MasterScope is an interactive program for analyzing and cross referencing user programs. It contains facilities for analyzing user functions to determine what other functions are called, how and where variables are bound, set, or referenced, and which functions use particular record declarations. MasterScope can analyze definitions directly from a file as well as in-memory definitions.

MasterScope maintains a data base of the results of the analyses it performs. Via a simple command language, you may interrogate the data base, call the editor on those expressions in functions that were analyzed which use variables or functions in a particular way, or display the tree structure of function calls among any set of functions.

MasterScope is interfaced with the editor and file manager so that when a function is edited or a new definition loaded in, MasterScope knows that it must reanalyze that function.

With the Medley release, MasterScope now understands Common Lisp **defun**, **defmacro**, and **defvar**.

Requirements

MSANALYZE, MSPARSE, MSCOMMON, MS-PACKAGE

You may also want to make use of Browser, DataBaseFns, and SEdit or DEdit.

Installation

Load MASTERSCOPE.DFASL and the other .DFASL files from the library.

MasterScope Command Language

You communicate with MasterScope using an English-like command language, e.g., WHO CALLS PRINT. With these commands, you can direct that functions be analyzed, interrogate the MasterScope data base, and perform other operations. The commands deal with sets of functions, variables, etc., and relations between them (e.g., call, bind). Sets correspond to English nouns, relations correspond to verbs.

A set of atoms can be specified in a variety of ways, either explicitly, e.g., FUNCTIONS ON FIE specifies the atoms in (FILEFNSLST 'FIE), or implicitly, e.g., NOT CALLING Y, where the meaning must be determined in the context of the rest of the

command. Such sets of atoms are the basic building blocks with which the command language deals.

MasterScope also deals with relations between sets.

For example, the relation CALL relates functions and other functions; the relations BIND and USE FREELY relate functions and variables. These relations get stored in the MasterScope data base when functions are analyzed. In addition, MasterScope "knows" about file manager conventions; CONTAIN relates files and various types of objects (functions, variables).

Sets and relations are used (along with a few additional words) to form sentence-like commands.

For example, the command WHO ON 'FOO USE 'X FREELY will print out the list of functions contained in the file FOO which use the variable X freely. The command EDIT WHERE ANY CALLS 'ERROR will call EDITF (see *IRM*) on those functions which have previously been analyzed that directly call ERROR, pointing at each successive expression where the call to ERROR actually occurs.

MasterScope Commands

The normal mode of communication with MasterScope is via commands. These are sentences in the MasterScope command language which direct MasterScope to answer questions or perform various operations.

MasterScope commands are typed into the Executive window, preceded by a period (.) to distinguish them from other commands to the Exec. MasterScope keywords can be in any package, so MasterScope commands can be issued in any type of Exec. The commands may be typed uppercase or lowercase.

Note: Any MasterScope command may be followed by OUTPUT *FILENAME* to send output to the given file rather than the terminal, e.g. WHO CALLS WHO OUTPUT CROSSREF.

ANALYZE *SET* [MasterScope command]

Analyzes the functions in *SET* (and any functions called by them) and includes the information gathered in the data base. MasterScope will not reanalyze a function if it thinks it already has valid information about that function in its data base. You may use the command REANALYZE to force reanalysis.

Note that whenever a function is referred to in a command as a subject of one of the relations, it is automatically analyzed; you need not give an explicit ANALYZE command. Thus, WHO IN MYFNS CALLS FIE will automatically analyze the functions in MYFNS if they have not already been analyzed.

Note also that only EXPR definitions will be analyzed; that is, MasterScope will not analyze compiled code. If necessary, the definition will be DWIMIFYed before analysis. If there is no in-core definition for a function (either in the function definition cell or an EXPR property), MasterScope will attempt to read in

the definition from a file. Files which have been explicitly mentioned previously in some command are searched first. If the definition cannot be found on any of those files, MasterScope looks among the files on FILELST for a definition. If a function is found in this manner, MasterScope will print a message "(reading from *FILENAME*)". If no definition can be found at all, MasterScope will print a message "*FN* can't be analyzed". If the function previously was known, the message "*FN* disappeared!" is printed.

REANALYZE *SET* [MasterScope command]

Causes MasterScope to reanalyze the functions in *SET* (and any functions called by them) even if it already has valid information in its data base. This would be necessary if you had disabled or subverted the file manager; e.g. performed PUTD's to change the definition of functions.

ERASE *SET* [MasterScope command]

Erases all information about the functions in *SET* from the data base. ERASE by itself clears the entire data base.

SHOW PATHS *PATHOPTIONS* [MasterScope command]

Displays a tree of function calls. This is described fully in "SHOW PATHS" below.

SET RELATION SET [MasterScope command]
SET IS *SET* [MasterScope command]
SET ARE *SET* [MasterScope command]

These commands have the same format as an English sentence with a subject (the first *SET*), a verb (*RELATION* or *IS* or *ARE*), and an object (the second *SET*). Any of the *SET*s within the command may be preceded by the question determiners WHICH or WHO (or just WHO alone).

For example, WHICH FUNCTIONS CALL X prints the list of functions that call the function X.

RELATION may be one of the relation words in present tense (CALL, BIND, TEST, SMASH, etc.) or used as a passive (e.g., WHO IS CALLED BY WHO). Other variants are allowed, e.g. WHO DOES X CALL, IS FOO CALLED BY FIE, etc.

The interpretation of the command depends on the number of question elements present:

If there is no question element, the command is treated as an assertion and MasterScope returns either T or NIL, depending on whether that assertion is true. Thus, ANY IN MYFNS CALL HELP will print T if any function in MYFNS call the function HELP, and NIL otherwise.

If there is one question element, MasterScope returns the list of items for which the assertion would be true.

For example,

```
MYFN BINDS WHO USED FREELY BY YOURFN
```

prints the list of variables bound by MYFN which are also used freely by YOURFN.

If there are two question elements, MasterScope will print a doubly indexed list:

```
_. WHO CALLS WHO IN /FNS
RECORDSTATEMENT --   /RPLNODE
RECORDECL1 --        /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 --     /PUTHASH
UNCLISPTRAN --       /PUTHASH, /RPLNODE2
RECORDWORD --        /RPLACA
RECORD1 --           /RPLACA, /SETTOPVAL
EDITREC --           /SETTOPVAL
```

EDIT WHERE *SET RELATION SET [- EDITCOMS]* [MasterScope command]

(WHERE may be omitted.) The first *SET* refers to a set of functions. The EDIT command calls the editor on each expression where the *RELATION* actually occurs.

For example, EDIT WHERE ANY CALL ERROR will call EDITF on each (analyzed) function which calls ERROR stopping within a TTY: at each call to ERROR. Currently you cannot EDIT WHERE a file which CONTAINS a datum, nor where one function CALLS another SOMEHOW.

EDITCOMS, if given, is a list of commands passed to EDITF to be performed at each expression.

For example,

```
EDIT WHERE ANY CALLS MYFN DIRECTLY - (SW 2 3) P
```

will switch the first and second arguments to MYFN in every call to MYFN and print the result. EDIT WHERE ANY ON MYFILE CALL ANY NOT @ GETD will call the editor on any expression involving a call to an undefined function.

Note that EDIT WHERE X SETS Y will point only at those expressions where Y is actually set, and will skip over places where Y is otherwise mentioned.

SHOW WHERE *SET RELATION SET* [MasterScope command]

Like the EDIT command except merely prints out the expressions without calling the editor.

EDIT *SET [- EDITCOMS]* [MasterScope command]

Calls EDITF on each function in *SET*. *EDITCOMS*, if given, will be passed as a list of editor commands to be Executed.

For example,

```
EDIT ANY CALLING FN1 - (R FN1 FN2)
```

will replace FN1 by FN2 in those functions that call FN1.

DESCRIBE *SET* [MasterScope command]

Prints the BIND, USE FREELY and CALL information about the functions in *SET*.

For example, the command DESCRIBE PRINTARGS might print out:

```
PRINTARGS[N,FLG]
   binds:        TEM,LST,X
   calls:        MSRECORDFILE,SPACES,PRIN1
   called by:    PRINTSENTENCE,MSHELP,CHECKER
```

This shows that PRINTARGS has two arguments, N and FLG; binds internally the variables TEM, LST and X; calls MSRECORDFILE, SPACES and PRIN1; and is called by PRINTSENTENCE, MSHELP, and CHECKER.

You can specify additional information to be included in the description. DESCRIBELST is a list each of whose elements is a list containing a descriptive string and a form. The form is evaluated (it can refer to the name of the funtion being described by the free variable FN). If it returns a non-NIL value, the description string is printed followed by the value. If the value is a list, its elements are printed with commas between them.

For example, the entry

```
("types:  " (GETRELATION FN '(USE TYPE) T)
```

would include a listing of the types used by each function.

CHECK *SET* [MasterScope command]

Checks for various anomalous conditions (mainly in the compiler declarations) for the files in *SET* (if *SET* is not given, FILELST is used).

For example, this command will warn about:

Variables which are bound but never referenced.

Functions in BLOCKS declarations which aren't on the file containing the declaration.

Functions declared as ENTRIES but not in the block.

Variables which may not need to be declared SPECVARS because they are not used freely below the places where they are bound.

etc.

FOR *VARIABLE SET I.S.TAIL* [MasterScope command]

This command provides a way of combining CLISP iterative statements with MasterScope. An iterative statement will be constructed in which *VARIABLE* is iteratively assigned to each element of *SET*, and then the iterative statement tail *I.S.TAIL* is Executed.

For example,

```
FOR X CALLED BY FOO WHEN CCODEP DO (PRINTOUT T X
,,,  (ARGLIST X) T)
```

will print out the name and argument list of all of the compiled functions which are called by FOO.

MasterScope Relations

A relation is specified by one of the keywords below. Some of these "verbs" accept modifiers.

For example, USE, SET, SMASH and REFERENCE all may be modified by FREELY. The modifier may occur anywhere within the command. If there is more than one verb, any modifier between two verbs is assumed to modify the first one.

For example, in

```
USING ANY FREELY OR SETTING X,
```

FREELY modifies USING but not SETTING. The entire phrase is interpreted as the set of all functions which either use any variable freely or set the variable X, whether or not X is set freely. Verbs can occur in the present tense (e.g., USE, CALLS, BINDS, USES) or as present or past participles (e.g., CALLING, BOUND, TESTED). The relations (with their modifiers) recognized by MasterScope are:

CALL [MasterScope relation]

Function F1 calls F2 if the definition of F1 contains a form (F2 --). The CALL relation also includes any instance where a function uses a name as a function, as in

```
(APPLY (QUOTE F2) --), (FUNCTION F2), etc.
```

(CALL and CALLS are equivalent.)

CALL SOMEHOW [MasterScope relation]

One function calls another SOMEHOW if there is some path from the first to the other. That is, if F1 calls F2, and F2 calls F3, then F1 CALLS F3 SOMEHOW.

This information is not stored directly in the data base; instead, MasterScope stores only information about direct function calls, and (re)computes the CALL SOMEHOW relation as necessary.

USE [MasterScope relation]

If unmodified, the relation USE denotes variable usage in any way; it is the union of the relations SET, SMASH, TEST, and REFERENCE.

SET [MasterScope relation]

A function SETs a variable if the function contains a form

```
(SETQ var --), (SETQQ var --), etc.
```

SMASH [MasterScope relation]

A function SMASHes a variable if the function calls a destructive list operation (RPLACA, RPLACD, DREMOVE, SORT, etc.) on the value of that variable. MasterScope will also find instances where the operation is performed on a part of the value of the variable. For example, if a function contains a form (RPLACA (NTH X 3) T), it will be noted as SMASHing X.

If the function contains a sequence (SETQ Y X), (RPLACA Y T), then Y is noted as being SMASHed, but not X.

TEST [MasterScope relation]

A variable is TESTed by a function if its value is only distinguished between NIL and non-NIL.

For example, the form (COND ((AND X --) --)) tests the value of X.

REFERENCE [MasterScope relation]

This relation includes all variable usage except for SET.

Note: The verbs USE, SET, SMASH, TEST and REFERENCE may be modified by the words FREELY or LOCALLY. A variable is used FREELY if it is not bound in the function at the place of its use. It is used LOCALLY if the use occurs within a PROG or LAMBDA that binds the variable.

MasterScope also distinguishes between CALL DIRECTLY and CALL INDIRECTLY. A function is called directly if it occurs as CAR-of-form in a normal evaluation context. A function is called indirectly if its name appears in a context which does not imply its immediate evaluation, for example (SETQ Y (LIST (FUNCTION FOO) 3)). The distinction is whether or not the compiled code of the caller would contain a direct call to the callee.

Note that an occurrence of (FUNCTION FOO) as the functional argument to one of the built-in mapping functions which compile open is considered to be a direct call.

In addition, CALL FOR EFFECT (where the value of the function is not used) is distinguished from CALL FOR VALUE.

BIND [MasterScope relation]

The BIND relation between functions and variables includes both variables bound as function arguments and those bound in an internal PROG or LAMBDA expression.

USE AS A FIELD [MasterScope relation]

MasterScope notes all uses of record field names within FETCH, REPLACE or CREATE expressions.

FETCH [MasterScope relation]

Use of a field within a FETCH expression.

REPLACE [MasterScope relation]

Use of a record field name within a REPLACE or CREATE expression.

USE AS A RECORD [MasterScope relation]

MasterScope notes all uses of record names within CREATE or TYPE? expressions. Additionally, in (fetch (FOO FIE) of X), FOO is used as a record name.

CREATE [MasterScope relation]

Use of a record name within a CREATE expression.

USE AS A PROPERTY NAME [MasterScope relation]

MasterScope notes the property names used in expressions such as GETPROP, PUTPROP, GETLIS, etc., if the name is quoted; e.g. if a function contains a form (GETPROP X (QUOTE INTERP)), then that function USEs INTERP as a property name.

USE AS A CLISP WORD [MasterScope relation]

MasterScope notes all iterative statement operators and user defined CLISP words as being used as a CLISP word.

CONTAIN [MasterScope relation]

Files CONTAIN functions, records, and variables. This relation is not stored in the data base but is computed using the file manager.

DECLARE AS LOCALVAR [MasterScope relation]
DECLARE AS SPECVAR [MasterScope relation]

MasterScope notes internal calls to DECLARE from within functions.

ACCEPT [MasterScope relation]
SPECIFY [MasterScope relation]
KEYCALL [MasterScope relation]

MasterScope notes keyword arguments of Common Lisp functions when they are analyzed and when they are called.

FOO ACCEPTS :BAR is true if FOO is a Common Lisp function that accepts the keyword :BAR. FOO ACCEPTS &ALLOW-OTHER-KEYS is true if FOO has &ACCEPT-OTHER-KEYS in its lambda list.

FOO SPECIFIES :BAR is true if FOO is a function that calls any function with the keyword :BAR; the function in question must ACCEPT :BAR.

FOO KEYCALLS BAR is true if FOO is a function and calls BAR with one or more keywords it ACCEPTS.

FLET [MasterScope relation]
LABEL [MasterScope relation]
MACROLET [MasterScope relation]
LOCAL-DEFINE [MasterScope relation]

MasterScope tracks uses of Common Lisp local definition forms (it currently does not expand them while analyzing them, however).

FOO FLETS BAR is true of FOO is a function with a FLET defining BAR local to FOO.

LABELS and MACROLETS are similar. LOCAL-DECLARES is the union of FLETS, LABELS, and MACROLETS.

Abbreviations

The following abbreviations are recognized:

FREE = FREELY
LOCAL = LOCALLY
PROP = PROPERTY
REF = REFERENCE

Also, the words A, AN and NAME (after AS) are "noise" words and may be omitted.

MasterScope Templates

MasterScope uses templates (see "Effecting MasterScope Analysis" below) to decide which relations hold between functions and their arguments.

For example, the information that SORT SMASHes its first argument is contained in the template for SORT. MasterScope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations. You may change existing templates or insert new ones in MasterScope's tables via the SETTEMPLATE function (below).

MasterScope also constructs templates to handle Common Lisp functions with keyword arguments. These constructed templates are noticed by FILES? and can be saved if desired, or MasterScope can recreate them by analyzing the functions again.

MasterScope Set Specifications

A set is a collection of things (functions, variables, etc.). A set is specified by a set phrase, consisting of a determiner (e.g., ANY, WHICH, WHO) followed by a type (e.g., FUNCTIONS, VARIABLES) followed by a specification (e.g., IN MYFNS). The determiner, type and specification may be used alone or in combination.

For example,

```
ANY FUNCTIONS IN MYFNS,
VARIABLES IN GLOBALVARS, and
WHO
```

are all acceptable set phrases.

Note: Sets may also be specified with relative clauses introduced by the word THAT, e.g. THE FUNCTIONS THAT BIND 'X.

'ATOM [MasterScope set specification]

The simplest way to specify a set consisting of a single thing is by the name of that thing.

For example, in the command WHO CALLS 'ERROR, the function ERROR is referred to by its name. Although the ' (apostrophe)

can be left out, to resolve possible ambiguities names should usually be quoted; e.g., WHO CALLS 'CALLS will return the list of functions which call the function CALLS.

'LIST [MasterScope set specification]

Sets consisting of several atoms may be specified by naming the atoms.

For example, the command WHO USES '(A B) returns the list of functions that use the variables A or B.

IN *EXPRESSION* [MasterScope set specification]

The form *EXPRESSION* is evaluated, and its value is treated as a list of the elements of a set.

For example, IN GLOBALVARS specifies the list of variables in the value of the variable GLOBALVARS.

@ PREDICATE [MasterScope set specification]

A set may also be specified by giving a predicate which the elements of that set must satisfy. *PREDICATE* is either a function name, a LAMBDA expression, or an expression in terms of the variable X. The specification @ *PREDICATE* represents all atoms for which the value of *PREDICATE* is non-NIL.

For example, @ EXPRP specifies all those atoms which have EXPR definitions; @ (STRPOSL X CLISPCHARRAY) specifies those atoms which contain CLISP characters. The universe to be searched is either determined by the context within the command (e.g., in WHO IN FOOFNS CALLS ANY NOT @ GETD, the predicate is only applied to functions which are called by any functions in the list FOOFNS), or in the extreme case, the universe defaults to the entire set of things which have been noticed by MasterScope, as in the command WHO IS @ EXPRP.

LIKE *ATOM* [MasterScope set specification]

ATOM may contain ESCapes; it is used as a pattern to be matched, as in the editor.

For example, WHO LIKE /R$ IS CALLED BY ANY would find both /RPLACA and /RPLNODE.

(The ESC character prints out as a $; it is a wildcard for any number of characters.)

FIELDS OF *SET* [MasterScope set specification]

SET is a set of records. This denotes the field names of those records.

For example, the command WHO USES ANY FIELDS OF BRECORD returns the list of all functions which do a fetch or replace with any of the field names declared in the record declaration of BRECORD.

KNOWN [MasterScope set specification]

The set of all functions which have been analyzed.

For example, the command WHO IS KNOWN will print out the list of functions which have been analyzed.

THOSE [MasterScope set specification]

The set of things printed out by the last MasterScope question.

For example, following the command

```
WHO IS USED FREELY BY PARSE
```

you could ask WHO BINDS THOSE to find out where those variables are bound.

ON PATH *PATHOPTIONS* [MasterScope set specification]

Refers to the set of functions which would be printed by the command SHOW PATHS *PATHOPTIONS*.

For example,

```
IS FOO BOUND BY ANY ON PATH TO 'PARSE
```

tests whether FOO might be bound above the function PARSE (that is, whether FOO is bound in any function that is higher up in the calling tree than PARSE is) . SHOW PATHS is explained in detail below.

Set Specifications by Relation

A set may also be specified by giving a relation its members must have with the members of another set:

*RELATION*ING *SET* [MasterScope set specification]

*RELATION*ING is used here generically to mean any of the relation words in the present participle form (possibly with a modifier), e.g., USING, SETTING, CALLING, BINDING. *RELATION*ING *SET* specifies the set of all objects which have that relation with some element of *SET*.

For example, CALLING X specifies the set of functions which call the function X; USING ANY IN FOOVARS FREELY specifies the set of functions which uses freely any variable in the value of FOOVARS.

*RELATION*ED BY *SET* [MasterScope set specification]
*RELATION*ED IN *SET* [MasterScope set specification]

This is similar to the *RELATION*ING construction.

For example, CALLED BY ANY IN FOOFNS represents the set of functions which are called by any element of FOOFNS; USED FREELY BY ANY CALLING ERROR is the set of variables which are used freely by any function which also calls the function ERROR.

Set Specifications by Blocktypes

BLOCKTYPE OF *FUNCTIONS* [MasterScope set specification]
BLOCKTYPE ON *FILES* [MasterScope set specification]

These phrases allow you to ask about BLOCKS declarations on files (see *IRM*). *BLOCKTYPE* is one of LOCALVARS, SPECVARS, GLOBALVARS, ENTRIES, BLKFNS, BLKAPPLYFNS, or RETFNS.

BLOCKTYPE OF *FUNCTIONS* specifies the names which are declared to be *BLOCKTYPE* in any blocks declaration which contain any of *FUNCTIONS* (a "set" of functions). The "functions" in *FUNCTIONS* can either be block names or just functions in a block.

For example,

```
WHICH ENTRIES OF ANY CALLING 'Y BIND ANY
GLOBALVARS ON 'FOO.
```

BLOCKTYPE ON *FILES* specifies all names which are declared to be *BLOCKTYPE* on any of the given *FILES* (a "set" of files).

Set Determiners

Set phrases may be preceded by a determiner, which is one of the words THE, ANY, WHO or WHICH. The question determiners (WHO and WHICH) are meaningful in only some of the commands, namely those that take the form of questions. ANY and WHO (or WHOM) can be used alone; they are wild-card elements, e.g., the command WHO USES ANY FREELY, will print out the names of all (known) functions which use any variable freely. If the determiner is omitted, ANY is assumed; e.g. the command WHO CALLS '(PRINT PRIN1 PRIN2) will print the list of functions which call any of PRINT, PRIN1, PRIN2. THE is also allowed, e.g. WHO USES THE RECORD FIELD FIELDX.

Set Types

Any set phrase has a type; that is, a set may specify either functions, variables, files, record names, record field names or property names. The type may be determined by the context within the command (e.g., in CALLED BY ANY ON FOO, the set ANY ON FOO is interpreted as meaning the functions on FOO since only functions can be CALLED), or you may give the type explicitly (e.g., FUNCTIONS ON FIE).

The following types are recognized: FUNCTIONS, VARIABLES, FILES, PROPERTY NAMES, RECORDS, FIELDS, I.S.OPRS. Also, the abbreviations FNS, VARS, PROPNAMES or the singular forms FUNCTION, FN, VARIABLE, VAR, FILE, PROPNAME, RECORD, FIELD are recognized.

Note that most of these types correspond to built-in file manager types (see *IRM*).

The type is used by MasterScope in a variety of ways when interpreting the set phrase:

(1) Set types are used to disambiguate possible parsings.

For example, both commands

```
WHO SETS ANY BOUND IN X OR USED BY Y
```

```
WHO SETS ANY BOUND IN X OR CALLED BY Y
```

have the same general form. However, the first case is parsed as

```
WHO SETS ANY (BOUND BY X OR USED BY Y)
```

since both BOUND BY X and USED BY Y refer to variables; while the second case is parsed as

```
WHO SETS ANY BOUND IN (X OR CALLED BY Y),
```

since CALLED BY Y and X must refer to functions.

Note that parentheses may be used to group phrases.

(2) The type is used to determine the modifier for USE:

```
FOO USES WHICH RECORDS
```

is equivalent to

```
FOO USES WHO AS A RECORD FIELD.
```

(3) The interpretation of CONTAIN depends on the type of its object: the command

```
WHAT FUNCTIONS ARE CONTAINED IN MYFILE
```

prints the list of functions in MYFILE.

```
WHAT RECORDS ARE ON MYFILE
```

prints the list of records.

(4) The implicit universe in which a set expression is interpreted depends on the type:

```
ANY VARIABLES @ GETD
```

is interpreted as the set of all variables which have been noticed by MasterScope (i.e., bound or used in any function which has been analyzed) that also have a definition.

```
ANY FUNCTIONS @ (NEQ (GETTOPVAL X) 'NOBIND)
```

is interpreted as the set of all functions which have been noticed (either analyzed or called by a function which has been analyzed) that also have a top-level value.

Conjunctions of Sets

Sets may be joined by the conjunctions AND and OR or preceded by NOT to form new sets. AND is always interpreted as meaning intersection; OR as union; NOT as complement.

For example, the set CALLING X AND NOT CALLED BY Y specifies the set of all functions which call the function X but are not called by Y.

Note: MasterScope's interpretation of AND and OR follow Lisp conventions rather than the conventional English interpretation.

"Calling X and Y" would, in English, be interpreted as the intersection of (CALLING X) and (CALLING Y); but MasterScope interprets CALLING X AND Y as CALLING ('X AND 'Y), which is the null set.

Only sets may be joined with conjunctions. Joining modifiers, as in

```
USING X AS A RECORD FIELD OR PROPERTY NAME
```

is not allowed; in this case, you must type

```
USING X AS A RECORD FIELD OR USING X AS A
PROPERTY NAME
```

As described above, the type of set is used to disambiguate parsings. The algorithm used is to first try to match the type of the phrases being joined and then try to join with the longest preceding phrase.

In any case, you may group phrases with parentheses to specify the manner in which conjunctions should be parsed.

SHOW PATHS

In trying to work with large programs, you can lose track of the hierarchy of functions. The MasterScope SHOW PATHS command aids you by providing a map showing the calling structure of a set of functions. SHOW PATHS prints out a tree structure showing which functions call which other functions.

Loading the Browser library module modifies the SHOW PATHS command so the command's output is displayed as an undirected graph.

The SHOW PATHS command takes the form: SHOW PATHS followed by some combination of the following path options:

FROM *SET* [MasterScope path option]

Display the function calls from the elements of *SET*.

TO *SET* [MasterScope path option]

Display the function calls leading to elements of *SET*. If TO is given before FROM (or no FROM is given), the tree is inverted and a message (inverted tree) is printed to warn you that if FN1 appears after FN2 it is because FN1 is called by FN2.

Note: When both FROM and TO are given, the first one indicates a set of functions which are to be displayed while the second restricts the paths that will be traced; i.e., the command SHOW PATHS FROM X TO Y will trace the elements of the set CALLED SOMEHOW BY X AND CALLING Y SOMEHOW.

If TO is not given, TO KNOWN OR NOT @ GETD is assumed; that is, only functions which have been analyzed or which are undefined will be included.

Note that MasterScope will analyze a function while printing out the tree if that function has not previously been seen and it currently has an EXPR definition. Thus, any function which can be analyzed will be displayed.

AVOIDING *SET* [MasterScope path option]

Do not display any function in *SET*. AMONG is recognized as a synonym for AVOIDING NOT.

For example, SHOW PATHS TO ERROR AVOIDING ON FILE2 will not display (or trace) any function on FILE2.

NOTRACE *SET* [MasterScope path option]

Do not trace from any element of *SET*. NOTRACE differs from AVOIDING in that a function which is marked NOTRACE will be printed, but the tree beyond it will not be expanded. The functions in an AVOIDING set will not be printed at all.

For example,

```
SHOW PATHS FROM ANY ON FILE1 NOTRACE ON FILE2
```

will display the tree of calls eminating from FILE1, but will not expand any function on FILE2.

SEPARATE *SET* [MasterScope path option]

Give each element of *SET* a separate tree.

Note: FROM and TO only insure that the designated functions will be displayed. SEPARATE can be used to guarantee that certain functions will begin new tree structures. SEPARATE functions are displayed in the same manner as overflow lines; i.e., when one of the functions indicated by SEPARATE is found, it is printed followed by a forward reference (a lower-case letter in braces) and the tree for that function is then expanded below.

LINELENGTH *N* [MasterScope path option]

Resets LINELENGTH to *N* before displaying the tree. The linelength is used to determine when a part of the tree should "overflow" and be expanded lower.

Error Messages

When you give MasterScope a command, the command is first parsed, i.e. translated to an internal representation, and then the internal representation is interpreted.

If a command cannot be parsed, e.g. if you typed

```
SHOW WHERE CALLED BY X
```

MasterScope would reply

Sorry, I can't parse that!

and generate an error.

If the command is of the correct form but cannot be interpreted (e.g., the command EDIT WHERE ANY CONTAINS ANY) MasterScope will print the message

Sorry, that isn't implemented!

and generate an error.

If the command requires some functions having been analyzed (e.g., the command WHO CALLS X) and the data base is empty, MasterScope will print the message

Sorry, no functions have been analyzed!

and generate an error.

Macro Expansion

As part of analysis, MasterScope will expand the macro definition of called functions if they are not otherwise defined (see *IRM*). MasterScope always expands Common Lisp DEFMACRO definitions (unless it finds a template for the macro).

MasterScope Interlisp macro expansion is controlled by a variable:

MSMACROPROPS [Variable]

Value is an ordered list of macro-property names that MasterScope will search to find a macro definition. Only the kinds of macros that appear on MSMACROPROPS will be expanded. All others will be treated as function calls and left unexpanded. Initially (MACRO).

Note: MSMACROPROPS initially contains only MACRO (not 10MACRO, DMACRO, etc.) on the assumption that the machine-dependent macro definitions are more likely "optimizers".

If you edit a macro, MasterScope will know to reanalyze the functions which call that macro.

Note: If your macro is of the "computed-macro" style, and it calls functions which you edit, MasterScope will not notice. You must be careful to tell masterscope to REANALYZE the appropriate functions (e.g., if you edit FOOEXPANDER which is used to expand FOO macros, you have to REANALYZE ANY CALLING FOO.

Effecting MasterScope Analysis

MasterScope analyzes the EXPR definition of a function, and notes in its data base the relations that this function has with other functions and with variables. To perform this analysis, MasterScope uses templates which describe the behavior of functions.

For example, the information that SORT destructively modifies its first argument is contained in the template for SORT. MasterScope initially contains templates for most system functions that set variables, test their arguments, or perform destructive operations.

A template is a list structure containing any of the following atoms:

PPE [in MasterScope template]

If an expression appears in this location, there is most likely a parenthesis error.

MasterScope notes this as a call to the function ppe (lowercase). Therefore, SHOW WHERE ANY CALLS ppe will print out all possible parenthesis errors. When MasterScope finds a possible parenthesis error in the course of analyzing a function definition, rather than printing the usual ".", it prints out a "?" instead. MasterScope notes functions called with keywords they do not accept as calls to ppe.

NIL [in MasterScope template]

The expression occuring at this location is not evaluated.

SET [in MasterScope template]

A variable appearing at this place is set.

SMASH [in MasterScope template]

The value of this expression is smashed.

TEST [in MasterScope template]

Is used as a predicate (that is, the only use of the value of the expression is whether it is NIL or non-NIL).

PROP [in MasterScope template]

Is used as a property name. If the value of this expression is of the form (QUOTE *ATOM*), MasterScope will note that *ATOM* is USED AS A PROPERTY NAME.

For example, the template for GETPROP is (EVAL PROP . PPE).

KEYWORD key1... [in MasterScope template]

Must appear at the end of a template followed by the keywords the templated function accepts.

For example, the template for CL:MEMBER is (EVAL EVAL KEYWORDS :TEST :TEST-NOT :KEY).

FUNCTION [in MasterScope template]

The expression at this point is used as a functional argument.

For example, the template for MAPC is

```
(SMASH FUNCTION FUNCTION . PPE).
```

FUNCTIONAL [in MasterScope template]

The expression at this point is used as a functional argument. This is like FUNCTION, except that MasterScope distinguishes between functional arguments to functions which compile open from those that do not. For the latter (e.g. SORT and APPLY), FUNCTIONAL should be used rather than FUNCTION.

EVAL [in MasterScope template]

The expression at this location is evaluated (but not set, smashed, tested, used as a functional argument, etc.).

RETURN [in MasterScope template]

The value of the function (of which this is the template) is the value of this expression.

TESTRETURN [in MasterScope template]

A combination of TEST and RETURN: If the value of the function is non-NIL, then it is returned. For instance, a one-element COND clause is this way.

EFFECT [in MasterScope template]

The expression at this location is evaluated, but the value is not used. (That is, it is evaluated for its side effect only.)

FETCH [in MasterScope template]

An atom at this location is a field which is fetched.

REPLACE [in MasterScope template]

An atom at this location is a field which is replaced.

RECORD [in MasterScope template]

An atom at this location is used as a record name.

CREATE [in MasterScope template]

An atom at this location is a record which is created.

BIND [in MasterScope template]

An atom at this location is a variable which is bound.

CALL [in MasterScope template]

An atom at this location is a function which is called.

CLISP [in MasterScope template]

An atom at this location is used as a CLISP word.

! [in MasterScope template]

This atom, which can only occur as the first element of a template, allows you to specify a template for the CAR of the

function form. If ! doesn't appear, the CAR of the form is treated as if it had a CALL specified for it. In other words, the templates (.. EVAL) and (! CALL .. EVAL) are equivalent.

If the next atom after a ! is NIL, this specifies that the function name should not be remembered.

For example, the template for AND is (! NIL .. TEST RETURN), which means that if you see an AND, don't remember it as being called. This keeps the MasterScope data base from being cluttered by too many uninteresting relations. MasterScope also throws away relations for COND, CAR, CDR, and a couple of others.

Special Forms

In addition to the above atoms that occur in templates, there are some special forms which are lists keyed by their CAR.

.. *TEMPLATE* [in MasterScope template]

Any part of a template may be preceded by the atom .. (two periods) which specifies that the template should be repeated an indefinite number ($N >= 0$) of times to fill out the expression.

For example, the template for COND might be

```
(.. (TEST .. EFFECT RETURN))
```

while the template for SELECTQ is

```
(EVAL .. (NIL .. EFFECT RETURN) RETURN).
```

(Although MasterScope "throws away" the relations for COND, it makes sense to template COND because there may be important information within the arguments of COND.)

(BOTH *TEMPLATE1 TEMPLATE2*) [in MasterScope template]

Analyze the current expression twice, using the each of the templates in turn.

(IF *EXPRESSION TEMPLATE*$_1$ *TEMPLATE*$_2$) [in MasterScope template]

Evaluate *EXPRESSION* at analysis time (the variable EXPR will be bound to the expression which corresponds to the IF), and if the result is non-NIL, use *TEMPLATE1*, otherwise *TEMPLATE2*. If *EXPRESSION* is a literal atom, it is APPLYd to EXPR.

For example,

```
(IF LISTP (RECORD FETCH) FETCH)
```

specifies that if the current expression is a list, then the first element is a record name and the second element a field name, otherwise it is a field name.

(@ *EXPRFORM TEMPLATEFORM*) [in MasterScope template]

Evaluate *EXPRFORM* giving *EXPR*, evaluate *TEMPLATEFORM* giving *TEMPLATE*. Then analyze *EXPR* with *TEMPLATE*. @ lets you compute on the fly both a template and an expression to analyze with it. The forms can use the variable EXPR, which is bound to the current expression.

(MACRO . *MACRO*) [in MasterScope template]

MACRO is interpreted in the same way as macros (see *IRM*) and the resulting form is analyzed. If the template is the atom MACRO alone, MasterScope will use the MACRO property of the function itself. This is useful when analyzing code which contains calls to user-defined macros. If you change a macro property (e.g. by editing it) of an atom which has template of MACRO, MasterScope will mark any function which used that macro as needing to be reanalyzed.

Some examples of templates:

Function:	Template:
DREVERSE	(SMASH . PPE)
AND	(! NIL TEST .. RETURN)
MAPCAR	(EVAL FUNCTION FUNCTION)
COND	(! NIL .. (IF CDR (TEST .. EFFECT RETURN) (TESTRETURN . PPE)))

Templates may be changed and new templates defined using the following functions:

(GETTEMPLATE *FN*) [Function]

Returns the current template of *FN*.

(SETTEMPLATE *FN TEMPLATE*) [Function]

Changes the template for the function *FN* and returns the old value. If any functions in the data base are marked as calling *FN*, they will be marked as needing reanalysis.

Updating the MasterScope Data Base

MasterScope is interfaced to the editor and file manager so that it notes whenever a function has been changed, either through editing or loading in a new definition. Whenever a command is given which requires knowing the information about a specific function, if that function has been noted as being changed, the function is automatically reanalyzed before the command is interpreted. If the command requires that all the information in the data base be consistent (e.g., you ask WHO CALLS X) then all functions which have been marked as changed are reanalyzed.

MasterScope Entries

(MASTERSCOPE *COMMAND*—) [Function]

Top level entry to MasterScope. If *COMMAND* is NIL, will enter into an Executive in which you may enter commands. If *COMMAND* is not NIL, the command is interpreted and MASTERSCOPE will return the value that would be printed by the command.

Note that only the question commands return meaningful values.

(CALLS *FN USEDATABASE*—) [Function]

FN can be a function name, a definition, or a form.

Note: CALLS will also work on compiled code. CALLS returns a list of four elements:

Functions called by *FN*
Variables bound in *FN*
Variables used freely in *FN*
Variables used globally in *FN*

For the purpose of CALLS, variables used freely which are on GLOBALVARS or have a property GLOBALVAR value T are considered to be used globally. If *USEDATABASE* is NIL (or *FN* is not a symbol), CALLS will perform a one-time analysis of *FN*. Otherwise (i.e. if *USEDATABASE* is non-NIL and *FN* a function name), CALLS will use the information in MasterScope's data base (*FN* will be analyzed first if necessary).

(CALLSCCODE *FN* ——) [Function]

The subfunction of CALLS which analyzes compiled code. CALLSCCODE returns a list of elements:

Functions called via "linked" function calls (not implemented in Interlisp-D)

Functions called regularly

Variables bound in *FN*

Variables used freely

Variables used globally

(FREEVARS *FN USEDATABASE*) [Function]

Equivalent to (CADDR (CALLS *FN USEDATABASE*)). Returns the list of variables used freely within *FN*.

(SETSYNONYM *PHRASE MEANING*—) [Function]

Defines a new synonym for MasterScope's parser. Both *OLDPHRASE* and *NEWPHRASE* are words or lists of words; anywhere *OLDPHRASE* is seen in a command, *NEWPHRASE* will be substituted.

For example,

```
(SETSYNONYM 'GLOBALS '(VARS IN GLOBALVARS OR
@(GETPROP X 'GLOBALVAR)))
```

would allow you to refer with the single word GLOBALS to the set of variables which are either in GLOBALVARS or have a GLOBALVAR property.

Functions for Writing Routines

The following functions are provided for users who wish to write their own routines using MasterScope's data base:

(PARSERELATION *RELATION*) [Function]

RELATION is a relation phrase; e.g., (PARSERELATION '(USE FREELY)). PARSERELATION returns an internal representation for *RELATION*. For use in conjunction with GETRELATION.

(GETRELATION *ITEM RELATION INVERTED*) [Function]

RELATION is an internal representation as returned by PARSERELATION (if not, GETRELATION will first perform (PARSERELATION *RELATION*)).

ITEM is an atom. GETRELATION returns the list of all atoms which have the given relation to *ITEM*.

For example,

```
(GETRELATION 'X '(USE FREELY))
```

returns the list of variables that X uses freely.

If *INVERTED* is T, the inverse relation is used; e.g.

```
(GETRELATION 'X '(USE FREELY) T)
```

returns the list of functions which use X freely.

If *ITEM* is NIL, GETRELATION will return the list of atoms which have *RELATION* with *any* other item; i.e., it answers the question WHO *RELATION*S ANY.

Note that GETRELATION does not check to see if *ITEM* has been analyzed, or that other functions that have been changed have been reanalyzed.

(TESTRELATION *ITEM RELATION ITEM2 INVERTED*) [Function]

Is equivalent to (MEMB *ITEM2* (GETRELATION *ITEM RELATION INVERTED*)); that is, it tests if *ITEM* and *ITEM2* are related via *RELATION*.

If *ITEM2* is NIL, the call is equivalent to

(NOT (NULL (GETRELATION *ITEM RELATION INVERTED*)))

i.e., TESTRELATION tests if *ITEM* has the given *RELATION* with any other item.

(MAPRELATION *RELATION MAPFN*) [Function]

Calls the function *MAPFN* on every pair of items related via *RELATION*. If (NARGS *MAPFN*) is 1, then *MAPFN* is called on every item which has the given *RELATION* to *any* other item.

(MSNEEDUNSAVE *FNS MSG MARKCHANGEFLG*) [Function]

Used to mark functions which depend on a changed record declaration (or macro, etc.), and which must be LOADed or UNSAVEd (see below). *FNS* is a list of functions to be marked, and *MSG* is a string describing the records, macros, etc. on which they depend. If *MARKCHANGEFLG* is non-NIL, each function in the list is marked as needing reanalysis.

(UPDATEFN *FN EVENIFVALID* —) [Function]

Equivalent to the command ANALYZE '*FN*; that is, UPDATEFN will analyze *FN* if *FN* has not been analyzed before or if it has been changed since the time it was analyzed. If *EVENIFVALID* is non-NIL, UPDATEFN will reanalyze *FN* even if MasterScope thinks it has a valid analysis in the data base.

(UPDATECHANGED) [Function]

Performs (UPDATEFN *FN*) on every function which has been marked as changed.

(MSMARKCHANGED *NAME TYPE REASON*) [Function]

Mark that *NAME* has been changed and needs to be reanalyzed. See MARKASCHANGED in the *IRM*.

(DUMPDATABASE *FNLST*) [Function]

Dumps the current MasterScope data base on the current output file in a LOADable form. If *FNLST* is not NIL, DUMPDATABASE will only dump the information for the list of functions in *FNLST*. The variable DATABASECOMS is initialized to

```
((E (DUMPDATABASE)))
```

Thus, you may merely perform (MAKEFILE 'DATABASE.*EXTENSION*) to save the current MasterScope data base. If a MasterScope data base already exists when a DATABASE file is loaded, the data base on the file will be merged with the one in memory.

Note: Functions whose definitions are different from their definition when the data base was made must be REANALYZEd if their new definitions are to be noticed.

Note: The DataBaseFns library module provides a more convenient way of saving data bases along with the source files to which they correspond.

Noticing Changes that Require Recompiling

When a record declaration, iterative statement operator or macro is changed, and MasterScope has noticed a use of that declaration or macro (i.e. it is used by some function known about in the data base), MasterScope will alert you about those functions which might need to be recompiled (e.g. they do not

currently have EXPR definitions). Extra functions may be noticed.

For example if FOO contains (fetch (REC X) --), and some declaration other than REC which contains X is changed, MasterScope will still think that FOO needs to be loaded/unsaved. The functions which need recompiling are added to the list MSNEEDUNSAVE and a message is printed out:

The functions *FN1, FN2,...* use macros which have changed.

Call UNSAVEFNS() to load and/or unsave them.

In this situation, the following function is useful:

(UNSAVEFNS —) [Function]

Uses LOADFNS or UNSAVEDEF to make sure that all functions in the list MSNEEDUNSAVE have EXPR definitions, and then sets MSNEEDUNSAVE to NIL.

Note: If RECOMPILEDEFAULT (see *IRM*) is set to CHANGES, UNSAVEFNS prints out

"WARNING: you must set RECOMPILEDEFAULT to EXPRS in order to have these functions recompiled automatically."

Implementation Notes

MasterScope keeps a data base of the relations noticed when functions are analyzed. The relations are intersected to form primitive relationships such that there is little or no overlap of any of the primitives.

For example, the relation SET is stored as the union of SET LOCAL and SET FREE. The BIND relation is divided into BIND AS ARG, BIND AND NOT USE, and SET LOCAL, SMASH LOCAL, etc. Splitting the relations in this manner reduces the size of the data base considerably, to the point where it is reasonable to maintain a MasterScope data base for a large system of functions during a normal debugging session.

Each primitive relationship is stored in a pair of hash tables, one for the forward direction and one for the reverse.

For example, there are two hash tables, USE AS PROPERTY and USED AS PROPERTY. To retrieve the information from the data base, MasterScope performs unions of the hash values.

For example, to answer FOO BINDS WHO, MasterScope will look in all of the tables which make up the BIND relation. The internal representation returned by PARSERELATION is a list of dotted pairs of hash tables. To perform GETRELATION requires only mapping down that list, doing GETHASHs on the appropriate hash tables and UNIONing the result.

Hash tables are used for a variety of reasons: storage space is smaller; it is not necessary to maintain separate lists of which functions have been analyzed (a special table, DOESN'T DO ANYTHING is maintained for functions which neither call other functions nor bind or use any variables); and accessing is relatively fast. Within any of the tables, if the hash value is a list of one atom, then the atom itself, rather than the list, is stored as the hash value. This also reduces the size of the data base significantly.

Example

Sample Session

The following illustrates some of the MasterScope facilities.

```
50_. ANALYZE FUNCTIONS ON RECORD
.............................
NIL
51_. WHO CALLS RECFIELDLOOK
(RECFIELDLOOK ACCESSDEF ACCESSDEF2 EDITREC)
52_. EDIT WHERE ANY CALL RECFIELDLOOK
RECFIELDLOOK :
(RECFIELDLOOK (CDR Y) FIELD)
tty:
5*OK
ACCESSDEF :
(RECFIELDLOOK DECLST FIELD VAR1)
6*OK
(RECFIELDLOOK USERRECLST FIELD)
7*N VAR1
8*OK
ACCESSDEF2 :
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
tty:
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
9*N (CAR TAIL]
10*OK
EDITREC :
(RECFIELDLOOK USERRECLST (CAR EDITRECX))
11*OK
NIL
53_. WHO CALLS ERROR
..
(EDITREC)
54_. SHOW PATHS TO RECFIELDLOOK FROM ACCESSDEF
(inverted tree)
```

```
1. RECFIELDLOOK RECFIELDLOOK
2.                    ACCESSDEF
3.                    ACCESSDEF2 ACCESSDEF2
4.                                        ACCESSDEF
5.
RECORDCHAIN ACCESSDEF
NIL
55_. WHO CALLS WHO IN /FNS
RECORDSTATEMENT --  /RPLNODE
RECORDECL1 --       /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 --    /PUTHASH
UNCLISPTRAN --      /PUTHASH, /RPLNODE2
RECORDWORD --       /RPLACA
RECORD1 --          /RPLACA, /SETTOPVAL
EDITREC --          /SETTOPVAL
```

Event 50 You direct that the functions on file RECORD be analyzed. The leading period and space specify that this line is a MasterScope command. MasterScope prints a greeting and prompts with __. Within the top-level Executive of MasterScope, you may issue MasterScope commands, programmer's assistant commands, (e.g., REDO, FIX), or run programs. You can exit from the MasterScope Executive by typing OK. The function "." is defined as a Nlambda NoSpread function which interprets its argument as a MasterScope command, Executes the command and returns.

MasterScope prints a"." whenever it (re)analyzes a function, to let you know what it is happening. The feedback when MasterScope analyzes a function is controlled by the flag MSPRINTFLG: if MSPRINTFLG is the atom ".", MasterScope will print out a period. (If an error in the function is detected, "?" is printed instead.) If MSPRINTFLG is a number *N*, MasterScope will print the name of the function it is analyzing every *N*th function. If MSPRINTFLG is NIL, MasterScope won't print anything. Initial setting is ".".

Note that the function name is printed when MasterScope starts analyzing, and the comma is printed when it finishes.

Event 51 You ask which functions call RECFIELDLOOK. MasterScope responds with the list.

Statement 52 You ask to edit the expressions where the function RECFIELDLOOK is called. MasterScope calls EDITF on the functions it had analyzed that call RECFIELDLOOK, directing the editor to the appropriate expressions. You then edit some of those expressions. In this example, the teletype editor is used. If DEdit is enabled as the primary editor, it would be called to edit the appropriate functions.

Statement 53 Next you ask which functions call ERROR. Since some of the functions in the data base have been changed, MasterScope reanalyzes the changed definitions (and prints out .'s for each function it analyzes). MasterScope responds that EDITREC is the only analyzed function that calls ERROR.

Statement 54 — You ask to see a map of the ways in which RECFIELDLOOK is called from ACCESSDEF. A tree structure of the calls is displayed.

Statement 55 — You then ask to see which functions call which functions in the list /FNS. MasterScope responds with a structured printout of these relations.

SHOW PATHS

The command SHOW PATHS FROM MSPARSE will print out the structure of MasterScope's parser:

```
1.MSPARSE  MSINIT MSMARKINVALID
2.         |      MSINITH MSINITH
3.         MSINTERPRET MSRECORDFILE
4.         |           MSPRINTWORDS
5.         |           PARSECOMMAND GETNEXTWORD CHECKADV
6.         |           |            PARSERELATION {a}
7.         |           |            PARSESET {b}
8.         |           |            PARSEOPTIONS {c}
9.         |           |            MERGECONJ GETNEXTWORD
{5}
10.        |           GETNEXTWORD {5}
11.        |           FIXUPTYPES SUBJTYPE
12.        |           |          OBJTYPE
13.        |           FIXUPCONJUNCTIONS MERGECONJ {9}
14.        |                             MATCHSCORE
15.        MSPRINTSENTENCE
-------------------------------------------------------
overflow - a
16.PARSERELATION GETNEXTWORD {5}
17.              CHECKADV
-------------------------------------------------------
overflow - b
19.PARSESET PARSESET
20.         GETNEXTWORD {5}
21.         PARSERELATION {6}
22.         SUBPARSE GETNEXTWORD {5}
-------------------------------------------------------
overflow - c
23.PARSEOPTIONS GETNEXTWORD {5}
24.             PARSESET {19}
```

This example shows that the function MSPARSE calls MSINIT, MSINTERPRET, and MSPRINTSENTENCE. MSINTERPRET in turn calls MSRECORDFILE, MSPRINTWORDS, PARSECOMMAND, GETNEXTWORD, FIXUPTYPES, and FIXUPCONJUNCTIONS. The numbers in braces {} after a function name are backward references: they indicate that the tree for that function was expanded on a previous line. The lowercase letters in braces are forward references: they indicate that the tree for that function

will be expanded below, since there is no more room on the line. The vertical bar is used to keep the output aligned.

Match provides a fairly general pattern match facility that allows you to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match.

Essentially, you write "Does the (expression) X look like (the pattern) P?"

For example, (MATCH X WITH (& 'A -- 'B)) asks whether the second element of X is an A, and the last element a B.

Requirements

DWIM must be enabled.

Installation

Load MATCH.LCOM from the library.

Programmer's Interface

(MATCH *OBJECT* WITH *PATTERN*) [CLISP operator]

Matches the *OBJECT* with the *PATTERN*.

The implementation of the matching is performed by computing (once) the equivalent Lisp expression which will perform the indicated operation, and substituting this for the pattern (rather than by invoking each time a general purpose capability such as that found in the AI languages FLIP or PLANNER).

For example, the translation of

```
(MATCH X WITH (& 'A -- 'B)) is:
(AND (EQ (CADR X) 'A)
     (EQ (CAR (LAST (CDDR X))) 'B))
```

Thus the pattern match facility is really a pattern match compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. Wherever possible, already existing Lisp functions are used in the translation, e.g., the translation of ($ 'A $) uses MEMB, ($ ('A $) $) uses ASSOC, etc.

The syntax for pattern match expressions is (MATCH *FORM* WITH *PATTERN*), where *PATTERN* is a list as described below. If *FORM* appears more than once in the translation, and it is not either a variable or an expression that is easy to (re)compute, such as

(CAR Y), (CDDR Z), etc., a dummy variable will be generated and bound to the value of *FORM* so that *FORM* is not evaluated a multiple number of times.

For example, the translation of
`(MATCH (FOO X) WITH ($ 'A $))` is simply
`(MEMB 'A (FOO X))`,
while the translation of
`(MATCH (FOO X) WITH ('A 'B --))` is:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CAR (SETQ $$2 (FOO X))) 'A)
           (EQ (CADR $$2) 'B]
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in NIL, i.e., there are no LISTP checks inserted in the translation to check tails.

For example, the translation of
`(MATCH X WITH ('A & --))` is
`(AND (EQ (CAR X) (QUOTE A)) (CDR X))`,
which will match with (A B) as well as (A . B).

Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g.,
`(MATCH X WITH (('A --) --))` translates simply as
`(EQ (CAAR X) 'A)`,
and
`(MATCH X WITH (($1 $1 --) --))` translates as
`(CDAR X)`.

Note that you can explicitly insert LISTP checks yourself by using @, as described below, e.g.,

`(MATCH X WITH (($1 $1 --)@LISTP --))` translates as
`(CDR (LISTP (CAR X)))`.

PATLISPCHECK [Variable]

The insertion of LISTP checks for *ELEMENTS* is controlled by the variable PATLISTPCHECK. When PATLISTPCHECK is T, LISTP checks are inserted, e.g.,

`(MATCH X WITH (('A --) --))` translates as:
`(EQ (CAR (LISTP (CAR (LISTP X)))) 'A)`.

PATLISTPCHECK is initially NIL. Its value can be changed within a particular function by using a local CLISP declaration (see *IRM*).

PATVARDEFAULT [Variable]

Controls the treatment of !*ATOM* patterns (see below).

If PATVARDEFAULT is ' or QUOTE, !*ATOM* is treated the same as '*ATOM*.

If PATVARDEFAULT is = or EQUAL, same as = *ATOM*.

If PATVARDEFAULT is = = or EQ, same as = = *ATOM*.

If PATVARDEFAULT is _ or SETQ, same as *ATOM* _ &.

PATVARDEFAULT is initially ' (quote).

PATVARDEFAULT can be changed within a particular function by using a local CLISP declaration (see *IRM*).

Note: Numbers and strings are always interpreted as though PATVARDEFAULT were =, regardless of its setting. EQ, MEMB, and ASSOC are used for comparisons involving small integers.

Note: Pattern match expressions are translated using the DWIM and CLISP facilities, using all CLISP declarations in effect (standard/fast/undoable; see *IRM*).

Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment.

For example, in the TTY editor's pattern matcher (see *IRM*), "--" matches any arbitrary segment of a list, while & or a subpattern match only one element of a list. Those patterns which may match a segment of a list are called segment patterns; those that match a single element are called element patterns.

Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or & — Matches an arbitrary element of a list.

'*EXPRESSION* — Matches only an element which is equal to the given expression e.g., 'A, '(A B).

EQ, MEMB, and ASSOC are automatically used in the translation when the quoted expression is atomic, otherwise EQUAL, MEMBER, and SASSOC.

=*FORM* — Matches only an element which is EQUAL to the value of *FORM*; e.g., =X, =(REVERSE Y).

= =*FORM* — Same as =, but uses an EQ check instead of EQUAL.

ATOM — The treatment depends on setting of PATVARDEFAULT (see above).

(*PATTERN1 ... PATTERNn*) — Matches a list which matches the given patterns; e.g., (& &), (-- 'A).

ELEMENT-PATTERN@FN — Matches an element if *ELEMENT-PATTERN* matches it, and *FN* (name of a function or a LAMBDA expression) applied to that element returns non-NIL.

For example, &@NUMBERP matches a number, and ('A --)@FOO matches a list whose first element is A and for which FOO applied to that list is non-NIL.

For simple tests, the function-object is applied before a match is attempted with the pattern, e.g.,

((-- 'A --)@LISTP --) translates as
(AND (LISTP (CAR X)) (MEMB 'A (CAR X))),
not the other way around. *FN* may also be a *FORM* in terms of the variable @, e.g., &@(EQ @ 3) is equivalent to = 3.

* Matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-NIL if the match succeeds and NIL if it fails. However, if a * appears in the pattern, the expression generated could also return NIL if the match succeeds and * was matched to NIL.

For example,
(MATCH X WITH ('A * --)) translates as
(AND (EQ (CAR X) 'A) (CADR X)),
so if X is equal to (A NIL B) then (MATCH X WITH ('A * --)) returns NIL even though the match succeeded.

~*ELEMENT-PATTERN* Matches an element if the element is not (~) matched by *ELEMENT-PATTERN*, e.g., ~'A, ~ = X, ~(-- 'A --).

(*ANY* *ELEMENT-PATTERN ELEMENT-PATTERN* ...)

Matches if any of the contained patterns match.

Segment Patterns

$ or -- Matches any segment of a list (including one of zero length).

The difference between $ and -- is in the type of search they generate.

For example,
(MATCH X WITH ($ 'A 'B $)) translates as
(EQ (CADR (MEMB 'A X)) 'B), whereas
(MATCH X WITH (-- 'A 'B $)) translates as:

```
[SOME X (FUNCTION (LAMBDA ($$2 $$1)
  (AND (EQ $$2 'A)
       (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of ($ 'A 'B $) would be "Is B the element following the first A?", whereas a paraphrase of (-- 'A 'B $) would be "Is there any A immediately followed by a B?"

Note that the pattern employing $ will result in a more efficient search than that employing --. However, ($ 'A 'B $) will not match with (X Y Z A M O A B C), but (-- 'A 'B $) will.

Essentially, once a pattern following a $ matches, the $ never resumes searching, whereas -- produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both -- and $ will be the same.

For example, both
`(MATCH X WITH ($ 'A $3 $))` and
`(MATCH X WITH (-- 'A $3 --))` translate as
`(CDDDR (MEMB (QUOTE A) X))`
because if there are not three elements following the first A, there certainly will not be three elements following subsequent A's, so there is no reason to continue searching, even for --.

Similarly, ($ 'A $ 'B $) and (-- 'A -- 'B --) are equivalent.

$2, $3, etc. Matches a segment of the given length.

Note that $1 is not a segment pattern.

!*ELEMENT-PATTERN* Matches any segment which *ELEMENT-PATTERN* would match as a list.

For example, if the value of FOO is (A B C), ! = FOO will match the segment ... A B C ... etc.

Note: Since ! appearing in front of the last pattern specifies a match with some tail of the given expression, it also makes sense in this case for a ! to appear in front of a pattern that can only match with an atom, e.g., ($2 !'A) means match if CDDR of the expression is the atom A.

Similarly,

`(MATCH X WITH ($ ! 'A))` translates to
`(EQ (CDR (LAST X)) 'A)`.

!*ATOM* The treatment depends on setting of PATVARDEFAULT.

If PATVARDEFAULT is ' or QUOTE, same as !'*ATOM* (see above discussion).

If PATVARDEFAULT is = or EQUAL, same as ! = *ATOM*.

If PATVARDEFAULT is = = or EQ, same as ! = = *ATOM*.

If PATVARDEFAULT is _ or SETQ, same as *ATOM*_$.

The atom "." is treated *exactly* like "!". In addition, if a pattern ends in an atom, the "." is first changed to "!", e.g., ($1 . A) and ($1 ! A) are equivalent, even though the atom "." does not explicitly appear in the pattern.

One exception where "." is not treated like "!" is when "." preceding an assignment does not have the special interpretation that "!" has preceding an assignment (see below).

For example,

`(MATCH X WITH ('A . FOO_'B))` translates as:

```
(AND (EQ (CAR X) 'A)
     (EQ (CDR X) 'B)
     (SETQ FOO (CDR X)))
```

but

```
(MATCH X WITH ('A ! FOO_'B)) translates as:
(AND (EQ (CAR X) 'A)
     (NULL (CDDR X))
     (EQ (CADR X) 'B)
     (SETQ FOO (CDR X)))
```

SEGMENT-PATTERN@FUNCTION-OBJECT Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL.

For example, ($@CDDR 'D $) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: An @ pattern applied to a segment will require computing the corresponding structure (with LDIFF) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

Assignments

Any pattern element may be preceded by "*VARIABLE*_", meaning that if the match succeeds (i.e., everything matches), *VARIABLE* is to be set to the thing that matches that pattern element.

For example, if X is (A B C D E), (MATCH X WITH ($2 Y_$3)) will set Y to (C D E).

Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match, e.g., (MATCH X WITH (Y_$1 = Y --)) will not match with (A A B C ...), unless, of course, the value of Y was A before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a !, the assignment is to the tail of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern.

For example, if X is (A B C D E), (MATCH X WITH ($!Y_'C 'D $)) sets Y to (C D E), i.e., CDDR of X. In other words, when ! precedes an assignment, it acts as a modifier to the _, and has no effect whatsoever on the pattern itself, e.g., (MATCH X WITH ('A 'B)) and (MATCH X WITH ('A !FOO_'B)) match identically, and in the latter case, FOO will be set to CDR of X.

Note: *_*PATTERN-ELEMENT* and !*_*PATTERN-ELEMENT* are acceptable, e.g.,

```
(MATCH X WITH ($ 'A *_('B --) --)) translates as:
[PROG ($$2) (RETURN
 (AND (EQ (CAADR (SETQ $$2 (MEMB 'A X)))  'B)
      (CADR $$2]
```

Place Markers

Variables of the form #*N*, where *N* is a number, are called place markers, and are interpreted specially by the pattern match compiler. Place markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern.

For example,

```
(MATCH X WITH (#1_$1 =(ADD1 #1)))
```

will match the list (2 3).

However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of PATVARDEFAULT, the first appearance of a defaulted place-marker is interpreted as though PATVARDEFAULT were _.

Thus the above pattern could have been written as

```
(MATCH X WITH ( 1 =(ADD1  1))).
```

Subsequent appearances of a place-marker are interpreted as though PATVARDEFAULT were =.

For example,

`(MATCH X WITH (#1 #1 --))` is equivalent to
`(MATCH X WITH (#1_$1 =#1 --))`, and translates as
`(AND (CDR X) (EQUAL (CAR X) (CADR X)))`.

Note that (EQUAL (CAR X) (CADR X)) would incorrectly match with (NIL).

Replacements

The construct *PATTERN-ELEMENT* _ *FORM* specifies that if the match succeeds, the part of the data that matched is to be replaced with the value of *FORM*.

For example, if X = (A B C D E), (MATCH X WITH ($ 'C $1 _ Y $1)) will replace the third element of X with the value of Y. As with assignments, replacements are not performed until after it is determined that the entire match will be successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if X is (A B C D E F) and FOO is (1 2 3), after the pattern ('A $ _ FOO 'D $) is matched with X, X will be (A 1 2 3 D E F), and FOO will be EQ to CDR of X, i.e., (1 2 3 D E F).

Note that ($ FOO _ FIE $) is ambiguous, since it is not clear whether FOO or FIE is the pattern element, i.e., whether _ specifies assignment or replacement.

For example, if PATVARDEFAULT is =, this pattern can be interpreted as ($ FOO _ = FIE $), meaning search for the value of

FIE, and if found set FOO to it, or ($ =FOO_FIE $) meaning search for the value of FOO, and if found, store the value of FIE into the corresponding position. In such cases, you should disambiguate by not using the PATVARDEFAULT option, i.e., by specifying ' or = .

Note: Replacements are normally done with RPLACA or RPLACD. You can specify that /RPLACA and /RPLACD should be used, or FRPLACA and FRPLACD, by means of CLISP declarations (see *IRM*).

Reconstruction

You can specify a value for a pattern match operation other than what is returned by the match by writing (MATCH *FORM1* WITH *PATTERN* = > *FORM2*).

For example,

```
(MATCH X WITH (FOO_$ 'A --) => (REVERSE FOO))
```

translates as:

```
[PROG ($$2)
   (RETURN
      (COND ((SETQ $$2 (MEMB 'A X))
             (SETQ FOO (LDIFF X $2))
             (REVERSE FOO]
```

Place markers in the pattern can be referred to from within *FORM*, e.g., the above could also have been written as

```
(MATCH X WITH (!#1 'A --) => (REVERSE #1)).
```

If -> is used in place of = >, the expression being matched is also physically changed to the value of *FORM*.

For example,

```
(MATCH X WITH (#1 'A !#2) -> (CONS #1 #2))
```

would remove the second element from X, if it were equal to A.

In general, (MATCH *FORM1* WITH *PATTERN* -> *FORM2*) is translated so as to compute *FORM2* if the match is successful, and then smash its value into the first node of *FORM1*. However, whenever possible, the translation does not actually require *FORM2* to be computed in its entirety, but instead the pattern match compiler uses *FORM2* as an indication of what should be done to *FORM1*.

For example,

```
(MATCH X WITH (#1 'A !#2) -> (CONS #1 #2))
```

translates as

```
(AND (EQ (CADR X) 'A) (RPLACD X (CDDR X))).
```

Limitations

The pattern match facility does not contain some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, you can be confident that what facilities it does provide will result in Lisp expressions comparable to those you would generate by hand.

Examples

```
(MATCH X WITH (-- 'A --))
```

-- matches any arbitrary segment. 'A matches only an A, and the second -- again matches an arbitrary segment; thus this translates to (MEMB 'A X).

```
(MATCH X WITH (-- 'A))
```

Again, -- matches an arbitrary segment; however, since there is no -- after the 'A, A must be the last element of X. Thus this translates to: `(EQ (CAR (LAST X)) 'A)`.

```
(MATCH X WITH ('A 'B -- 'C $3 --))
```

CAR of X must be A, and CADR must be B, and there must be at least three elements after the first C, so the translation is:

```
(AND (EQ (CAR X) 'A)
     (EQ (CADR X) 'B)
     (CDDDR (MEMB 'C (CDDR X))))
```

```
(MATCH X WITH (('A 'B) 'C Y_$1 $))
```

Since ('A 'B) does not end in $ or --, (CDDAR X) must be NIL. The translation is:

```
(COND
   ((AND (EQ (CAAR X) 'A)
         (EQ (CADAR X) 'B)
         (NULL (CDDAR X))
         (EQ (CADR X) 'C)
         (CDDR X))
    (SETQ Y (CADDR X)) T))
```

```
(MATCH X WITH (#1 'A $ 'B 'C #1 $))
```

#1 is implicitly assigned to the first element in the list. The $ searches for the first B following A. This B must be followed by a C, and the C by an expression equal to the first element. The translation is:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CADR X) 'A)
           (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
           (CDDR $$2)
           (EQUAL (CADDR $$2) (CAR X]

(MATCH X WITH (#1 'A -- 'B 'C #1 $))
```

Similar to the pattern above, except that -- specifies a search for *any* B followed by a C followed by the first element, so the translation is:

```
[AND (EQ (CADR X) 'A)
     (SOME (CDDR X)
           (FUNCTION (LAMBDA ($$2 $$1)
              (AND (EQ $$2 'B)
                   (EQ (CADR $$1) 'C)
                   (CDDR $$1)
                   (EQUAL (CADDR $$1) (CAR X]
```

Two dimensional graphical transformations, such as rotations, scalings, and translations are conveniently represented as homogeneous 3-by-3 matrices, which operate on homogeneous 3-vectors. Similarly, three dimensional graphical transformations are conveniently represented as homogeneous 4-by-4 matrices, which operate on homogeneous 4-vectors. MatMult provides utilities for creating and manipulating such matrices and vectors, and takes advantage of microcode support for high-speed 3-by-3 and 4-by-4 matrix multiplication.

All matrices and vectors in MatMult are represented as Common Lisp arrays of element type single-float, so the Common Lisp array functions are sufficient to create and access individual elements of these specialized arrays. However, MatMult provides convenient wrapper functions for most common operations on these arrays.

All the following functions that return arrays accept optional array arguments. If given a result argument, these functions alter the contents of that argument rather then allocating new storage. It is an error for the optional array argument to be not of element type single-float, or to have incorrect dimensions.

Requirements

MatMult should be run on an 1109 with a Weitek floating point chip set, but is also quite efficient on an 1186.

Installation

Load MATMULT.LCOM from the library.

Matrix Creation Functions

(MAKE-HOMOGENEOUS-3-VECTOR *X Y*) [Function]

Returns a 3-vector of element type single-float. If *X* or *Y* is provided, then the corresponding element of the vector is set appropriately, otherwise it defaults to 0.0. The third element of the vector is always initialized to 1.0.

Note: Throughout this text, "set" is used to emphasize that the value of the result element is altered and that no new storage is allocated to it.

(MAKE-HOMOGENEOUS-3-BY-3 *&KEY A00 A01 A10 A20 A21*) [Function]

Returns a 3-by-3 matrix of element type single-float. If a keyword argument is provided, the corresponding element of the matrix

is set appropriately, otherwise entries default to 0.0. The (2 ,2) is always initialized to 1.0.

(MAKE-HOMOGENEOUS-N-BY-3 *N &KEY INITIAL-ELEMENT*) [Function]

Returns an N-by-3 matrix of element type single-float. If the keyword argument is provided, all the elements in the first two columns are set appropriately, otherwise they default to 0.0. The third column is always initialized to 1.0.

(MAKE-HOMOGENEOUS-4-VECTOR *X Y Z*) [Function]

Returns a 4-vector of element type single-float. If *X*, *Y* or *Z* is provided then the corresponding element of the vector is set appropriately, otherwise it defaults to 0.0. The forth element of the vector is always initialized to 1.0.

(MAKE-HOMOGENEOUS-4-BY-4 *&KEY A00 A01 A02 A03 A10 A11 A12 A13 A20 A21 A22 A23 A30 A31 A32* [Function]

Returns a 4-by-4 matrix of element type single-float. If a keyword arguments is provided, the corresponding element of the matrix is set appropriately, otherwise entries default to 0.0. The (3 ,3) is always initialized to 1.0.

(MAKE-HOMOGENEOUS-N-BY-4 *N &KEY INITIAL-ELEMENT*) [Function]

Returns an N-by-4 matrix of element type single-float. If the keyword argument is provided, all the elements in the first three columns are set appropriately, otherwise they default to 0.0. The forth column is always initialized to 1.0.

(IDENTITY-3-BY-3 *RESULT*) [Function]

Returns a 3-by-3 identity matrix.

If *RESULT* is supplied, it is side effected and returned.

(That is, the storage associated with the optional result argument is reused for the result, rather than allocating new storage for the result.)

(IDENTITY-4-BY-4 *RESULT*) [Function]

Returns a 4-by-4 identity matrix. If *RESULT* is supplied, it is side effected and returned.

(ROTATE-3-BY-3 *RADIANS RESULT*) [Function]

Returns a 3-by-3 rotation matrix specified by a counter-clockwise rotation of *RADIANS* radians. If *RESULT* is supplied, it is set and returned.

(ROTATE-4-BY-4-ABOUT-X *RADIANS RESULT*) [Function]

Returns a 4-by-4 rotation matrix specified by a positive right-handed rotation of *RADIANS* radians about the X axis. If *RESULT* is supplied, it is set and returned.

(ROTATE-4-BY-4-ABOUT-Y *RADIANS RESULT*) [Function]

Returns a 4-by-4 rotation matrix specified by a positive right-handed rotation of *RADIANS* radians about the Y axis. If *RESULT* is supplied, it is set and returned.

(ROTATE-4-BY-4-ABOUT-Z *RADIANS RESULT*) [Function]

Returns a 4-by-4 rotation matrix specified by a positive right-handed rotation of *RADIANS* radians about the Z axis. If *RESULT* is supplied, it is set and returned.

(SCALE-3-BY-3 *SX SY RESULT*) [Function]

Returns a 3-by-3 homogeneous scaling transformation that scales by a factor of *SX* along the X-axis and *SY* along the Y-axis. If *RESULT* is supplied, it is set and returned.

(SCALE-4-BY-4 *SX SY SZ RESULT*) [Function]

Returns a 4-by-4 homogeneous scaling transformation that scales by a factor of *SX* along the X-axis, *SY* along the Y-axis, and *SZ* along the Z axis. If *RESULT* is supplied, it is set and returned.

(TRANSLATE-3-BY-3 *TX TY RESULT*) [Function]

Returns a 3-by-3 homogeneous translation that translates by *TX* along the X-axis and *TY* along the Y-axis. If *RESULT* is supplied, it is set and returned.

(TRANSLATE-4-BY-4 *TX TY TZ RESULT*) [Function]

Returns a 4-by-4 homogeneous translation that translates by *TX* along the X-axis, *TY* along the Y-axis and *TZ* along the Z axis. If *RESULT* is supplied, it is set and returned.

(PERSPECTIVE-4-BY-4 *PX PY PZ RESULT*) [Function]

Returns a 4-by-4 homogeneous perspective transformation defined by *PX, PY*, and *PZ*. If *RESULT* is supplied, it is set and returned.

Matrix Multiplication Functions

If run on workstations equipped with the extended processor option, these functions make good use of the hardware floating-point unit. The three digits at the end of each function's name describe the dimensions of their arguments.

Note: The results of the following matrix multiplication functions are not guaranteed to be correct unless the matrix arguments are all different (Not EQ).

(MATMULT-133 *VECTOR MATRIX RESULT*) [Function]

Returns the inner product of a 3-vector, *VECTOR*, and a 3-by-3 matrix, *MATRIX*. If *RESULT* is supplied, it is set and returned.

(MATMULT-331 *MATRIX VECTOR RESULT*) [Function]

Returns the inner product of a 3-by-3 matrix, *MATRIX*, and a 3-vector, *VECTOR*. If *RESULT* is supplied, it is set and returned.

(MATMULT-333 *MATRIX-1 MATRIX-2 RESULT*) [Function]

Returns the inner product of a 3-by-3 matrix, *MATRIX-1*, and another 3-by-3 matrix, *MATRIX-2*. If *RESULT* is supplied, it is set and returned.

(MATMULT-N33 *MATRIX-1 MATRIX-2 RESULT*) [Function]

Returns the inner product of an N-by-3 matrix, *MATRIX-1*, and a 3-by-3 matrix, *MATRIX-2*. If *RESULT* is supplied, it is set and returned.

(MATMULT-144 *VECTOR MATRIX RESULT*) [Function]

Returns the inner product of a 4-vector, *VECTOR*, and a 4-by-4 matrix, *MATRIX*. If *RESULT* is supplied, it is set and returned.

(MATMULT-441 *MATRIX VECTOR RESULT*) [Function]

Returns the inner product of a 4-by-4 matrix, *MATRIX*, and a 4-vector, *VECTOR*. If *RESULT* is supplied, it is set and returned.

(MATMULT-444 *MATRIX-1 MATRIX-2 RESULT*) [Function]

Returns the inner product of a 4-by-4 matrix, *MATRIX-1*, and another 4-by-4 matrix, *MATRIX-2*. If *RESULT* is supplied, it is set and returned.

(MATMULT-N44 *MATRIX-1 MATRIX-2 RESULT*) [Function]

Returns the inner product of an N-by-4 matrix, *MATRIX-1*, and a 4-by-4 matrix, MATRIX-2. If *RESULT* is supplied, it is set and returned.

Miscellaneous Functions

(PROJECT-AND-FIX-3-VECTOR *3-VECTOR 2-VECTOR*) [Function]

The homogeneous *3-VECTOR* is projected onto the X-Y plane, coerced to integer coordinates (rounding by truncation) and returned. If *2-VECTOR* is supplied, it is set and returned.

(PROJECT-AND-FIX-N-BY-3 *N-3-MATRIX N-2-MATRIX*) [Function]

The homogeneous N-by-3 matrix, *N-3-MATRIX*, is projected onto the X-Y plane row-by-row, coerced to integer coordinates (rounding by truncation) and returned. If *N-2-MATRIX* is supplied, it is set and returned.

(PROJECT-AND-FIX-4-VECTOR *4-VECTOR 2-VECTOR*) [Function]

The homogeneous 4-vector, *4-VECTOR*, is projected onto the X-Y plane, coerced to integer coordinates (rounding by truncation) and returned. If *2-VECTOR* is supplied, it is set and returned.

(PROJECT-AND-FIX-N-BY-4 *N-4-MATRIX N-2-MATRIX*) [Function]

The homogeneous N-by-4 MATRIX, *N-3-MATRIX*, is projected onto the X-Y plane row-by-row, coerced to integer coordinates (rounding by truncation) and returned. If *N-2-MATRIX* is supplied, it is set and returned.

(DEGREES-TO-RADIANS *DEGREES*) [Function]

Returns *DEGREES* converted to radians.

Limitations

MatMult is not intended as a general matrix manipulation package; it is specialized for the 3-by-3 and 4-by-4 cases.

Use CmlFloatArray for more general floating point array facilities.

Example

```
(* ; "Try (spiral)")

(CL:DEFUN SPIRAL (&OPTIONAL (WINDOW (CREATEW))
                        &AUX
                        (WIDTH (WINDOWPROP WINDOW 'WIDTH))
                        (HALF-WIDTH (QUOTIENT WIDTH 2))
                        (HEIGHT (WINDOWPROP WINDOW 'HEIGHT))
                        (HALF-HEIGHT (QUOTIENT HEIGHT 2))
                        (SCALE-FACTOR (CL:EXP (QUOTIENT
                             (CL:LOG (QUOTIENT (MIN WIDTH HEIGHT) 2.0))
1440.0))))
   (LET ((LINE-1 (MAKE-HOMOGENEOUS-3-VECTOR 1.0 0.0))
         (LINE-2 (MAKE-HOMOGENEOUS-3-VECTOR))
         (TEMP (MAKE-HOMOGENEOUS-3-VECTOR))
         (POINTS (CL:MAKE-ARRAY 2))
         (TRANSFORM (MATMULT-333 (ROTATE-3-BY-3 (DEGREES-TO-RADIANS 2.5))
                           (SCALE-3-BY-3 SCALE-FACTOR SCALE-FACTOR)))
         (TRANSLATION (TRANSLATE-3-BY-3 HALF-WIDTH HALF-HEIGHT)))
        (CL:DO ((L-1 LINE-1)
                (L-2 LINE-2)
                (I 0 (CL:1+ I)))
               ((EQ I 1728))
               (MATMULT-133 L-1 TRANSFORM L-2)
               (MATMULT-133 L-2 TRANSLATION TEMP)
               (PROJECT-AND-FIX-3-VECTOR TEMP POINTS)
               (DRAWLINE HALF-WIDTH HALF-HEIGHT (CL:AREF POINTS 0)
                      (CL:AREF POINTS 1)
                      1
                      'REPLACE WINDOW)
               (CL:ROTATEF L-1 L-2))))
```

[This page intentionally left blank]

MiniServe contains servers for three simple protocols: Time Service (both PUP and XNS versions) and PUP ID Service. The servers are intended to run in the background on an 1108 or 1186 on networks that lack other sources of these services.

Requirements

The time must be correctly set on the machine running MiniServe (see "NS Time Service" below).

Installation

Load MINISERVE.LCOM from the library.

Either set the variable NS.TO.PUP.ALIST correctly, or make sure that the variable NS.TO.PUP.FILE is the name of a file containing a single form which will be used to set NS.TO.PUP.ALIST (see "PUP ID Service" below).

Evaluate (STARTMINISERVER).

Functions

(STARTMINISERVE) [Function]

This function has no arguments; it adds three background processes to the environment, one for each of the protocols that miniserve handles. These processes and protocols are:

\NSTIMESERVER	Provides the XNS Time Service
\PUPTIMESERVER	Provides the PUP Time Service
\PUP.ID.SERVER	Provides the PUP ID Service

XNS Time Service

XNS Time Service answers requests for the time using the XNS Time Protocol.

You must already have set the correct date and time on your workstation, either via one of the installation utilities or by evaluating

```
(SETTIME "dd-MMM-yy hh:mm:ss").
```

If you are not in the Pacific time zone, you should also make sure the following variables are set correctly:

\BEGINDST [Variable]

The ordinal day of the year (1 = January 1, 366 = December 31) on or before which daylight saving time starts in your area. Set it to 367 if your area does not observe daylight saving time.

\ENDDST [Variable]

The ordinal day of the year on or before which daylight saving time ends.

\TIMEZONECOMP [Variable]

The number of hours west of Greenwich; e.g., Eastern standard time = 5.

PUP Time Service

PUP Time Service is like NS Time Service, but using a PUP protocol. This service is not required by any Xerox workstation as long as XNS Time Service is available, but may be of use to other workstations.

You can disable it by evaluating

```
(MOVD 'NILL '\PUPTIMESERVER).
```

PUP ID Service

PUP ID Service supplies workstations with PUP host numbers, given their 48-bit XNS host numbers, so that they may communicate via PUP protocols.

NS.TO.PUP.FILE [Variable]

The name of a file containing a single form which will be used to set NS.TO.PUP.ALIST. Either this variable or NS.TO.PUP.ALIST must be set for the PUP ID Service to work.

NS.TO.PUP.ALIST [Variable]

A list which maps a workstation's XNS host number to a pup host number. Elements of this list are dotted pairs of the form:

```
((NSHOSTNUMBER A B C) . PUPNUMBER)
```

where *A, B, C* are the three 16-bit components of the workstation's 48-bit XNS host number (the value of the variable \MY.NSHOSTNUMBER), and PUPNUMBER is the corresponding PUP host number to be assigned to the workstation. PUP host numbers are integers in the range [1,254], and must be unique among hosts on a single net.

To set up this list correctly you can do the following on each workstation which will use the service (including the workstation running MiniServe):

1. Decide on a unique PUP host number for this workstation. It must be an integer inthe range [1,254]. For example we'll choose PUP Host number 2.

2. Get the workstation's NS host number and add it to the PUP host number. Evaluate the following form:

   (CONS \MY.NSHOSTNUMBER *YOURPUPNUMBER*)

   Using our chosen PUP host number of "2" and an example value for \MY.NSHOSTNUMBER the result might be:

   ```
   ((NSHOSTNUMBER 0 43520 14312) . 2)
   ```

3. Back on the workstation which is about to run MINISERVE, insert the dotted pair into NS.TO.PUP.ALIST.

Restarting MiniServe

If you need to restart MiniServe:

Use the PSW window to kill the three processes that were started by STARTMINISERVE.

Evaluate (STARTMINISERVE).

[This page intentionally left blank]