powers of sines to cosines and vice-versa.

USAGE:

```
SIN (expression),
COS (expression),
TAN (expression),
CSC (expression),
SEC (expression),
COT (expression).
```

REMARKS:

1. SIN(0) —> 0, and COS(0) —> 1.

2. Symmetry is exploited to simplify the arguments of sines and cosines. For example, SIN(-X) —> -SIN(X) and COS(-X) —> COS(X).

3. When TRGEXPD is a positive multiple of 2, then tangents, cotangents, secants, and cosecants are replaced by corresponding expressions involving sines and/or cosines. For example, when TRGEXPD = 30, CSC(X) --> 1/SIN(X).

4. When TRGEXPD is a positive multiple of 3, then integer powers of sines and cosines are expanded in terms of sines and cosines of multiple angles. For example, when TRGEXPD = 30, COS(X)^2 --> (1+COS(2*X))/2. These transformations usually give the most attractive results if NUMNUM and perhaps also DENNUM are positive multiples of 6.

5. When TRGEXPD is a positive multiple of 5, then products of sines and cosines are expanded in terms of angle sums. For example, when TRGEXPD is 30, SIN(X)*SIN(Y) --> (COS(X-Y) - COS(X+Y))/2. These transformations usually give the most attractive results if NUMNUM is a positive multiple of 30 and DENNUM is a positive multiple of 2.

6. Expanding over a common denominator with TRGEXPD = 30 yields a normal form for a large class of trigonometric-rational expressions. Thus, the most straightforward way to prove most trig identities is to evaluate the difference in the two sides with TRGEXPD: NUMNUM: DENDEN: 30, PWREXPD: 6, and DENNUM: -30.

7. TRGEXPD = 30 has the effect of "linearizing" trigonometric polynomials, thus facilitating harmonic or Fourier analysis.

8. For integer n with $|n| > 1$ and for all u, when TRGSQ is a positive integer, then

COS(u)^n --> COS(u)^REMAINDER(n,2) * (1 - SIN(u)^QUOTIENT(n,2))^2.

Conversely, when TRGSQ is a negative integer, then

SIN(u)^n --> SIN(u)^REMAINDER(n,2) * (1 - COS(u)^QUOTIENT(n,2))^2.

These transformations are sometimes useful for transforming a trigonometric polynomial to a more compact equivalent trig polynomial.

9. Even when a trig polynomial is preferred for the final form, net simplification is often achieved by evaluating with TRGEXPD = 30, then -30, then perhaps again with TRGSQ = 1 or -1 according to the appearance of the result produced by -30.

10. File TRGNEG.ALG provides for the negative settings of TRGEXPD to yield the converse of the above transformations.

Negative TRIGONOMETRIC Simplification Package Documentation

PURPOSE: File TRGNEG.ALG provides the following trigonometric transformations:

1. exploitation of symmetries to simplify trig arguments
2. angle reduction
3. multiple-angle expansion
4. angle-sum expansion
5. elimination of reciprocals of trigonometric forms
6. elimination of certain products of trigonometric forms
7. simplification of trig functions of their own inverses
8. replacement of sines and cosines by complex exponentials

PREREQUISITE FILE: ALGEBRA.ARI

(Note: Loading TRGPOS.ALG after TRGNEG.ALG destroys the angle-reduction capabilities of the latter, thus saving some space. Loading TRGNEG.ALG after TRGPOS.ALG preserves the full capabilities of both files.)

CONTROL VARIABLES:

1. TRGEXPD controls the use of multiple angle and angle sum expansions and replacement of trig functions by complex exponentials. Only negative values of TRGEXPD are significant when TRGNEG.ALG is loaded without TRGPOS.ALG.

USAGE:

```
SIN (expression),
COS (expression),
TAN (expression),
CSC (expression),
SEC (expression),
COT (expression),
TRGEXPD (expression, integer).
```

REMARKS:

1. Since the emphasis of muMATH-79 is on exact results, there is no attempt to approximate irrational trig expressions.

2. The ratio of the circumference to the diameter of a circle is represented by the unbound variable #PI. The user is of course free to assign a rational approximation to #PI and use series approximations to the trig functions.

3. Angles are assumed to be measured in radians. Those who would prefer some other unit such as degrees may wish to define additional functions named SIND, COSD, etc.

4. Sines and cosines of angles which are numeric multiples of #PI are reduced to equivalent sines or cosines in the range [0, #PI/4], then sines and cosines of the special angles 0, #PI/6, and #PI/4 are evaluated exactly. For example,

SIN (20*#PI/7) --> SIN (#PI/7),
and SIN (7*#PI/3) --> 3^(1/2)/2.

5. Symmetry is exploited to simplify the arguments of sines and cosines. For example,

SIN (-X) --> -SIN (X),
and COS (-X) --> COS (X).

6. Trigonometric functions of the corresponding inverse trig functions simplify. For example, SIN(ASIN(X+5)) --> X+5. The inverse trig functions are named ATAN, ASIN, ACOS, ACOT, ACSC, and ASEC.

7. Products of a tangent, cotangent, secant, or cosecant with another trig function of the same argument are simplified to 1 or to a single form where possible. For example,

SEC(X)*COS(X) --> 1,
and TAN(X)*COS(X) --> SIN(X).

For an expression such as SEC(X)^2*COS(X)^2 it is necessary to reevaluate with EXPBAS being a negative multiple of 2 in order to acheive the desired trig transformation.

8. When TRGEXPD is a negative multiple of 2, then negative powers of tangents, cotangents, secants, and cosecants are replaced by corresponding positive powers of the corresponding reciprocal trig functions. For example, when TRGEXPD = -6, 1/TAN(X+7)^3 --> COT(X+7)^3. For technical reasons, negative powers of sines and cosines are treated in file TRGPOS.ALG.

9. When TRGEXPD is a negative multiple of 3, then sines and cosines of mulitple angles are expanded in terms of sines and cosines of non-multiple angles. For example, when TRGEXPD = -6,

SIN (2*X) --> 2*SIN(X)*COS(X)
and COS (3*X) --> 4*COS(X)^3 - 3*COS(X).

These transformations usually give the most attractive results if NUMNUM is a positive multiple of 6.

10. When TRGEXPD is a negative multiple of 5, then sines and cosines of angle sums and differences are expanded in terms of sines and cosines of nonsums and nondifferences. For example, when TRGEXPD=-15, COS(X+Y) --> COS(X)*COS(Y) - SIN(X)*SIN(Y). These transformations usually give the most attractive results if NUMNUM is a positive multiple of 6.

11. When TRGEXPD is a POSITIVE multiple of 7, then sines and cosines are converted to complex exponentials. For example, when TRGEXPD = 14, then COS(X) --> (#E^(#I*X) + 1/#E^(#I*X)) / 2. The opposite transformation, provided in file ARITH.MUS, is requested when TRGEXPD is a negative multiple of 7. A worthwhile net trig simplification can sometimes be achieved by converting to complex exponentials, expanding or factoring judiciously, then converting back to trig functions.

12. In muMATH-79 changing the value of an option variable does not affect the values of expressions which have already been evaluated. Thus, after changing the value of TRGEXPD and other relevant variables it may be necessary to use EVAL to get the desired effect.

13. Function TRGEXPD reevaluates its first argument with TRGEXPD temporarily set to the value of the second argument. Thus, it provides a convenient way to accomplish a trigonometric transformation without the necessity of altering the global setting of the TRGEXPD control variable.

14. File TRGPOS.ALG has other important trig transformations, many of which are the opposite of those provided in file TRGNEG.ALG. Generally, the positive settings yield a more canonical (but not necessarily more compact) representation. A net simplification is often achieved by evaluating an expression with the relevant option variables set positive, then reevaluating with them set the other way. Thus, files TRGPOS.ALG and TRGNEG.ALG comprise an important complementary pair pair of files. Since together the files are relatively large, for some applications it may be desirable to extract and combine a few of the required features from both files, together perhaps with a few additional transformations modeled after them.

DIFFERENTIATION Package Documentation

PURPOSE:
File DIF.ALG provides a function which returns the symbolic first partial derivative of its first argument with respect to its second argument.

PREREQUISITE FILE: ALGEBRA.ARI

USAGE:
DIF (expression, variable),

EXAMPLES:
DIF (A*X^2, X) —> 2*A*X,
DIF (LN(X+A), X) —> 1/(X+A).

REMARKS:

1. When the differentiation rule for a function or operator is not known to the system:

a. The derivative is 0 if none of the arguments or operands contain the differentiation variable. For example,

DIF (F(Y), X) —> 0.

b. The derivative is not evaluated otherwise. For example,

DIF (F(X), X) —> DIF (F(X), X).

2. A careful study of file DIF.ALG reveals how additional differentiation rules can be inserted.

3. The differentiation "variable" can actually be an arbitrary expression, which is then treated the same as a simple variable for differentiation purposes. (This is occasionally quite useful, such as when performing a square-free factorization or when deriving the Euler-Lagrange equations for a specific variational calculus problem.)

4. Higher-order partial derivatives can be requested directly by nested use of DIF, such as DIF (DIF(SIN(X*Y),X), Y). However, beware that repeated differentiation can require dramatically increasing time and space, especially for products, quotients, and composite expressions.

5. The useful utility function FREEOF (expr1, expr2) is a predicate which returns TRUE iff expr1 is free of (i.e. contains no occurrences of) expr2.

INTEGRATION Package Documentation

PURPOSE:
File INT.DIF provides facilities for indefinite symbolic integration.

PREREQUISITE FILE: DIF.ALG

USAGE:
INT (expression, variable).

EXAMPLE:
INT (A*X + SIN(X), X) —> A*X^2/2 - COS(X).

REMARKS:

1. When INT is unable to determine a closed-form integral of portions of an expression, the returned expression will contain unevaluated integrals of those portions. For example,
INT (X + A*#E^X/X, X) —> X^2/2 + A*INT(#E^X/X,X).

2. INT merely uses distribution over sums, extraction of factors which do not depend upon the integration variable, known integrals of the built-in functions, a few reduction rules, and a "derivatives-divides" substitution rule. Consequently, integration succeeds only for a relatively modest class of integrands. However:

a. The class is large enough to be quite useful,
b. File INTMORE.INT contains additional rules,
c. Integration of a truncated Taylor-series approximation of an integrand can often yield a truncated series representation of otherwise intractable integrals.

3. A careful study of files INT.DIF and INTMORE.INT reveals how additional integration rules can be inserted.

4. The integration "variable" can be an arbitrary expression, which is then treated the same as a simple variable for integration purposes.

5. Successful integration may depend upon the form of the integrand, after it is simplified according to the current flag settings. Generally speaking, it is best to employ conservative flag settings which do relatively little to alter the form of an expression. INT will automatically expand, factor, employ trigonometric transformations, etc. as necessary.

Extended INTEGRATION Package Documentation

PURPOSE:

File INTMORE.INT provides symbolic definite integration and extends the power of the indefinite integration provided by file INT.DIF.

PREREQUISITE FILE: INT.DIF

USAGE:

INT (expression, variable),
DEFINT (expression, variable, lowerlimit, upperlimit).

EXAMPLE:

DEFINT (A*X^2, X, 0, 1) —> A/3.

REMARKS:

1. DEFINT merely uses substitution into the indefinite integral, which is appropriate only for proper integrals.

2. When DEFINT is unable to determine a closed-form integral, the unevaluated integral is returned. For example,

DEFINT (X+A*%E^X/X, X, 0, 1) —> DEFINT (X+A*%E^X/X, X, 0, 1).

3. Nested integration can be used to request directly an iterated integration, such as occurs for appropriate multiple-integrations. For example, to integrate the expression y*x^2 over the upper unit semi-disk, we could evaluate

DEFINT (DEFINT(Y*X^2,Y,0,(1-X^2)^(1/2)), X, -1, 1).

However, beware that the class of expressions which is repeatedly integrable is dramatically smaller than the class which is once integrable.

4. File INT.DOC contains other appropriate remarks.

muSIMP/muMATH-79 Function Name and Variable Name INDEX

The following is an index of all the important function, variable, and constant names in both muSIMP and muMATH. Each name is followed by the module in which it occurs, a descriptor indicating the name's use, the page in the module's documentation on which it is explained, and finally the page in the module's muMATH source file on which it is defined. Function names are indicated by a set of parentheses following the name which contains the usual number of arguments given to the function. An asterisk (*) in the "Page Defined" column indicates that the item is incrementally defined in a number of places within the source.

Name	Module	Descriptor	Documented	Defined
DENDEN (1)	ALGEBRA	Recognizer		3
DENNUM	ALGEBRA	Control variable	2	2
DENNUM (1)	ALGEBRA	Recognizer		2
DENOM (1)	ARITH	Selector	2	4
DIF (2)	DIF	Numerical	1	1
DIFFERENCE (2)	muSIMP	Numerical	19	
DIVIDE (2)	muSIMP	Numerical	19	
DRIVER (0)	muSIMP	Evaluator	31	
ECHO	muSIMP	Control variable	22	
EMPTY (1)	muSIMP	Recognizer	10	
ENDBLOCK	muSIMP	Delimiter	31	
ENDFUN	muSIMP	Delimiter	17	
ENDLOOP	muSIMP	Delimiter	30	
ENDSUB	muSIMP	Delimiter	17	
EQ (2)	muSIMP	Comparator	11	
EVAL (1)	muSIMP	Evaluator	27	
EVSUB (3)	ARITH	Constructor/Evaluator	2	1
EXIT	muSIMP	Delimiter	31	
EXPAND (1)	ALGEBRA	Evaluator	4	5
EXPBAS	ALGEBRA	Control variable	2	4
EXPBAS (1)	ALGEBRA	Recognizer		4
EXPD (1)	ALGEBRA	Evaluator	4	7
EXPLODE (1)	muSIMP	Sub-atomic	18	
EXPON (1)	ARITH	Selector	2	4
FCTR (1)	ALGEBRA	Evaluator	4	8
FIRST (1)	muSIMP	Selector	7	
FLAGS	ALGEBRA	Global variable		*
FLAGS (0)	ALGEBRA	Printer	4	8
FOURONPI	TRGNEG	Constant		1
FREE (2)	SOLVE	Recognizer		1
FREE (2)	DIF	Recognizer	1	1
FUNCTION	muSIMP	Keyword	16	
GCD (2)	ARITH	Numerical	2	3
GET (2)	muSIMP	Property	14	
GETD (1)	muSIMP	Definition	16	
HALF	TRGPOS	Constant		1
IDENTITY (1)	ARITH	Identity function	2	1
IDMAT (1)	MATRIX	Constructor	1	2
INFIX	muSIMP	Parse property	35	
INT (2)	INT	Numerical	1	2
INTEGER (1)	muSIMP	Recognizer	10	
LBP	muSIMP	Parse property	35	
LCM (2)	ARITH	Numerical	2	3
LENGTH (1)	muSIMP	Sub-atomic	18	
LINELENGTH (1)	muSIMP	Printer	25	
LIST (N)	muSIMP	Constructor	8	
LN (1)	LOG	Numerical	1	1
LOAD (3)	muSIMP	System function	34	

Name	Module	Descriptor	Documented	Defined
LOG (2)	LOG	Numerical	1	1
LOGARITHM (1)	LOG	Recognizer		2
LOGBAS	LOG	Control variable	1	1
LOGEXPD	LOG	Control variable	2	1
LOGEXPD (2)	LOG	Evaluator		1
LOOP (N)	muSIMP	Evaluator	30	30
LOOP	muSIMP	Keyword	30	
LPAR	muSIMP	Constant	35	
MAPFUN (2)	EQN	Mapping		1
MATCH (2)	muSIMP	Reader	36	22
MATCHNOP (2)	muSIMP	Reader	36	
MIN (2)	ARITH	Numerical	2	3
MINUS (1)	muSIMP	Numerical	18	
MKPROD (1)	ARITH	Constructor		2
MKRAT (1)	ARITH	Constructor		5
MKSUM (1)	ARITH	Constructor		2
MOD (2)	muSIMP	Numerical	19	
MOVD (2)	muSIMP	Definition	16	
MULTIPLE (2)	ARITH	Comparator	2	1
NAME (1)	muSIMP	Recognizer	10	
NEGATIVE (1)	muSIMP	Recognizer	10	
NEGCOEFF (1)	ARITH	Recognizer	2	2
NEGMULT (2)	ARITH	Comparator	3	1
NEWLINE (0)	muSIMP	Printer	24	
NOT (1)	muSIMP	Logical	12	
NUM (1)	ARITH	Selector	3	2
NUMDEN	ALGEBRA	Control variable	2	3
NUMDEN (1)	ALGEBRA	Recognizer		3
NUMNUM	ALGEBRA	Control variable	1	1
NUMNUM (1)	ALGEBRA	Recognizer		1
NUMBER (1)	ARITH	Recognizer	3	1
OBLIST (0)	muSIMP	Constructor	8	
OR (N)	muSIMP	Logical	12	
ORDERP (2)	muSIMP	Comparator	11	
PARSE (2)	muSIMP	Reader	36	21
PBRCH	ARITH	Control variable	1,4	10
PION2	ARITH	Constant		12
PION4	TRGNEG	Constant		1
PLUS (2)	muSIMP	Numerical	18	
POSITIVE (1)	muSIMP	Recognizer	10	
POSMULT (2)	ARITH	Comparator	3	1
POWER (1)	ARITH	Recognizer	3	1
PREFIX	muSIMP	Parse property	35	
PRIMES	ARITH	Global variable	4	11
PRINT (1)	muSIMP	Printer	24	
PRINTLINE (1)	muSIMP	Printer	24	
PRODUCT (1)	ARITH	Recognizer	3	1
PROPERTY	muSIMP	Keyword	15	
PRTMATH (4)	muSIMP	Printer	25	
PUT (3)	muSIMP	Property	14	

Name	Module	Descriptor	Documented	Defined
PUTD (2)	muSIMP	Definition	16	
PWREXPD	ALGEBRA	Control variable	2	
QUERY (2)	INT	Reader/Printer		1
QUOTIENT (2)	muSIMP	Numerical	19	
RADIX (1)	muSIMP	Printer	25	
RBP	muSIMP	Parse property	35	
RDS (3)	muSIMP	Reader	22	
READ (0)	muSIMP	Reader	21	
READCHAR (0)	muSIMP	Reader	21	
RECIP (1)	ARITH	Recognizer	3	1
RECLAIM (0)	muSIMP	Storage	33	
REMPROP (2)	muSIMP	Property	14	
REPLACEF (2)	muSIMP	Modifier	9	
REPLACER (2)	muSIMP	Modifier	9	
REST (1)	muSIMP	Selector	7	
REVERSE (2)	muSIMP	Constructor	8	
ROW (1)	ARRAY	Recognizer		1
RPAR	muSIMP	Constant	35	
RREST (1)	muSIMP	Selector	7	
RRREST (1)	muSIMP	Selctor	7	
SAVE (3)	muSIMP	System	34	
SCAN (0)	muSIMP	Reader	21	
SEC (1)	TRGPOS	Numerical	1	1
SECOND (1)	muSIMP	Selector	7	
SIGN (1)	INT	Recognizer		1
SIMPU (2)	ARITH	Evaluator	3	4
SIN (1)	TRGPOS	Numerical	1	1
SIN (1)	TRGNEG	Numerical	2	1
SOLVE (2)	SOLVE	Numerical	1	3
SPACES (1)	muSIMP	Printer	24	
SUB (3)	ARITH	Constructor	3	1
SUBROUTINE	muSIMP	Keyword	35	
SUM (1)	ARITH	Recognizer	3	1
SYNTAX (N)	muSIMP	Reader	36	22
TAN (1)	TRGPOS	Numerical	1	1
TERMINATOR (0)	muSIMP	Recognizer		22
THIRD (1)	muSIMP	Selector	7	
TIMES (2)	muSIMP	Numerical	19	
TRACE (N)	TRACE	Debugger	1	1
TRGEXPD	TRGPOS	Control variable	2	1
TRGEXPD	TRGNEG	Control variable	2	
TRGEXPD (2)	TRGNEG	Evaluator	3	3
TRGSQ	TRGPOS	Control variable	2	1
UNION (2)	SOLVE	Constructor		1
UNTRACE (N)	TRACE	Debugger	1	3
WHEN	muSIMP	Keyword	31	
WRS (3)	muSIMP	Printer	25	

Name	Module	Descriptor	Documented	Defined
ZERO (1)	muSIMP	Recognizer	10	
ZEROBASE	ARITH	Control variable	1	9
ZEROEXPT	ARITH	Control variable	1	9
=	muSIMP	Comparator	11	
>	muSIMP	Comparator	11	
>	ARITH	Comparator		3
<	muSIMP	Comparator	11	
<	ARITH	Comparator		3
:	muSIMP	Assignment	13	
+	muSIMP	Numerical	20	
+	ARITH	Numerical		6
-	muSIMP	Numerical	20	
-	ARITH	Numerical		6
*	muSIMP	Numerical	20	
*	ARITH	Numerical		8
/	muSIMP	Numerical	20	
/	ARITH	Numerical		8
^	ARITH	Numerical		9
!	ARITH	Numerical	5	12
?	ARITH	Error function		8
\	MATRIX	Numerical	3	3
	MATRIX	Numerical	2	2
#ARB	SOLVE	Constant	1	1
#E	ARITH	Constant	4	
#I	ARITH	Constant	4	
#PI	ARITH	Constant	4	