MediaMath
An Interactive System for Image and Audio Analysis

Introduction to MediaMath
programming language

Part 1.

Introduction to MediaMath programming language

1. A Gentle Introduction

MediaMath is designed to be easy to use and to look familiar to the user from the first encounter. So it uses the familiar syntax of C (with some omissions and extensions). It is an interpreted language, like Basic and Lisp. This means that you can give it an expression and get the answer right away. This is invaluable for incremental debugging. It is extensible in more than one ways: one can define functions that can be called seamlessly from the interpreter and the other is to include new primitives in C.

The fact that it is an interactive language means that one can execute statements interactively by just typing them in

```
a=3.14;
b=a*2;
c=sin(b);
b=to_integer(b);
```

The result of every statement is printed. Any executable statement can be executed interactively. And when we load a file, MediaMath pretends that it was typed in. If an error occurs then a helpful message is printed and we go back to the top level interpreter. We can see the history of function evaluations that led to the error by typing

```
BackTraceOld;
```

The variable `BackTraceOld` holds the backtrace of the previous error. It is a list of all the function invocations.

1.1. A simple example

The best way to learn a language is by using it. We start with a few examples. Let’s write a routine that adds three to a number.

```
function plus3(x)
     "Adds 3 to x\n\
no matter what is the type of x\n"
{
     x+3;
};
```

We will take every piece of code one by one. First a function definition begins with the word `function`. The name and argument list follow. There are no type declarations for the arguments, because it is a soft typed language. The next two lines contain the documentation string. This string doubles as internal documentation and on line help. After you define this function, this string will be available as help. Just type `?plus3;` and this string will appear. Also notice that the strings in MediaMath are like C strings.

Now we enter the body of the function. There is only one expression here. When the function is called this statement is executed and the result is returned as the value of the function. In general, if there are many expressions in the body then the value of the last one is returned as the value of the function (much like the progn and let of Lisp).

1.2. Another example

Now we can go into a more serious example. A function that computes the exponential of a floating point number. We name it *newexp* to avoid redefinition of the built in *exp*.

```
function newexp(x)
"New version of exp written in MediaMath;\n\
Under Development\n"
{
  local temp, prev, res, i;

  res = 1.0;
  x = to_float(x);   /* if it is not float make it float */
  i = 0;
  prev = -1.0;
  temp = 1.0;
  while (res-prev != 0.0)
    {
      prev = res;
      i++;
      temp *= x/i;
      res += temp;
    };
  printf("iterations: %d\n",i);
  res;
};
```

There are a few new features here. The first is the *local* keyword. It defines a few variables as local so that we don't get conflicts with other routines. The local variables are accessible only from within the function *and* the functions that are called by it. This is what we call *Dynamic Binding*. The formal parameters in the function definition are also local variables in this sense. But bear in mind that dynamic binding is too powerful to be used often. It is better to write programs that do not care what is the binding.

The other feature is the *while* statement. Its behavior is the same as in C except that it returns a value: the value of the last executed statement. There is also a *for* statement for which the same things apply. In the end of the function there is a debugging statement that prints the number of iterations with a call to *printf*, which is modeled after the C standard library function `printf`.

2. Syntax of the language

The syntax is the same as C with few exceptions, to accommodate the different features of the languages. These features are:

- *Functional programming*. MediaMath is a purely functional language so everything returns something, so we can have statements like `3 + while (a>0) a-5`. Even the definition of a function returns something (the symbol of the function).
- *Soft typing*. Variables are not bound to types. So we can have statements like `a=3;a=3.1;`. And of course there are no type declarations for variables. But we can define new types with the `struct` keyword.
- *Objects*. MediaMath has some object oriented features like multi argument dispatch methods, so it has the syntax to deal with them.
- *Variable number of arguments*. We can define functions with variable number of arguments and keywords. One can use `make_complex(3.1,4.2);` or `make_complex(:im=3.1,:re=4.2);`.
- *Image and matrix operators*. There is a variety of operators to use for image and matrix manipulations like `a1 (*) s_templ;` or `m1<->m2;` to convolve an image with a template and concatenate two matrices.
- *No inherited mistakes*. There was no reason to repeat some of the mistakes that happened with C, like the use of caret for `XOR`, the lack of a power operator to do things like 3^2 and the funny way to access 2-d arrays with `A[3][4]`.

Other than that the two languages have the same syntax.

2.1. Expressions

The expressions are composed from operators and operands like `a+1` or `b+c->val`.

2.1.1. Operators

The operators are of three kinds: binary and unary pre and post operators. The binary operators in increasing order of precedence are:

```
=       +=      -=      *=      /=      %=      ..
||
&&
|
^^
&
==      ===     !=      !==     !===
<       >       <=      >=
>>      <<
+       -
*       /       %       (*)     (|)     (-)     <|>     <->
^
```

The associativity of all the above operators is from left to right with the exception of the power operator and the assignment operators. The assignment expressions are a little different than the rest and we deal with them in the next section. All the unary pre-operators have the same precedence (it does not make sense otherwise) and the same for post operators. The precedence though of the post operators is higher than the pre operators. The pre operators are:

```
!          ~          ++          --          -          +          ′
```

and the post operators are:

```
++          --          ^T
```

Some of these operators are new and perform functions specific to MediaMath.

`^` The power operator. The type of the operands can be any number: integer, unsigned integer, character, unsigned character or float. Depending on the type the interpreter will call the appropriate library function to do the job.

`(*)` The 2-D convolution operator. One operand must be an image or a scan line and the other a template or both templates.

`(|)` Vertical convolution. When the template is one dimensional it is treated as a column. This is useful for separable convolutions.

`(-)` Horizontal convolution. As above but the template is treated as a row.

`<|>` Vertical concatenation. Two objects like matrices images etc are stacked one on top of the other.

`<->` Horizontal concatenation. As above, but the concatenation is sideways.

`^T` Transpose. Matrix transpose operator.

`′` Quote. Protects its argument from evaluation. It returns the expression unevaluated. This is an advanced feature.

`..` Range. It returns a data type that indicates range. It is used for subarray access and some other functions that understand it.

As in most languages precedence and associativity can be changed with the use of parentheses.

2.1.2. Operands

Anything can be an operand in this language (there are some restrictions if the operation is assignment and we deal with it later). Anything from variables to type definitions to function calls, because everything returns something. More specificly:

- Variables: The symbols play the role of the variables. A symbol is an entity with several attributes and not just the name of a place in memory. The attributes of a symbol are: name, value and property list. When we type in the name of a variable the interpreter looks it up in the hash table and finds the symbol that corresponds to the name (if this is the first time it sees the name creates a new symbol with function `intern`). Then depending on whether it is on the left or right side of an assignment retrieves or modifies the value field of the symbol. So if we type
  ```
  a=5;
  a+3;
  ```
  then in the first case we modify the value field of `a` and in the second we just access it. If the second expression was evaluated we would have a call to `gplus` with `5` and `3` as

arguments.

- Function calls. A function call is the same as in C with a few more fancy ways to pass variable number of arguments. If the function is defined with fixed number of arguments, all the arguments have to be present in the correct order. If it has optional arguments as well then the required arguments come first and all of them have to be present, and then the optional ones. If there are three optional arguments and we want to specify only the last one we have to either specify the other two as well or use keywords. If we have the function `make_image_info` that accepts one required argument and three optional ones the last of which is `focal_length` then we could use
  `make_image_info(img1, :focal_length=28.2);`
  The keywords can be used also when the function uses a global variable but we want to use something else for this invocation only. For instance if the function `fix_aspect` consults the global variable `aspect_ratio` for the default camera but we want for one call to use the value `new_aspect` we could write `fix_aspect(img1, :aspect_ratio=new_aspect);`.
  There is also a mechanism to pass to a function any number of parameters. The function "sees" a list of values that can be accessed with calls to `car` and `cdr`.
- Array references. There are several kinds of arrays already built in and one can add more. These include the regular arrays, which are heterogeneous arrays, the matrices, the vectors, the templates, the images, the scan lines and the strings. All of them can be accessed with square brackets.
  ```
  het = make_array(5,nil);
  het[1]=1;
  het[2]="Media";
  mat1 = mk_fmat(1..3,1..2,[[1,2],[2,3],[3,4]]);
  1 + mat1[3,2];
  ```
  A very useful feature is that the indices of the arrays can be ranges
  ```
  vec = mat1[1..2,2];
  ```
  so that `vec` gets assigned a vector that contains the first through second elements of the second column of matrix `mat1`.
- Structure references. Structures can be referenced with the `->` operator. So the real value of a complex number is `z->re`. The `->` operator can be used in other situations as well.
- While statements. A while statement as in `5 + while (x>=3) x -= 3;` returns the value of the last expression evaluated, in this case `x` modulo `3` so the value of the whole expression is `5+(x %= 3)`.
- For statements. A for statement also returns a value. For instance `for (i=1; i<=10; i++) ar1[i]=i^2;` will return `100`.
- If statements. An if statement like `if (x>0) x+1 else -x+1;` will return the value of `x+1` if `x` is positive and `-x+1` if `x` is negative. If the else statement is missing then `nil` is returned. Notice that there is no `;` before the `else`.
- Compound statements. A compound statement like `{ local temp; temp = x; x = y; y = temp; };` will return as value the value of the last executed statement, in this

case the original value of `x`. The `local` declaration is optional and can have many variables. Notice that there is a `;` after the closing brace.

- Lists. A list like `[2,2,5,2]` is a collection of possibly heterogeneous items. It is used for advanced programming or for passing arguments to some functions like `mk_fvec`.
- Prefix statements. MediaMath provides the means to write everything in Lisp-like prefix notation. For example, `x = 3*x +1` can be written as `$(set $(quote x) $(gplus $(gmult 3 x) 1))`. The prefix syntax is much more powerful but hard to use. It is intended for advanced programming only.
- Constants. There are a few types of constants. The `nil` and `t` are built in constants. `nil` represents the false value (e.g. `3==4` evaluates to `nil`), the empty list, the null pointer and the default initialization of uninitialized entities and it is a symbol. `t` represents the true value (e.g. `3==3` evaluates to `t`) and it is a symbol. Then are the integers like `45`. And floats like `3.14` or `1.2e-3`. And strings like `"MediaMath Version %d.%d\n"` that are exactly like C strings.
- Function definitions. A function definition returns the symbol that holds the function. For example
```
function my_fun(x, y, &optional z &init 3, &rest rest)
        "just an example"
{
        printf("x is %d, y is %d, z is %d", x, y, z);
        if (rest)
        {
          printf(" and the rest are: ");
          for ( ; rest; rest=cdr(rest))
            printf(", %d",car(rest));
          printf("0);
        }
        else printf(", and no rest0);
};
```
  The value of this function definition is the symbol `my_fun`. The syntax of a function definition is the word `function`, the symbol of the function `my_fun`, a parenthesized list of the formal arguments, an optional documentation string and a compound statement. The list of formal arguments has the following structure. All the required parameters, if any, first. The optional parameters, if any, next. We indicate that where the optional parameters start with the word `&optional`. If an optional parameter needs to be initialized then we put the word `&init` after the name and then the initialization as in `z &init 3`. If there are rest arguments the word `&rest` follows and one symbol (it foes not make sense to declare more than one rest arguments). The rest argument can be initialized in the same way. If there are rest arguments we cannot use keywords.
- Structure definitions. A structure definition looks like a C structure definition, just simpler. For example
```
struct complex
        "Structure for complex entities."
```

```
{
  re = 0;
  im = 0;
};
z1 = make_complex(:re=2);
```
defines a structure with the name `complex`, with two fields that are initialized (if we don't want to initialize then we omit the `= 0`) and a documentation string. After the evaluation of the structure definition the interpreter defines the function `make_complex` automatically. It also defines the function `complex_p` and knows how to access the fields of the structure with the `->` operator.

- Generic function definitions. A generic function is a function that has different behavior depending on the type of the arguments provided at every function call. The function `gmult`, that implements operator `*`, is such an example. When the operands are `float`s it multiplies them as real numbers. When the one is `float` and the other `int` first converts and then multiplies. The same for matrices, images etc. If we want to extend the functionality of the `*` operator to work for `complex` structures as well we can define a few functions to do the job and then tell `gmult` how to call them.
```
function complex_by_complex(x,y)
{
  make_complex(:re = x->re * y->re - x->im * y->im,
               :im = x->re * y->im + x->im * y->re);
};
function number_by_complex(x,y)
{
  make_complex(:re = x * y->re, :im = x * y->im);
};
etc
generic gmult
{
  complex, complex: 'complex_by_complex;
  { int, unint, float, char, unchar }, complex :
      'number_by_complex;
  complex, { int, unint, float, char, unchar } :
      'complex_by_number;
};
```
When the above statements are evaluated then the multiplication will work seamlessly for complex numbers. Function `gmult` is called generic function, and the functions like `number_by_complex` (which are normal functions) are called methods. When several data types specialize on the same method, like `{ int, unint.. }` above, then we can enclose them in curly brackets and the interpreter will create one entry in the look up table for each one of them.

2.2. Assignment expressions

An assignment is an expression that returns the new value of the variable, the array position or the structure field. It has the side effect that it modifies this place in memory. The assignment operators are the same as in C: `=`, `+=`, `-=`, `*=`, `/=`, `%=`. All the following are valid statements

```
a = 3;
a = b = 3+1;
mat1[3,3] = 1.1;
cm->re = 0;
a += 1;
mat1[3,3] += 2;
cm->re += 1;
```

One can assign to anything that represents a position in memory: a symbol, an array and a structure. There are a few more cases that an assignment can take place but that's beyond the scope of this manual.

3. Miscellaneous

No interpreted language is complete without a few more things. First comes on line help. There are two ways to get information about the system. The one is *apropos*. If we want to find all functions or variables that deal with images and templates we type interactively

```
??"[Ii]mage","[Tt]emplate";
```

and all the symbols that satisfy both regular expressions will show up along with one line of documentation.

The other is the *describe* function. It gives the full available documentation for this function or variable.

```
?gconvolve;
```

Both *apropos* and *describe* can be called as functions but the use of *??* and *?* save us typing. We can find more information about the regular expressions in the Unix commands `ed` and `grep`.

The comments in MediaMath are like C comments but are not allowed to span two lines. A comment must end in the line it started.

4. Library functions

There are several libraries that are accessible through MediaMath. First all the functions in the Unix math library are accessible with the same names. Then there is an extensive Lisp style list processing library, a small string library and miscellaneous utilities and the most important library, the image and matrix library.

4.1. Image and matrix library

This library introduces a set of types to operate upon, that can represent images and matrices. There 18 such types but a bit of taxonomy makes them look fewer and easy to understand.

These types are of two kinds. One dimensional and two dimensional. The 1-D ones are vectors *vec*, scan lines *scln* and 1-D templates *tmpl*. The 2-D ones are matrices *mat*, images *img* and 2-D templates *tmpl2*. So far six. Each one of them can be either float *f*, integer *i* or unsigned character `uc`. The names of all of them are composed by the initials of the underlying type and the short name of the type:

fvec	*fscln*	*ftmpl*	*fmat*	*fimg*	*ftmpl2*
ivec	*iscln*	*itmpl*	*imat*	*iimg*	*itmpl2*
ucvec	*ucscln*	*uctmpl*	*ucmat*	*ucimg*	*uctmpl2*

4.2. Creating and destroying images and matrices

For every one of these types there is a routine to create an instance. We just prefix the name of the type with `mk_` as in `mk_fvec`. All of them accept the dimension (or dimensions) of the object as argument(s) and an optional initialization.

The indices of matrix and vector objects of size *n* range from `1..n`. The range can be specified using the range operator

```
mk_fvec(4);
mk_fvec(1..4,[1,2,3.3,4]);
mk_ucmat(2,3,[[1,2],[3,4],[5,6]]);
mk_ucmat(1..2,1..3,[[1,2],[3,4],[-5,-6]]);
mk_imat(1..2,4);
```

Notice that if we choose to write `1..3` we do not give any more information. It is just different style. It is an error to write `2..4` for matrix creation.

Images and scan lines are pretty similar to matrices and vectors but start at `0`. All the following statements are valid:

```
mk_fimg(128,128);
mk_fimg(0..127,0..127);
mk_fimg(0..1,0..1,[[1,2],[1,2]]);
mk_ucscln(0..255);
```

Notice that the first two statements do exactly the same thing.

The range of templates can be anything, so they are not constrained to start from zero or one. And we can use either a range or the bounds of the range like:

```
mk_ftmpl(-2,2);
mk_ftmpl(-2..2);
mk_ftmpl2(-1..1,-2..2);
mk_ftmpl(-3,3,[1,1,1,0,2,2,2]);
```

We should not be particularly concerned with deallocation of these data structures, although they are big, because there is an efficient garbage collector to do that. The garbage collector scans the memory every now and then and finds data that are inaccessible and deallocates them. But if we feel the urge to deallocate something there is a set of routines to do that. All of them are `free_` followed by the name of the data type like `free_fimg`.

4.3. Accessing images and matrices

We can access the elements in any of the above data structures using the same syntax as with arrays.

```
a1[3];
```

```
a2[3,3];
```
where `a1` has one dimension and `a2` has two. Nothing would be different if `a1` and `a2` where vector and matrix or scan line and image or one and two dimension template. The result will be of the same type as the underlying type of the structure (float if it is a matrix of floats, integer if it is a matrix of integers etc). Also we can write
```
a1[2..4];
a2[2..5,4];
a2[2,2..4];
a2[2..4,2..4];
```
to get part of `a1` or `a2`. The type of the result of the above operations is of the same or the next smaller type that would fit it. E.g. if `a1` and `a2` where vector and matrix the first three lines above would return a vector and the forth a matrix. If they were scan line and image the first three should return scan line and the last an image. The bounds of the result might be shifted to follow the conventions. The above expressions can appear to the left of the right of an assignment statement. For more information look at `graref` and `graset`.

In many cases during a computation we might need the size of a matrix or a template. We can use the `->` operator to access them but not to modify the e.g.
```
a1->vmin;
a1->vmax;
a1->vsize;
a2->hmin;
a2->hmax;
a2->hsize;
a2->vmin;
a2->vmax;
a2->vsize;
```
The initial `v` indicates vertical dimension and the initial `h` indicates horizontal. A vector, a scan line or a template are assumed, by convention to be vertical[†].

4.4. Building images and matrices

We can concatenate two images or two matrices together to make a bigger matrix or image. If the underlying types are different the one is upgraded to the other (an unsigned character to integer and an integer to floating point). We can concatenate an image and a scan line or a scan line and a number. The same goes for matrices vectors and numbers. The concatenation operations cannot be used for templates.
```
a1 = mk_fvec(1..2,[10,20]);
a2 = mk_fmat(1..2,1..2,[[1,2],[3,4]]);
a1<->a1;         /* 2x2 matrix */
a1<|>a1;         /* 2x2 matrix */
a1<->a2;         /* 2x3 matrix */
a1<|>a2;         /* 3x2 matrix */
```

[†] Since the vertical index is written first the `(x,y)` point of an image `im` is `im[y,x]`.

```
a2<|>a2;          /* 4x2 matrix */
a2<->a2;          /* 2x4 matrix */
a1<|>3;           /* 3x1 vector */
```

We would get similar results if we had scan lines and images instead of vectors and matrices.

4.5. Convolutions

There are three convolution operators. The general convolution operator `(*)` which applies to images and templates (you cannot convolve two images though) when there is no ambiguity of the orientation of these data structures. The horizontal `(-)` that means that the template is horizontal and the vertical `(|)` that means that the template is vertical.

```
im1 = mk_fimg(4,4,[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,1]]);
t1  = mk_ftmpl2(-1,1,-1,1,[[0,1,0],[1,0,-1],[0,-1,0]]);
im1(*)t1;
t1(*)t1;
t4  = mk_ftmpl(0..2,[-1,0,1]);
t1(-)t4;
t1(|)t4;
im1(-)t4;
im1(|)t4;
t4(-)t4;
t4(|)t4;
```

There is one more convention regarding convolutions. The templates are considered zero outside their bounds (this does not affect array access, if we request the value of a template outside its bounds we get an `ArrayOutOfBounds` error signal). The images are periodic so the whole plane is tiled with the same image. So when we convolve an image (or a scan line) with a template the size of the resulting image is the same as the original. When we convolve two templates the resulting template is bigger that the originals.

4.6. Arithmetic operations

Most of the operations that apply to numbers apply to images, matrices and templates as well. The above convention (that templates are considered zero outside their bounds and images are periodic) applies here also. These operations on matrices and vectors are the linear algebra operations. Dot product is done using transposition.

```
v1 ^T * v2;
```

These operations work between images, matrices etc and numbers the way one would expect. All operations are done in floating point (it is faster on most modern machines).

There is an important difference between arithmetic with images and numbers. The operations

```
a = b;
a += 2;
```

have different effects when `b` is a number or an image. If it is a number the operation `a += 2;` creates a new object to store the number. If it is an image then it modifies the object that contains the image by adding 2 to every element. This is a mechanism to avoid allocating new images or matrices under the programmers control. It can be a way to optimize the code for speed or a way to introduce bugs.

5. Math library

The whole math library can be accessed through MediaMath using the same names. There is some minor loss of accuracy because MediaMath uses single floating point arithmetic. All of them have a short line of documentation. For more information look at the Unix man pages. The functions are:

Trigonometric

acos	*asin*	*atan*
atan2	*cos*	*hypot*
sin	*tan*	

Exponential

acosh	*asinh*	*atanh*
cosh	*exp*	*exp10*
exp2	*expm1*	*log*
log10	*log1p*	*log2*
sinh	*tanh*	

Bessel

j0	*j1*	*jn*
y0	*y1*	*yn*

Miscellaneous

cbrt	*erf*	*erfc*
fabs	*remainder*	*sqrt*

Limits

finite	*infinity*	*isinf*
isnan	*isnormal*	*issubnormal*
iszero	*max_normal*	*max_subnormal*
min_normal	*min_subnormal*	*signbit*

6. List processing library

The internal representation of MediaMath is influenced by Lisp. As a result it comes with an extensive list processing library. This library is usefull for implementing serious extensions to the system, but the typical user does not need to be aware of its existence, save a couple of functions to handle *&rest* arguments.

append	*assoc*	*assq*
atom	*boundp*	*car*
cdr	*compose_symbol*	*compose_symbol_soft*
cons	*consp*	*copy*
copy_alist	*delq*	*eq*

fboundp	*fmakunbound*	*fset*
get	*getenv*	*getenvlist*
intern	*intern_soft*	*length*
list	*listp*	*make_array*
make_list	*make_string*	*make_symbol*
makunbound	*memq*	*nconc*
nlistp	*not*	*nreverse*
nth	*nthcdr*	*null*
put	*range_car*	*range_cdr*
range_cons	*rassoc*	*rassq*
reverse	*rplaca*	*rplacd*
set	*setplist*	*setq*
symbol_function	*symbol_name*	*symbol_plist*
symbol_value	*symbolp*	*time*

For each one of them there is on line documentation, examples etc. It is a fairly complete list processing system, that could support symbolic computation etc.

MediaMath

An Interactive System for Image and Audio Analysis

Run time linking of C modules.

Part 2.

Run time linking of C modules

1. Overview of the C modules

The MediaMath system provides a simple mechanism to extend its functionality by adding functions written in C. The mechanism is fundamental to the system and not an extra feature. It is used by the developers to write everything except the central part of the interpreter. As such it is extensively debugged and tested during the development. It does not incur any speed penalty and it is intended to be simple and flexible, and easy to master for somebody familiar with the UNIX™ system.

Modules written in C can be linked and unlinked to the executable at run time by a simple command, e.g. if the module is named "hough.so" you can load it with the command[†]

```
dlopen("hough.so");
```

If for some reason you want to unlink "hough.so" then

```
dlclose("hough.so");
```

is enough. You rarely need this however. If while using "hough.so" you discover a bug you can go back to the C code of "hough.so" correct the bug and link it again in the same manner. The old version is automaticly unlinked and you will notice no side effect. This comes in handy when you discover a bug and you already have a couple of hours useful work in the memory.

The C routines have to be written in a certain style in order to be linked. The interpreter has to be informed of the name of the function the number of arguments, the type of function (for most of the things you would like to do the type "function" is enough; the others are macro and special form, people familiar with Lisp should recognize them). Then the documentation has to be provided. All this seems quite an overhead for the programmer, but two mechanisms are provided to make this task trivial. First there is an emacs function to insert a header template for all this information. The user has just to fill it up.[*] The rest are done automaticly by the second mechanism that uses the awk language: The C source file is scanned by an awk program that extracts this information from the headers and writes the appropriate code to inform the interpreter. Again, this mechanism is very well tested because this is how all the functions of the system are written. As a result a substantial part of the code and the declarations are produced by the awk program.

All the arguments to your C functions that are going to be called by the interpreter are of type `L_Ptr` which stands for "lisp pointer". This is a pointer to a union that can accommodate

[†] Users familiar with SunOs will realize that the name is borrowed from there. The actual function `dlopen_lisp` that is called in response does little more than set up the arguments for calling SunOs `dlopen` when working on a Sun

[*] A programmer that always puts headers at the beginning of his functions should not have any problem with that (assuming such a programmer exists)

all the types used by the system including the ones you are going to define in your modules. It is the responsibility of the programmer to check if the types wrapped in the union are acceptable. A wide variety of cpp macros are provided to make this easy. If the types of the arguments are deemed acceptable then you can use them directly if they are lisp types (like cons cells, lisp arrays, nil etc). Otherwise they are C types and you have to extract them. Another set of cpp macros is provided for this purpose.

The last issue is the garbage collector. Luckily for the most part you should not worry about it. The garbage collector is called only from within the `eval` function and in general only advanced system utilities normally call `eval`. But in case you don't have a choice you can use one of two mechanisms: Either suspend garbage collection for the duration of this call to `eval`, a simple but wasteful technique, or protect the arguments and variables of your function. The reason you have to do one of these is that the garbage collector will mess up anything that he thinks is inaccessible and thus useless. So your function has to notify the garbage collector that its local variables are accessible.

When you start a new module you have to first copy into a fresh directory the contents of the ModulePrototype that contains a simple example. You can add and remove files with C code or declarations (*.c and *.h) to write your own module. You have to edit the makefile to let the make utility know your SRC files and your local headers. Then you type

```
make setup
```

and you are ready to start. Put your C declarations in a .h file and your C code in a .c file. The first thing you write is routines to generate and destroy instances of the data types you declare. Before every function that can be called by the interpreter include a header. Every function first checks the type of its arguments and extracts the corresponding C values from them. Then most of them should call a function to do the actual work and use the result to construct a Lisp data type to be returned. After you have written enough routines that can be tested together type

```
make -k
```

to compile and link with the dynamic loader. You have a module ready to be linked to an already running MediaMath. Type

```
dlopen("hough.so");
```

at the MediaMath prompt and you are ready to call your routines in exactly the same fashion as any routine in the system.

2. Writing a simple function

Let's write a very simple function that accepts one argument, a Float and returns the sine of it using the standard math library `sin`. This does not need any new C types other than Float so we don't need a .h file. Notice that the type Float is capitalized, which means that is not the same as `float`. It is a compile time option to specify Float as either float of double.

The first thing to do is to create a header. If you are using emacs and have loaded the appropriate emacs lisp files just type `meta-h`. A ten line header will be inserted and you have to fill it out.

```
/*awkstart****************************************************
Name: mysin
MinArgs: 1
```

```
MaxArgs: 1
Type: function
Synopsis: mysin(<float>)
Doc: This function computes the sine of <float>.
Doc: It is just an example of how to write simple functions.
Example: mysin(3.14/2);
****************************************************awkstop*/

L_Ptr mysin(Lnum)
                    L_Ptr Lnum;
{
  Float f,res;
  if (!FLOATP(Lnum))
    error_signal(Smysin,SWrongTypeArg,Sfloatp,Lnum);
  f = XFLOAT(Lnum);
  res = sin(f);
  return C_2_L_float(res);
}
```

This is how your function will look like. If you use one of the files provided you can just copy your function in and compile by typing `meta-x compile`. You can load your function in a running MediaMath by typing `dlopen("example.so")`. Try it.

Lets look at these fields one by one. First the name field provides no surprise (so far). The minargs and maxargs fields are what one would expect too. The type of the function is “function” which means that the arguments to your function are evaluated. In other words if you have the sequence

```
a = 1.0;
b = mysin(a);
```

then your function will see the `Float` *1.0* whereas if it was `specialfrm` (special form) then your function would see the symbol *a* which is not convenient here.

The next field is synopsis. It just gives a synopsis of the syntax of the function. It is up to the programmer to choose if he wants to write `<float>` or `<number>`. The next field is doc. This can be several lines long but each line has to start with a “`Doc: `”. The first line of these is kept as a string in the main memory so that the apropos command can read it to try to match strings. So better have a good summary of the description as a first line. All the lines go to a file “mysin.doc” to be read by the help command. Finally the example field records a few simple examples to be used by the help command.

The header seems pretty straightforward so far. It does not get that much more complicated. The body of the function is also simple.

Both the arguments and the return value are pointers to this catchall union. The first thing to do is to check if the argument is indeed a `Float`. This is done with the cpp macro FLOATP(). If you are using emacs you can type `meta-x c-macro-expand` to see what it is doing. Just checks a number stored in the first word of the data structure. If the test is negative then the `error_signal` function is called to signal the error, print a message and return to the read eval loop. The message contains what is in the arguments of the `error_signal` call. Lets have a look into each one of them. The first argument is `Smysin`. This is the symbol whose value is the actual function. You don't have to declare and initialize it. This is done by the awk program[†]. Every function that can be invoked by the interpreter, has a symbol which has the same name (there are two exceptions to this) and there is always a variable that points to it. The name of the variable is the same as the name of the function with an `S` attached before it.

The next field is `SWrongTypeArg`. This is the error message. Notice the funny capitalization. Words that don't have to be typed in often use it. Error messages are among them. Again we provide the symbol `SWrongTypeArg`. The user can retrieve the documentation of this symbol to get more information on this type of error.

The next field is the comment field. We gave `Sfloatp` to give a hint that floatp would fail for this type of argument. The programmer should give whatever might help the user find what kind of error happened. The last argument is the value that did the damage.

If the test is positive then the execution goes on. At this pont we know that the argument is a `Float`. We can extract it using the `XFLOAT` macro. This returns the `Float` stored in the third word of the data structure and you can see it by expanding the macro with emacs.

We have the actual value of the `Float` now, so we call `sin` which returns another `Float` (well actually double) which we pass as argument to the function `C_2_L_float` (which stands for C to Lisp `Float`). This function creates a data structure to hold the `Float` that has all the correct tags etc. This is what `mysin` returns.

2.1. Improving our first module

If you were to add this module to the system, you would surprise the user. First the function `sin` is called `mysin`. Second even in a staticly typed language like ANSI C, one can give an integer as an argument to a function that needs a `Float`. Lets see how we can fix it.

We can try to change the name of the function to `sin` from `mysin`. That would be an invitation for trouble because `sin` is the name of the math library function. We already saw how this problem was solved. If you notice the example about dynamic linking above, the name of the function *`dlopen`* is known inside the C code by the name `dlopen_lisp`. Our function is `mysin` both inside the interpreter and inside the C program.

The whole job is done by the awk programs. When a function is declared by a name that ends with `_lisp` like `dlopen_lisp` the awk programs keep two versions of the name: one with and one without `_lisp` in the end. The one without is what appears in the run time hash table of the system and the one with `_lisp` is what the C compiler and the linker see. So they are not confused by the names.

The other problem can also be easily solved. One can replace the check

```
  if (!FLOATP(Lnum))
```

[†] a symbol is a collection of three things: the name which is a string, the value which can be anything or undefined and the property list where any combination of key value pairs can be stored. The property list is used mainly by the system to store important information and if you don't know what your are doing leave it alone. A symbol can be interned in a hash table, in which case it can be retrieved by its name.

```
    error_signal(Smysin,SWrongTypeArg,Sfloatp,Lnum);
```
with
```
  Lnum = to_float(Lnum);
```
Function `to_float` returns a `Float` if `Lnum` is convertible to `Float`, or signals an error otherwise. After the call to `to_float` we know that `Lnum` is a `Float`. The rest of the program proceeds in the same way. We solved a problem but we created another one. If the user provides `sin` with an invalid type, the error will be issued by `to_float` and there is no simple way to trace the origin of the error. If this is a concern then the best thing to do is an error checking inside `sin`. You can use the `numberp` function for that. `numberp` returns symbol `t` if the argument is a number and `nil` otherwise. The program then would look like
```
/*awkstart**************************************************
Name: sin_lisp
Minargs: 1
Maxargs: 1
Type: function
Synopsis: sin(<number>)
Doc: This function computes the sine of <number>.
Doc: It is just an example of how to write simple functions.
Example: sin(3);
****************************************************awkstop*/

L_Ptr sin_lisp(Lnum)
                   L_Ptr Lnum;
{
  Float f,res;
  if (NULLP(numberp(Lnum)))
    error_signal(Ssin_lisp,SWrongTypeArg,Snumberp,Lnum);
  Lnum = to_float(Lnum);
  f = XFLOAT(Lnum);
  res = sin(f);
  return C_2_L_float(res);
}
```
Notice the use of the macro `NULLP` that checks if the function returns `NIL`. You could write
```
NIL == numberp(Lnum)
```
or
```
St != numberP(Lnum);
```
where `St` and `NIL` are the *`t`* and *`nil`* in the C environment.

At this point we know enough to write any simple function. With the help of the Reference Manual we can write any function that does not need new types, variable arguments, garbage collector protection or list processing.

3. Variable Arguments

First, we have to inform the system that we use variable arguments by declaring different minimum and maximum number of them. If the maximum number of arguments is infinite use

`-1`.

The awk program is going to do all the work to notify the interpreter about the arguments of this function. The interpreter will match one by one the minimum number of arguments and will put the rest in a list and give it as the last argument. So your program has to accept the minimum plus one number of arguments.

Let's write a function that computes the sine of a number, that accepts an optional argument, that when non `nil` instead of the sine computes the hyperbolic sine. The code should look like this

```
/*awkstart****************************************************
Name: sin_lisp
Minargs: 1
Maxargs: 2
Type: function
Synopsis: sin(<number>[,<hyper>])
Doc: Computes the sine of <number>. With optional non nil
Doc: argument computes the hyperbolic sine. It is just
Doc: an example of how to write simple functions with
Doc: variable args
Example: sin(3,t);
****************************************************awkstop*/

L_Ptr sin_lisp(Lnum,rest)
                   L_Ptr Lnum, rest;
{
  Float f,res;
  if (NULLP(numberp(Lnum)))
    error_signal(Ssin_lisp,SWrongTypeArg,Snumberp,Lnum);
  Lnum = to_float(Lnum);
  f = XFLOAT(Lnum);
  if (!NULLP(rest))
    {
      if (!NULLP(cdr(rest)))
        error_signal(Ssin_lisp,STooManyArgs,NIL,rest);
      if (!NULLP(car(rest))) return C_2_L_float(sinh(f));
    }
  return C_2_L_float(sin(f));
}
```

The code is easy to follow. If no second argument is supplied then rest is `nil`. If a second argument is supplied it is a list that can be accessed with `car` and `cdr`. For those not familiar with Lisp, `car` returns the first element of a list and `cdr` the rest of the list. The statement `if (!NULLP(cdr(rest)))` checks if more than 2 arguments are supplied, in which case an error is signaled. The interpreter does not check for that.[϶]

3.1. Adding a few more features

This function does a pretty good job, but let's make it a bit fancier. If a second argument is supplied it should be either `hyperbolic` or `periodic`. This way the user is forced to include these words in his function call making the code more readable; you don't need to look up the manual to find out what's going on, it is there.

First of all we have to let the interpreter know about the symbols `hyperbolic` and `periodic`. This would be automatic inside a running MediaMath system, because the symbols are interned immediately as they are encountered. But inside the C code you have to intern it yourself and keep a global variable around to access it. It would be a lot of trouble to do it manually but the awk program does it for you. The mechanism is very similar to the headers for functions. It would look like this

```
/*awkvarstart*************************************************
Name: hyperbolic
Doc: Flag symbol. If present as the second arg of a trig
Doc: function, it returns the hyperbolic counterpart
Value:
****************************************************awkvarstop*/

/*awkvarstart*************************************************
Name: periodic
Doc: Flag symbol. The opposite of the hyperbolic flag.
Value:
****************************************************awkvarstop*/
```

If you are using emacs you type `meta-p` and an empty will appear. A global variable is declared for each header: `Shyperbolic` and `Speriodic`. You may notice the same convention. The leading `S` means symbol. If the value field is present, a global variable with the same value is declared that points to this value. We'll see its most common use when we talk about types. Here we do not use the value field.

The `doc` entry does exactly the same job as in function headers. If we use the same symbols anywhere in the module then we don't redefine them. If some other module has declared them too, then the last doc string and the last value are kept and nothing else changes. If you want to avoid that then you have to use the import mechanism.

Now that we have seen how to introduce new symbols we are ready to proceed with the code.

```
/*awkstart***************************************************
Name: sin_lisp
Minargs: 1
Maxargs: 2
Type: function
```

[϶] You might notice that the actual value of `MaxArgs`, is irrelevant. It only matters if it is equal or not to `MinArgs`.

```
Synopsis: sin(<number>[,{hyperbolic, periodic}])
Doc: Computes the sine of <number>. If the optional argument
Doc: is hyperbolic then the hyperbolic sine is returned. If
Doc: it is missing or it is periodic the periodic sine is
Doc: returned. It is just an example of how to write simple
Doc: functions with variable args that use new symbols.
Example: sin(3,'hyperbolic);
****************************************************awkstop*/

L_Ptr sin_lisp(Lnum,rest)
                   L_Ptr Lnum, rest;
{
  Float f,res;
  if (NULLP(numberp(Lnum)))
    error_signal(Ssin_lisp,SWrongTypeArg,Snumberp,Lnum);
  Lnum = to_float(Lnum);
  f = XFLOAT(Lnum);
  if (!NULLP(rest))
    {
      L_Ptr temp;

      if (!NULLP(cdr(rest)))
        error_signal(Ssin_lisp,STooManyArgs,NIL,rest);
      temp = car(rest);
      if (temp == Shyperbolic) return C_2_L_float(sinh(f));
      if (temp != Speriodic)
        error_signal(Ssin_lisp,SUnexpectedArg,NIL,rest);
    }
  return C_2_L_float(sin(f));
}
```
Again `Shyperbolic` and `Speriodic` are the symbols that inside a running MediaMath are *`hyperbolic`* and *`periodic`*. Also notice in the example above that we have to quote *`hyperbolic`* to avoid evaluation. If we leave it unquoted then we will trigger an error.

4. Types

Before we explain how to introduce new types we have to describe the type system. In this section we talk about how to introduce types and operations that handle conventional C structures, how to make the garbage collector dispose them and the printing routines display them.

4.1. Classification

All the types used in the MediaMath belong in one of two categories: Lisp types and C types. The Lisp types are further classified as evaled and unevaled, e.g. ones that the *`eval`* function of the interpreter evaluates and returns the result or does not evaluate and returns them as they are.

The lisp types that can be evaluated are among others the symbol (the value field of the symbol is returned) and the function call which is a list of a function and its arguments (the result of the function call is returned.). The evaluation procedure is complicated and it is described elsewhere.

The types that are not evaluated are almost all the rest: an array, a structure, nil etc. In other words these types represent themselves. All the C types are also unevaled.

C types are also classified in two categories. “Small” and “large”. “Small” are all the predefined ones and “large” are all the ones defined by the user. The name comes from the fact that most of the predefined ones are “small” (chars, integers, `Floats`, etc) whereas most of the user defined ones are “large” (images, matrices, etc). (See fig. 1)

4.2. Run time representation

Every lisp object is an array of words. Every word is the same size as the pointer on the particular machine. The first word of the array is the header which contains the markbit (for the mark and sweep garbage collector), the type tag which is a number that represents the type and one more integers for the extension. The extension represents different kinds of information depending on the type. For all lisp types contains the size of the array even if the size is implicit in the type (like cons cell which has size 2). For C types the use of this integer varies. Some of the predefined C types do not use it. Others like the ones that represent C functions use it to store the minimum number of arguments. The user defined types use the extension to store the actual number that represents the type (the type tag contains only the number that corresponds to

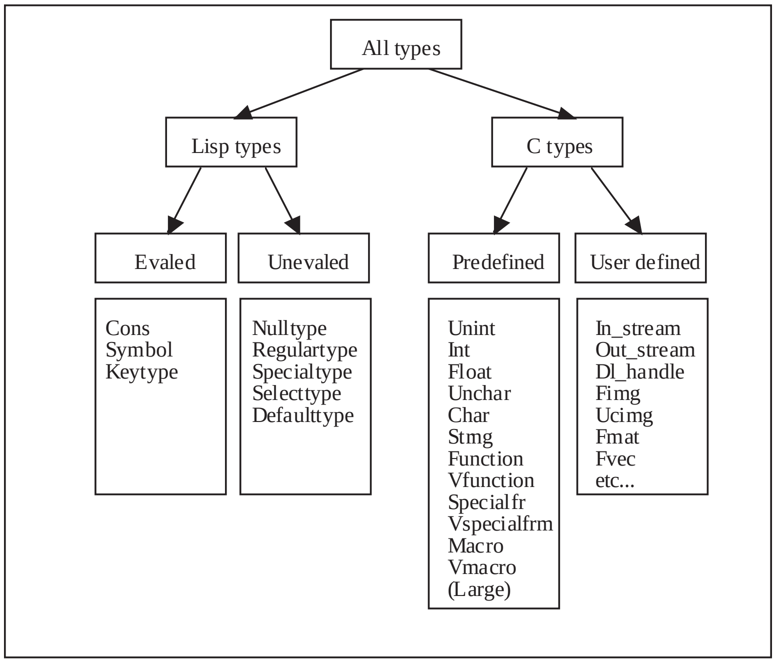

Fig. 1. All the types in the MediaMath system.

"large", a kind of escape tag).

The "large" type tag does not correspond to any real type. It is just an extension tag to take care of the user defined types. In other words the function `type_of` will never return `large`[•] All the user defined types are allocated numbers that are then stored in the extension in the header. The allocation is done by the system (function `create_C_type` takes care of this) and may vary from execution to execution. (See fig. 2)

4.3. Keeping track of the types

The function `type_of` will return a symbol which is the name of the type. It is the symbol `int` for integer, the symbol `float` for a floating number, the symbol `fimg` for a floating point image etc. This we call type name. The value of the type name is initially undefined (and the user can define it without any interference with the type system).

The next symbol involved in the types is the `Typenum` symbol. This is symbol `intTypenum` for integer, symbol `floatTypenum` for a floating number, symbol `fimgTypenum` for a floating point image etc. The rule to generate the `Typenum` symbol from the name is obvious. The value of the `Typenum` symbol is the `Typenum`, an integer unique to this type, at least during a session. The `Typenum` of a predefined type is stored in the type field of every instance of the type. The `Typenum` of the user defined C types is stored in the extension of the header. This symbol contains most of the useful information needed for typechecking etc and its name makes it hard for the user to use it accidentally as a variable.

The property lists of these symbols contain most of the useful information. The property list of the name symbol has the property `Typenum` which is the corresponding `Typenum` symbol.

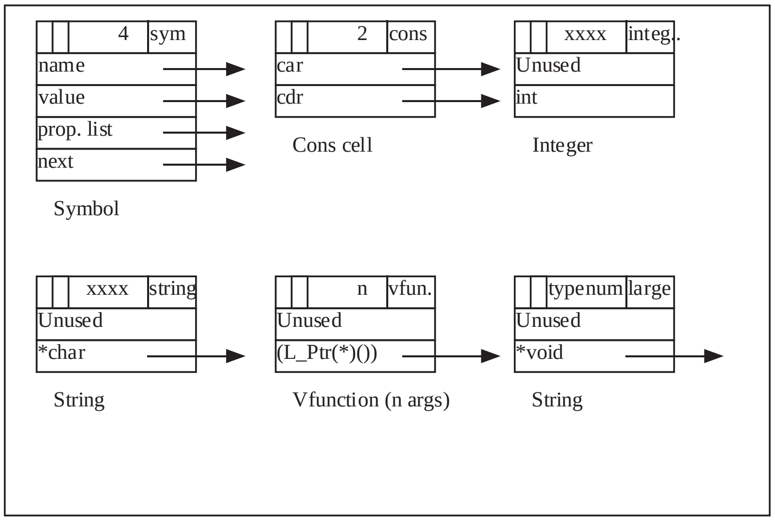

Fig. 2. Internal representation of various types.

[•] The same is true for the `special` type. It is an escape for all the user defined lisp structures. Again the function `type_of` will never return `special`.

This gives a faster way to get the *Typenum* symbol than constructing the symbol from its string and *intern*ing it. It also holds the first line of the documentation in the *Doc* property. The *Typenum* symbol contains a bit more information. First the *Doc* property contains the same documentation line. Then the *Name* property contains the name symbol. The *Destroy* property, present in user defined C types, contains the function that can be called by the garbage collector to deallocate the type (free the space for an image, close a file etc). It can also contain the *Printer* property to print an instance of the type (present in some C types).

Given the name or the *Typenum* symbol of a type you can find all the information available to the system. But given the *Typenum* integer, you cannot do it easily. For this purpose there is the `type_array` which maps these integers to the corresponding symbols.

4.4. Defining a new C type

After this brief exposition of the type system it is clear that it is not easy to deal all the details. For this reason there are tools to isolate the user from it. Defining a new type is as easy as defining the *Typenum* and at least one function to deallocate it (if deallocation does not make sense, still this function has to be defined to do nothing and just return `NIL`). Again the awk program does most of the work with the help of a couple of functions, macros and simple conventions.

4.4.1. Example: define floating image types

We can have a look into a real type now. It is not worth to find a toy example to do the introduction because it is really simple by itself. The code comes from the `ImageBasics` module and not the core interpreter, but it would be exactly the same if it was from the interpreter.

As most modules `ImageBasics` defines two levels of functions, one and two. The first level functions do the actual work and do not deal with headers, data types, error checking etc. They accept arguments that had their headers removed (e.g. a `Float` is a `Float` and not an `L_Ptr` whose tag is `FLOAT`). They are not called directly from the interpreter and they do not call any high level function including error handler. In general every effort has to be taken so that these functions can be used with as little modification as possible in other packages as library functions. And of course the opposite: any public domain function should easily fit among the level one functions. The level two functions do all the error checking, type casting etc and then call one or more level one functions. It is then obvious that level two functions require some familiarity with MediaMath whereas level one are conventional functions. So we present only level two code.

All the level two code we need in order to define a floating point image type is listed below. We might need to do a bit more than that, to have a complete abstract data type, like write functions to access or modify a pixel, add two images, display an image etc.

```
/*awkstart*************************************************
Name: mksimple_fimg
MinArgs: 2
MaxArgs: 2
Type: function
```

```
Synopsis: mksimple_fimg(<vdim>,<hdim>)
Doc: Allocates an image with <vdim> rows and <hdim> columns.
SEE: free_fimg, mksimple_fimg
Example:
****************************************************awkstop*/

L_Ptr mksimple_fimg(vdim,hdim)
     L_Ptr vdim, hdim;
{
  int Cvdim, Chdim;
  ftwo_Dptr fimg;
  L_Ptr res;

  TO_INT(vdim,Cvdim,Smksimple_fimg);
  TO_INT(hdim,Chdim,Smksimple_fimg);

  if ( (Cvdim <= 0)||(Chdim <= 0) )
    error_signal(Smksimple_fimg,SNonPosSize,NIL,LIST2(vdim,hdim));

  fimg = mk_ftwo_D(0,Cvdim-1,0,Chdim-1);

  res =  C_2_L_large((void*)fimg,XtypenumDefinition(SfimgTypenum));
  return res;
}
```

The function above allocates an instance of a floating pont image *`fimg`*. It accepts as arguments the dimensions of the image. It first extracts the integer values of the dimensions (cpp macro `TO_INT` does the extraction and typechecking in a smart way). Some additional error checking follows. Then space for the image is allocated by function `mk_ftwo_D`, which allocates space for two dimensional objects like images and matrices. This function returns a pointer to a newly allocated C structure. Then function `mksimple_fimg` returns this pointer encased in a structure that has a header. The encasing is done by `C_2_L_large` that accepts two arguments: the pointer to the image structure and the *`Typenum`* as a plain integer. XtypenumDefinition(Sfimg-Typenum) can help locate it.

```
/*awkstart****************************************************
Name: free_fimg
MinArgs: 1
MaxArgs: 1
Type: function
Synopsis: (free_fimg <fimg>)
Doc: Frees the space allocated to <fimg>. It is called
Doc: automatically by GC when <fimg> is no longer accessible.
SEE: mk_fimg
```

```
Example: free_fimg(mk_fimg(100,100));
****************************************************awkstop*/

L_Ptr free_fimg(fimg)
     L_Ptr fimg;
{
  if (!CHK_TYPE(Sfimg,fimg))
    error_signal(Sfree_fimg,SWrongTypeArg,NIL,fimg);

  if (NULLP((L_Ptr)XLARGE(fimg))) return NIL;
  free_ftwo_D(XLARGE(fimg));

  XLARGE(fimg) = (void*)NIL;
  return NIL;
}
```

The only function we need to define to have a descent data type is `free_fimg`. This function has to be defined, otherwise the Garbage Collector will not know what to do with an *`fimg`* if it sees one that is inaccessible. That's a simple routine that first checks the type of the argument, then makes sure that the data structure is not freed already (the convention is that the pointer to the data structure inside the header is replaced with a `NIL` if the image is already freed). Then frees the data structure and sets the no longer valid pointer in the header to `NIL` to obey the convention.

```
/*awkstart****************************************************
Name: print_fimg
MinArgs: 1
MaxArgs: 1
Type: function
Synopsis: print_fimg(<fimg>)
Doc: Prints <fimg>. Normaly it is invoked by the
Doc: read-eval-print loop.
SEE: prin
Example: print_fimg(a[1..10,5..10]);
****************************************************awkstop*/

L_Ptr print_fimg(fimg)
     L_Ptr fimg;
{
  if (!CHK_LIVETYPE(Sfimg,fimg))
    error_signal(Sprint_fimg,SWrongTypeArg,NIL,NIL);

  print_ftwo_D((ftwo_Dptr)XLARGE(fimg));
  return fimg;
}
```

The function `print_fimg` is optional. If we do not define it the function *`prin`* that is called to print the result of an operation is going to print something by default which might not always be useful. If we define it then function *`prin`* is going to look for it and use it.

The function by itself is very simple: Macro `CHK_LIVETYPE` tells if the type is correct and not already freed (the user can free an instance of a type since the *`free_fimg`* is available to him) and then calls a function that does the printing. As all functions related to *`prin`*, it has to return its argument.

```
/*awkvarstart****************************************************
Name: fimgTypenum
Doc: Type symbol. The type of an image of floats.
Value: create_C_type(SfimgTypenum,"print_fimg","free_fimg");
****************************************************awkvarstop*/
```

And in order to inform the world that a new data structure was born we use the good old awk headers. The conventions are simple. The name has the suffix `Typenum`. The Doc line contains the string `Type symbol` to make it easy to search with apropos. The Value part is just a call to `create_C_type`, with the symbol of the type a first argument, an optional name (C string) of the function that can print the data type as a second argument and the name of the function that frees it as the third. The routine `create_C_type()` does all the work (which would be pretty complicated otherwise) like putting the appropriate things in the hash table, initializing the property lists of the appropriate symbols and allocating new integer to represent the new *`Typenum`*.

5. Linking

Most of the details of linking are taken care of by the *`dlopen`*. There is one more though that *`dlopen`* cannot do. When we call *`dlopen`* the following things happen. The dynamic linker is called and all the variables in the module that are undefined are linked to the core interpreter. Then the function `init` that should be present in every module is called and inserts in the hash table of the interpreter the names of the functions that are available in the module. What is still needed is to link the function symbols in the new module to functions in other modules which should be already loaded. In other words take care of the interdependence of the modules. This is a difficult job to do for every module so there is a simple tool for that: yet another header! If you use emacs then the sequence `meta-m` will create an empty header for you to fill. As an example let's see how other modules would use the module `ImageBasics`.

```
/*awkimportstart*************************************************
Module: image.so
Symbol: CannotReadImg
Symbol: CannotWriteImg
Symbol: IncopatibleSizes
Symbol: NonConfRange
Symbol: NonPosSize
```

```
Symbol: fimgTypenum
Symbol: fmatTypenum
Symbol: fsclnTypenum
etc...
Function: free_fimg
Function: free_fmat
Function: free_fscln
Function: free_ftmpl
Function: free_ftmpl2
Function: free_fvec
etc..
************************************************awkimportstop*/
```

The first line is the module that these functions or symbols come from. The name is the name of the shared object that contains the module. If it is not already there it is linked. The rest are the names of the symbols and functions that the new module needs to get from the ImageBasics. These functions can be used then anywhere in the new module and have to be defined only once. All the details like C declarations etc are taken care of.

MediaMath

An Interactive System for Image and Audio Analysis

C modules: Reference manual

Part 3.

C modules: Reference manual

1. General

The MediaMath interpreter is structured like a Lisp interpreter. Every data structure has a header of type `union L_Header`. Any argument to any function and any value any function returns is a pointer to `L_Header` which is `typedef`ed to `L_Ptr`. Any function is thus defined to return `L_Ptr` and all the arguments to functions are of type `L_Ptr`. Any object that is defined as `L_Ptr` we call tagged object and it is actually just a pointer.

All the typechecking is done as a result in run time. Facilities are provided to check the type of an object, extract the actual value of an object and to create an object with a header out of a conventional C type.

2. Lisp types

There are two major kinds of types. The lisp types and the C types. The lisp types contain only other tagged objects. The `CONS` type for instance contains a pointer to the first element of a list and the rest of the list. For all these types we provide macros and functions for typechecking, accessing (and modifying) and creation.

A lisp type can be thought of as an array of tagged objects. The length of this array can be extracted with the macro `XREGLEN(t_obj)` where `t_obj` is any lisp object.

```
int XREGLEN(t_obj)
     L_Ptr t_obj;
```

The Lisp types are:

```
SPECIALTYPE          REGULARTYPE          CONS
SYMBOL               NULLTYPE             SELECTTYPE
DEFAULTTYPE          KEYTYPE              RANGETYPE
```

All the above types are intended for advanced MediaMath programming and can be ignored by the casual module writter. The recommended method of manipulating these structures is with the high level routines that are provided, which are also documented and accessible within the MediaMath interpreter.

SPECIALTYPE

Description:

The type of *`struct`*. Should not be normally used by C functions.

Typechecking:

`SPECIALP(t_obj,sz)`: cpp macro that returns 1 if `t_`obj is `SPECIALTYPE` and has length `sz`, 0 otherwise. If `sz` is zero does not check for size.

```
int SPECIALP(t_obj,sz)
    L_Ptr t_obj;
    int sz;
```

Access:

`XPTR(t_obj+n)`: cpp macro that returns the `n`th field of `t_obj`. The fields are numbered starting from 2. It is better to use `sref` to do the same thing. The typenum of the `t_obj` is `XPTR(t_obj+1)`. When the macro appears on the left side of an assignment statement it will modify the contents of `t_obj`. Again it is better to use `sset`.

```
L_Ptr XPTR(t_obj+n)
    L_Ptr t_obj;
    int n;
L_Ptr sref(curr_tnum,strct,indx)
    L_Ptr curr_tnum, strct, indx;
L_Ptr sset(curr_tnum,strct,indx,obj)
    L_Ptr curr_tnum, strct, indx, obj
```

Creation:

`mk_typenum_struct(num,sz)`: It creates a structure with `typenum num` and size `sz`.

```
L_Ptr mk_typenum_struct(num,sz)
    L_Ptr num, sz;
```

REGULARTYPE

Description:

The type of an array of lisp pointers. Every pointer can point to a different kind of element. It can be used to hold a sequence of images or a set of templates, etc.

Typechecking:

`REGULARP(t_obj,sz)`: cpp macro that returns 1 if t_obj is `REGULARTYPE` and has length `sz`, 0 otherwise. If `sz` is zero does not check for size.

```
int REGULARP(t_obj,sz)
    L_Ptr t_obj;
    int sz;
```

Access:

`XPTR(t_obj+n)`: cpp macro that returns the `n`th element of `t_obj`. The elements are numbered starting from 1. It is better to use `aref` to do the same thing. When the macro appears on the left side of an assignment statement it will modify the contents of `t_obj`. It is better to use `aset`.

```
L_Ptr XPTR(t_obj+n)
     L_Ptr t_obj;
     int n;
```

`aset(array, intgr, elm)`: Set the `intgr`th element of lisp `array` to `elm` and return `elm`.

```
L_Ptr aset(arr,indx,elm)
     L_Ptr arr, indx, elm;
```

`aref(arr,indx)`: Access the `indx` element of `arr`

```
L_Ptr aref(arr,indx)
     L_Ptr arr, indx;
```

Creation:

make_array(len,elm): Returns a new array 1..len every element of which contains elm.

```
L_Ptr make_array(len,elm)
     L_Ptr len, elm;
```

CONS

Description:

The type of a list. The name comes from lisp. Lists are overloaded with uses. The carry the multiple arguments for arguments declared *&rest*, they are the internal representation for MediaMath programs (in a Lisp style), as well as ordinary lists for general programming. There is an extensive list processing library to manipulate them.

Typechecking:

CONSP(t_obj) returns non-zero if t_obj is a cons cell zero otherwise. While NIL is the empty list CONSP(NIL) will return zero for that.

```
int CONSP(t_obj)
     L_Ptr t_obj;
```

Access:

XPTR(t_obj+1) returns the car of the list (eg. the first element)

XPTR(t_obj+2) returns the cdr of the list (eg. the rest of the list). Both of the can appear in the left side of an assignment statement to set the car and the cdr of a cons cell.

The recommended way to access them is using car(l) and cdr(l) and to modify them rplaca(l,elm) and rplacd(l,tl).

```
L_Ptr XPTR(t_obj+n)
     L_Ptr t_obj;
     int n;
```

car(cell) and cdr(cell): Return the first and second element of cons cell. If cell is viewed as a list the meaning is that they return the first element and the rest of the list.

```
L_Ptr cdr(cell)
     L_Ptr cell;
L_Ptr car(cell)
     L_Ptr cell;
```

rplaca(l,elm) and rplacd(l,tl): rplaca(l,elm) replaces the car of list with elm. rplacd(l,tl) replaces the cdr of list with tl.

```
L_Ptr rplaca(lst,newel)
     L_Ptr lst,newel;
L_Ptr rplacd(lst,newel)
     L_Ptr lst,newel;
```

Creation:

cons(a,b) creates a cons cell with a and b as elements.

```
L_Ptr cons(a,b)
     L_Ptr a, b;
```

`LIST2(el1,el2)..LIST6(el1..el6)`: A simple and convenient way to create lists up to length 6.

```
L_Ptr LIST2(el1,el2)
     L_Ptr el1, el2;
```

SYMBOL

Description:

The type of a symbol. Symbols serve as variables, place holders, function names etc. They are structures that contain four fields: name, value, property list and next. They can be in a hash table in which case they are called (`intern`ed and outside a hash table in which case they are called un`intern`ed. The next field serves only for the external chaining of the hash table and works better if left alone.

Typechecking:

`SYMBOLP(t_obj)`: returns non-zero if `t_obj` is a symbol, zero otherwise. While `NIL` is considered a symbol `SYMBOLP(NIL)` returns zero.

```
L_Ptr SYMBOLP(t_obj)
     L_Ptr t_obj;
```

Access:

`XPTR(t_obj+pos)`: returns the name, value, property list, or next when `pos` is `NAME_POSTN`, `VALU_POSTN`, `PLIST_POSTN`, `NEXT_POSTN`. `XPTR(t_obj+pos)` can appear on the left side of an assignment statement.

```
L_Ptr XPTR(t_obj+pos)
     L_Ptr t_obj;
     int pos;
```

Creation:

`intern(str,NIL)`: Returns an interned symbol with name `str`. The second argument has to be `NIL` for future versions that will support multiple hash tables. If the symbol does not already exist it is created and the initial value for the value field is `UNDEF` (undefined), for the property list `NIL` and the next field is used by the hash table. If there is a symbol with this name in the hash table it is simply returned.

```
L_Ptr intern(name,tbllst)
     L_Ptr name, tbllst;
```

`intern_soft(str,NIL)`: Same as `intern(str,NIL)` but if the symbol does not exist, it is not created.

```
L_Ptr intern_soft(str,tbllst)
     L_Ptr str, tbllst;
```

`make_symbol(str)`: Just creates a symbol with the name `str` without interning it. This means that if we call this function twice with the same arguments we get two different symbols, whereas `intern` will return the same symbol. Also note that if we loose track of an un`intern`ed symbol we cannot find it again. This function is used only for advanced programming using macros.

```
L_Ptr make_symbol(str)
     L_Ptr str;
```

NULLTYPE

Description:

The type of NIL. Cannot be accessed, cannot be modified, it is created only once during the initialization of a session. It should be seen as similar to `NULL` of standard C programming.

Typechecking:

`NULLP(t_obj)`: returns non-zero if `t_obj` is `NIL`, zero otherwise.

```
int NULLP(t_obj)
     L_Ptr t_obj;
```

SELECTTYPE

Description:

The type of a selection. Selection is the object that represents a generic function. It is an array that contains either selections or functions or `NIL`. For instance if the generic function dispatches on the two first arguments then it is an array of selections which in turn are arrays of functions. Every selection array can have a mix of selections, functions or `NIL`s, to dispatch on one or more arguments. A `NIL` means that there is no function defined for this combination of argument types if the `DEFAULTTYPE` entry is also `NIL`. If the `DEFAULTTYPE` entry is not `NIL` then this is the function that corresponds to the combination of argument types.

Typechecking:

`SELECTP(t_obj,sz)`: Returns non-zero if `t_obj` is a selection of size `sz`. If `sz` is zero there is no size checking.

```
int SELECTP(t_obj,sz)
     L_Ptr t_obj;
     int sz;
```

`selectp(t_obj)`: returns non-`NIL` if `t_obj` is a selection. This is the recommended way to typecheck.

```
L_Ptr selectp(t_obj)
     L_Ptr t_obj;
```

Access:

`XPTR(t_obj+typenum)`: Returns the selection for type `typenum`. `Typenum` is an integer that represents the corresponding type. It cannot be either `SPECIALTYPE` or `LARGE` since both are not single types but whole classes of types.

```
L_Ptr XPTR(t_obj+typenum)
     L_Ptr t_obj;
     int typenum;
```

`get_selection(t_obj,lst)`: `lst` is a list of integers (i1, i2, ...) representing `typenums` and `t_obj` is a selection. `Get_selection` retrieves the i1th element of `t_obj`, then the i2th element of that and so on until either the list is over or a non selection object is found which is returned. This is the recommended way to retrieve a selection.

```
L_Ptr get_selection(t_obj,lst_n)
     L_Ptr t_obj, lst_n;
```

Creation:

`add_selection(smbl,lst)`: this function is called in response to a *`generic`* definition. When called from within the MediaMath it does not evaluate its arguments, but if it is called from C behaves like an ordinary function. It assigns the selection object to `smbl`, but when `smbl` already contains a selection then this is updated. If `smbl` contains a function this function becomes the default, so whatever used to work before for this function continues to work.

```
L_Ptr add_selection(smbl,lst)
     L_Ptr smbl, lst;
```

DEFAULTTYPE

Description:

There is no instance of this type. It merely exists so that selection functions fall back to this when they find a `NIL`.

KEYTYPE

Description:

The type of the structure that represents key-value pairs. It appears in the argument list of functions that contain keys e.g. *`write_eps_img(img1, "face.eps", :dpi=2*res);`*. It is a structure that contains two fields: a symbol, in this case *`dpi`*, and a value, in this case the structure that represents the piece of code *`2*res`*.

Typechecking:

`KEYP(t_obj)`: returns non-zero if `t_obj` is a key.

```
int KEYP(t_obj)
     L_Ptr t_obj;
```

Access:

`XPTR(t_obj+pos)`: returns the symbol or the value of the key. `Pos` is either `KEY_NAME_POSTN` or `KEY_VALU_POSTN`. It can appear on the left side of an assignment statement.

```
L_Ptr XPTR(t_obj+pos)
     L_Ptr t_obj;
     int pos;
```

`KEY_SYMBOL(t_obj)` and `KEY_VALUE(t_obj)` return the symbol and the value of the key `t_obj`. They can appear on the left side of an assignment statement.

```
L_Ptr KEY_SYMBOL(t_obj)
     L_Ptr t_obj;
L_Ptr KEY_VALUE(t_obj)
     L_Ptr t_obj;
```

Creation:

`key_cons(smbl,t_obj)`: returns a new cell with `smbl` and `t_obj` as symbol and value.

```
L_Ptr key_cons(smbl,t_obj)
     L_Ptr smbl, t_obj;
```

RANGETYPE

Description:

The type of the structure that represents a range. It appears in the argument list of functions, or array dereferencing, that contain ranges e.g. *`mk_ftmpl( -2..2, [-1,2,-3,2,-1]);`*. It is a structure that contains two numbers in this case *`-2`* and *`2`*.

Typechecking:

`RANGEP(t_obj)`: returns non-zero if `t_obj` is a range.

```
int RANGEP(t_obj)
     L_Ptr t_obj;
```

Access:

`XPTR(t_obj+pos)`: returns the from and to part of the range. `Pos` is either `RANGE_INT1_POSTN` or `RANGE_INT2_POSTN`. It can appear on the left side of an assignment statement.

```
L_Ptr XPTR(t_obj+pos)
     L_Ptr t_obj;
     int pos;
```

`RANGE_INT1(t_obj)` and `RANGE_INT2(t_obj)` return the from and to numbers of the range. They can appear on the left side of an assignment statement.

```
L_Ptr RANGE_INT1(t_obj)
     L_Ptr t_obj;
L_Ptr RANGE_INT2(t_obj)
     L_Ptr t_obj;
```

Creation:

`range_cons(int1,int2)`: returns a new cell with `int1` and `int2` for from and to part of the range.

```
L_Ptr range_cons(int1,int2)
     L_Ptr int1, int2;
```

3. C objects

C objects contain a tag and a regular C type like integer, float, string (pointer to `NULL` terminated string), image or file pointer. C objects are again of two main types: small and large. Smaller or equal in size to a pointer is small and all else is large. Small ones are stored after the header and large ones are stored somewhere else and the pointer to this else is stored after the header. The large types are not defined in compilation type but in load time (e.g. when MediaMath is started or when a new module is loaded) so new large types can be defined dynamically. The extra `C_info` field has various uses: function types, macro types etc use it to store the minimum number of arguments. Large types use it to store the typenum, an index to an array of symbols that have all the needed info in their property lists. Every type has such a typenum.

```
typedef union L_Header *L_Ptr;

typedef union L_Header
{
...
  struct
    {
      unsigned int mrkbit:1;    /* GC marker bit                  */
      unsigned int usrmrkbit:1; /* Why not give one to the luser */
      unsigned int C_info:22;   /* Info about C_types (typenum)  */
      unsigned int type:8;      /* One of C_TYPE or L_TYPE       */
                                /* Always UNEVALED is on         */
                                /* The six first bits describe   */
                                /* the C data type               */
    } C;
...
} L_Header;
```

3.1. Typechecking, Accessing and Creating

For every C data type understood by MediaMath there is an accesser, typechecker and a creator. The first two are macros the later are functions.

Type	Typecheck	Accesser	Creator
`UNINT`	`UNINTEGERP`	`XUNINT`	`C_2_L_uninteger(uninteger)` `unsigned int uninteger;`
`INT`	`INTEGERP`	`XINT`	`C_2_L_integer(integer)` `int integer;`
`FLOAT`	`FLOATP`	`XFLOAT`	`C_2_L_float(fl)` `float fl;`
`UNCHAR`	`UNCHARP`	`XUNCHAR`	`C_2_L_unchar(unch)` `unsigned char unch;`
`CHAR`	`CHARP`	`XCHAR`	`C_2_L_char(ch)` `char ch;`
`STRNG`	`STRNGP`	`XSTRNG`	`C_2_L_string(str)` `char *str;`
`LARGE`	`LARGEP`	`XLARGE`	`C_2_L_large(C_obj,typenum)` `L_Ptr C_obj;` `int typenum;`
`FUNCTION`	`FUNCTIONP`	`XFUNCTION`	`C_2_L_function(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`
`VFUNCTION`	`VFUNCTIONP`	`XFUNCTION`	`C_2_L_function(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`
`SPECIALFRM`	`SPECIALFRMP`	`XSPECIALFRM`	`C_2_L_specialfrm(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`
`VSPECIALFRM`	`VSPECIALFRMP`	`XSPECIALFRM`	`C_2_L_specialfrm(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`
`MACRO`	`MACROP`	`XMACRO`	`C_2_L_macro(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`
`VMACRO`	`VMACROP`	`XMACRO`	`C_2_L_macro(fun,minarg,maxarg)` `L_Ptr (*fun)();` `int minarg, maxarg;`

Each one of the above types has the following use:

Type	Comment	Casted to
`UNINT`	unsigned integer	`unsigned int`
`INT`	signed integer	`int`
`FLOAT`	float	`float`
`UNCHAR`	unsigned character	`unsigned char`
`CHAR`	signed character	`char`
`STRNG`	NULL terminated string	`char *`
`LARGE`	anything larger than a pointer	`void *`
`FUNCTION`	Function with fixed number of args	`(L_Ptr (*)())`
`VFUNCTION`	Function with varying args	`(L_Ptr (*)())`
`SPECIALFRM`	Special fixed	`(L_Ptr (*)())`

`VSPECIALFRM`	Special varying	`(L_Ptr (*)())`
`MACRO`	Macro fixed	`(L_Ptr (*)())`
`VMACRO`	Macro varying	`(L_Ptr (*)())`

All the above types are very uniform in how they are accessed, typechecked and created. The only exception is the `LARGE` type that is not really a type but a class of types. All the type-checking macros accept one argument and return non zero if the argument is of the corresponding type. The accesser macros accept as argument a Lisp object and return the corresponding C object. The creator functions accept as argument a C type and return a lisp type.

The large types use the accesser and creator with a type cast (it is not necessary to use casts but it makes it more portable). The above mentioned `LARGEP` will return non zero if the argument is a large type, but we hardly ever need that, because an image and a vector are both large and `XLARGE` does not distinguish them.

```
int CHK_LIVETYPE(tname,t_obj)
     <typename> tname;
     L_Ptr t_obj;
int CHK_TYPE(tname,t_obj)
     <typename> tname;
     L_Ptr t_obj;
int ALIVE(t_obj)
     L_Ptr t_obj;
```

`CHK_LIVETYPE(tname,t_obj)` which returns non-nil if the type of `t_obj` is `tname` (`tname` does not end with `Typenum`; this is appended by the macro) and it is not as yet deallocated. If we want to check only if it is of the correct type we do `CHK_TYPE(tname,t_obj)`. We can check if it is still usable with `ALIVE(t_obj)` that returns zero if `t_obj` is deallocated.

```
int XLARGE_TYPENUM(t_obj)
     L_Ptr t_obj;
int int_get_typenum(t_obj)
    L_Ptr t_obj;
```

The typenum of a large object can be extracted with `XLARGE_TYPENUM(t_obj)`. The recommended way though is to use `int_get_typenum(t_obj)` that returns the `typenum` of `t_obj` no matter if it is large, integer or structure.

3.2. Converting

There are functions that convert between various kinds of numerical types. These are:

```
L_Ptr to_integer(num)
     L_Ptr num;
L_Ptr to_uninteger(num)
     L_Ptr num;
L_Ptr to_float(num)
     L_Ptr num;
L_Ptr to_char(num)
     L_Ptr num;
L_Ptr to_unchar(num)
     L_Ptr num;
```

All these functions accept one argument and return a tagged object of type integer, unsigned integer, float, character and unsigned character respectively. They will convert anything they can and issue an error if they cannot.

Quite often these functions are the preferred way to typecheck a function's arguments, because with one statement we can typecheck and convert and do it consistently.

4. Images and matrices

The image and matrix data types are defined in the `"image.so"` module. They are 18 different types. They can be classified in many different ways: According to dimensionality there are one (the first three columns below) or two dimensional (the last three columns).

fvec	fscln	ftmpl	fmat	fimg	ftmpl2
ivec	iscln	itmpl	imat	iimg	itmpl2
ucvec	ucscln	uctmpl	ucmat	ucimg	uctmpl2

According to the underlying data type: floating (first line above), integer (second line) and unsigned character (third line). And according to functionality: matrix-vector for linear algebra operations, image-scanline for image operations and 1d template - 2d template for convolutions.

There are a few conventions about these types. The two dimensional types have the vertical dimension and the horizontal dimension. So anything that relates to one of the two dimensions of the data structure has either a `v` or an `h` in front of it as in `vmax` or `hconvol_fimg_ftmpl`.

Another convention is that an image or a scanline of size N is from 0 to $N-1$ as in C, but a matrix or vector of size N is from 1 to N. And a template is from anything to anything.

All of the above data types are tagged objects of type large. That is they contain a pointer to a conventional C data structure that holds the image or the vector. There are six such data structures pointed to by pointers of type:

`fone_Dptr` used by:	fvec	fscln	ftmpl	
`ione_Dptr` used by:	ivec	iscln	itmpl	
`ucone_Dptr`	used by:	ucvec	ucscln	uctmpl
`ftwo_Dptr` used by:	fmat	fimg	ftmpl2	
`itwo_Dptr` used by:	imat	iimg	itmpl2	
`uctwo_Dptr`	used by:	ucmat	ucimg	uctmpl2

We have three different tagged objects sharing the same C structure so that operations like multiplication can work differently on matrices and images.

The C structures that hold the floating point image or matrix data are:

```
typedef struct fone_D *fone_Dptr;
typedef struct ftwo_D *ftwo_Dptr;

typedef struct fone_D
{
  int vmin, vmax;                /* the vector is a[vmin]..a[vmax] */
                                 /* so it has length vmax-vmin+1   */

  float *fdata;                  /* pointer to vmin positions      */
                                 /* before the beginning of the    */
                                 /* array                          */
} fone_D;

typedef struct ftwo_D
{
  int vmin, vmax;
  int hmin, hmax;                /* the upper left and lower right */
```

```
                                 /* corners of the 2-D vector are  */
                                 /* a[vmin][hmin] and a[vmax][hmax]*/

  float **fdata;                 /* a pointer that points vmin     */
                                 /* positions before the beginning */
                                 /* of the array of pointers that  */
                                 /* point hmin positions before the*/
                                 /* beginning of every row.        */
} ftwo_D;
```

The reason that the pointers point `vmin` or `hmin` positions before the actual data is to circumvent `a[0]..a[n-1]` convention of C. So if pointer `a` points `vmin` positions before the actual first element of an array then `a[vmin]` will be the first element of the array. This way we can have arrays `a[vmin]..a[vmax]` for arbitrary `vmin` and `vmax` and the index of the first element of a vector or a template can be different than zero. This technique might not be portable to architectures that use paged segmented memory, because ANSI C does not require it.

The rows in the 2D structure are allocated all at once from a contiguous space. If one wants to visit the whole matrix/image can start from `&a[vmin][hmin]` and increment until the end of the whole matrix.

The type definition for unsigned character structures and integer structures is similar.

```
typedef struct ucone_D *ucone_Dptr;
typedef struct uctwo_D *uctwo_Dptr;

typedef struct ucone_D
{
  int vmin, vmax;                /* the vector is a[vmin]..a[vmax] */
                                 /* so it has length vmax-vmin+1   */

  unsigned char *ucdata;         /* pointer to vmin positions      */
                                 /* before the beginning of the    */
                                 /* array                          */
} ucone_D;

typedef struct uctwo_D
{
  int vmin, vmax;
  int hmin, hmax;                 /* the upper left and lower right */
                                  /* corners of the 2-D vector are  */
                                  /* a[vmin][hmin] and a[vmax][hmax]*/

  unsigned char **ucdata;         /* a pointer that points vmin     */
                                  /* positions before the beginning */
                                  /* of the array of pointers that  */
                                  /* point hmin positions before the*/
                                  /* beginning of every row.        */
} uctwo_D;


typedef struct ione_D *ione_Dptr;
typedef struct itwo_D *itwo_Dptr;
typedef struct ione_D
{
  int vmin, vmax;               /* the vector is a[vmin]..a[vmax] */
                                /* so it has length vmax-vmin+1   */

  int *idata;                   /* pointer to vmin positions      */
                                /* before the beginning of the    */
                                /* array                          */
} ione_D;

typedef struct itwo_D
{
  int vmin, vmax;
  int hmin, hmax;               /* the upper left and lower right */
                                /* corners of the 2-D vector are  */
                                /* a[vmin][hmin] and a[vmax][hmax]*/

  int **idata;                  /* a pointer that points vmin     */
                                /* positions before the beginning */
                                /* of the array of pointers that  */
                                /* point hmin positions before the*/
                                /* beginning of every row.        */
} itwo_D;
```

4.1. Accessing the first and the last

There are a few macros that return the address of the first and the last element or pixel of a matrix or image.

```
unsigned char *UC2DFIRST(twoD)
    uctwo_Dptr twoD;
unsigned char *UC2DLAST(twoD)
    uctwo_Dptr twoD;
int *I2DFIRST(twoD)
    itwo_Dptr twoD;
int *I2DLAST(twoD)
    itwo_Dptr twoD;
float *F2DFIRST(twoD)
    ftwo_Dptr twoD;
float *F2DLAST(twoD)
    ftwo_Dptr twoD;
unsigned char *UC1DFIRST(oneD)
    ucone_Dptr oneD;
unsigned char *UC1DLAST(oneD)
    ucone_Dptr oneD;
int *I1DFIRST(oneD)
    ione_Dptr oneD;
int *I1DLAST(oneD)
    ione_Dptr oneD;
float *F1DFIRST(oneD)
    fone_Dptr oneD;
float *F1DLAST(oneD)
    fone_Dptr oneD;
```

The argument has to be a pointer to the C structure, not a tagged object.

4.2. Converting

Most types can be converted from one to the other. The functions that do that are:

```
to_fvec     to_fscln    to_ftmpl
to_fmat     to_fimg     to_ftmpl2
to_ivec     to_iscln    to_itmpl
to_imat     to_iimg     to_itmpl2
to_ucvec    to_ucscln   to_uctmpl
to_ucmat    to_ucimg    to_uctmpl2
```

These turn to the corresponding type whatever can be turned. All of them accept as argument a tagged type, and return another tagged type.

```
L_Ptr to_fvec(t_obj)
     L_Ptr t_obj;
L_Ptr to_fscln(t_obj)
     L_Ptr t_obj;
etc...
```

4.3. Creating and destroying

There are functions to create and destroy images and matrices.

`mksimple_fmat(vdim,hdim)`: Allocates an matrix with `vdim` rows and `hdim` columns.

```
L_Ptr mksimple_fmat(vdim,hdim)
     L_Ptr vdim, hdim;
```

`mksimple_fimg(vdim,hdim)`: Allocates an image with `vdim` rows and `hdim` columns.

```
L_Ptr mksimple_fimg(vdim,hdim)
     L_Ptr vdim, hdim;
```

`mksimple_ftmpl2(vmin,vmax,hmin,hmax)`: Allocates an 2D template from `vmin` to `vmax` and `hmin` to `hmax`.

```
L_Ptr mksimple_ftmpl2(vmin,vmax,
                                 hmin,hmax)
     L_Ptr vmin,vmax,hmin,hmax;
```

`mksimple_fvec(vdim)`: Allocates a vector of floats with `vdim` elements.

```
L_Ptr mksimple_fvec(vdim)
     L_Ptr vdim;
```

`mksimple_ftmpl(vmin,vmax)`: Allocates a template of floats from `vmin` to `vmax`.

```
L_Ptr mksimple_ftmpl(vmin,vmax)
     L_Ptr vmin,vmax;
```

`mksimple_fscln(vdim)`: Allocates a scanline of floats with `vdim` elements.

```
L_Ptr mksimple_fscln(vdim)
     L_Ptr vdim;
```

The functions above, are for floating point. There is a set for unsigned characters and one for integers. Just replace the `f` with `uc` or `i`.

There is a set of routines to deallocate images, matrices etc, and can be called explicitly. Normaly these are called by the garbage collector.

```
free_fvec      free_fscln      free_ftmpl
free_fmat      free_fimg       free_ftmpl2
free_ivec      free_iscln      free_itmpl
free_imat      free_iimg       free_itmpl2
free_ucvec     free_ucscln     free_uctmpl
free_ucmat     free_ucimg      free_uctmpl2
```

All of them accept as argument the tagged object we want to discard and return `NIL`. All parts of the abject that were obtained with `malloc()` are freed and the object is marked so that the macro `ALIVE` returns 0.

The casual writer of C modules will use a fraction only of the above macros and functions. It is better if we see their use with a few examples.

5. Examples

5.1. Thresholding an image

We can study how to accept arguments, create arrays that return the result etc. by having a look at `thresh_fimg`. Like any function that is accessible from MediaMath, it has a header that can be created on emacs by typing `META-x`. This will create a header that we can fill up.

It is a good practice to separate the driver function from the actual function that does the work. `thresh_fimg` is just the driver function and does all the error checking and data type manipulation.

```
/*awkstart***************************************************************
Name: thresh_fimg
MinArgs: 2
MaxArgs: 2
Type: function
Synopsis: thresh_fimg(<img>,<number>)
Doc: Threshold an image of floats.
Doc: Returns an image of unsigned characters of the same size as <img>
Doc: that is 1 where <img> exceeds <number> and 0 everywhere else.
Doc: All operations are in float.
SEE: max_fimg, min_fimg
Example: thresh_fimg(img1,10);
***********************************************************awkstop*/

L_Ptr thresh_fimg(img,lnum)
     L_Ptr img,lnum;
{
  L_Ptr res;
  int vd, hd;
  ftwo_Dptr fimg;
  float num;

  img = to_fimg(img);
  lnum = to_float(lnum);

  num = XFLOAT(lnum);
  fimg = (ftwo_Dptr)XLARGE(img);

  vd = fimg->vmax-fimg->vmin+1;
  hd = fimg->hmax-fimg->hmin+1;
  res = mksimple_ucimg(C_2_L_integer(vd),C_2_L_integer(hd));

  ftwo_D_thresh(fimg, num, (uctwo_Dptr) XLARGE(res));
  return res;
}
```

Every well behaved function that is callable from MediaMath should typecheck its arguments. If any argument is not among the kinds we expect then we issue an error. Then we convert every argument to the most convenient type. If we want for instance a number, we can accept an integer, a float, an unsigned character etc, and then we convert to the type we really want: a float. To make life easier we just call `to_float` to convert it to float (if it cannot be converted then function `to_float` will issue an error). The same goes for images. This explains the first two executable lines.

If all went well then we extract the contents of these two objects. Remember, everything that is declared `L_Ptr` is a whole data structure that contains tags etc. If the tag says there is an integer inside or an image we have to get it out. We can use the accesser macros to do that. The `XLARGE` macro requires casting because it can be used for many types including images, matrices and vectors. And that is what the next two lines do.

The next three lines create the image that will store the result. They extract the dimensions of the image (notice that although we know that for an image `fimg->vmin` is zero, we still do the subtraction to avoid having to rewrite it if the definition of an image ever becomes more general) and then call `mksimple_ucimg`. This is a function that can be called from the MediaMath interpreter also so needs tagged objects as arguments.

After we do all these we pass the proper arguments to a conventional C routine that knows nothing about MediaMath tags and headers. We also provide the space to it to store the result (again using `XLARGE`). And then we return the result.

The function `ftwo_D_thresh` does the actual work. It has minimum interference with the rest of MediaMath and can be used in other programs easily. It does not do any error checking other than things that cannot be checked out by the driver function above (in this case there is nothing that cannot be checked by the calling function.

```
int  ftwo_D_thresh(fimg,num,res)
     ftwo_Dptr fimg;
     uctwo_Dptr res;
     float num;
{
  int vmin, vmax, hmin, hmax;
  int i,j;
  float *ff;
  unsigned char *fr;

  vmin = fimg->vmin;
  vmax = fimg->vmax;
  hmin = fimg->hmin;
  hmax = fimg->hmax;

  ff = F2DFIRST(fimg);
  fr = UC2DFIRST(res);

  for (i=vmin; i<=vmax; i++)
```

```
    for (j=hmin; j<=hmax; j++)
      {
        if (num > *(ff++)) *(fr++) = 0;
        else *(fr++) = 1;
      }
  return 0;
}
```

The first thing the function does is retrieve the sizes of the image. The first 4 statements do just this. The next two get the address of the first pixel of both the source image and the resulting image. Both of them have the same number of pixels (the driver function made sure of that). This is what the two next statements do. The next set of statements is the double for-loop that does the job. It simply scans the two images at the same time.

The macros `F2DFIRST` and `UC2DFIRST` return the address of the first pixel. It is guaranteed that the rows of an image occupy consecutive places in memory, so for simple operations we can just scan from the top left to the bottom right image continuously.

5.2. Norm of an image

Another example is the `norm_fimg` function. It accepts as argument an image and returns a floating point number.

```
/*awkstart*************************************************************
Name: norm_fimg
MinArgs: 1
MaxArgs: 1
Type: function
Synopsis: norm_fimg(<fimg>)
Doc: Returns the Frobenious norm of <fimg>.
SEE: gnorm
Example: gnorm(mk_fimg(2,3,[[1,2,3],[4,5,6]]));
**************************************************************awkstop*/

L_Ptr norm_fimg(fimg)
     L_Ptr fimg;
{
  if (!CHK_LIVETYPE(Sfimg,fimg))
    error_signal(Snorm_fimg,SWrongTypeArg,NIL,fimg);

  return C_2_L_float(ftwo_D_norm((ftwo_Dptr)XLARGE(fimg)));
}
```

The driver of this function is much simpler because we need not allocate any image or other structure. The function only checks if the argument is an image of floating point numbers and it is alive (not deallocated). It then calls `ftwo_D_norm` to do the work which returns a float which is passed to `C_2_L_float` to put a tag on it. That’s it.

Function `ftwo_D_norm` is no more difficult. It accepts as argument a two dimensional function and returns a float (the argument in this function is `mat` because it is also used for the matrix norm (Frobenious)).

```
float ftwo_D_norm(mat)
     ftwo_Dptr mat;
{
  int i,j;
  int imin, imax, jmin, jmax;
  register float *ff1, temp, res;

  imin = mat->vmin;
  imax = mat->vmax;
  jmin = mat->hmin;
  jmax = mat->hmax;
  ff1  = F2DFIRST(mat);

  res = 0;
  for (i=imin; i<=imax; i++)
    for (j=jmin; j<=jmax; j++)
      {
                   temp = *(ff1++);
                   res += temp*temp;
      }
  return sqrt(res);
}
```

The function retrieves the bounds of the array and the address of the first element and the scans the whole array accumulating the result in the variable `res`. It then returns the square root of `res`.

5.3. Transpose a matrix

The transpose routine is a bit tricky because the transpose of a matrix might be a vector when the matrix has one row and many columns.

```
/*awkstart***************************************************************
Name: transp_fmat
MinArgs: 1
MaxArgs: 1
Type: function
Synopsis: transp_fmat(<fmat>)
Doc: Returns the transpose of a matrix.
SEE: gtranspose
Example: mk_fmat(2,2,[[1,2],[3,4]])^T;
**************************************************************awkstop*/

L_Ptr transp_fmat(fmat)
     L_Ptr fmat;
{
  ftwo_Dptr mat;
  L_Ptr res;
  int vd, hd;

  if (!CHK_LIVETYPE(Sfmat,fmat))
    error_signal(Stransp_fmat,SWrongTypeArg,NIL,fmat);
  mat = (ftwo_Dptr)XLARGE(fmat);

  vd = mat->vmax-mat->vmin+1;
  hd = mat->hmax-mat->hmin+1;
  if ( vd==1 )
    {
      res = mksimple_fvec(C_2_L_integer(hd));
      ftwo_D_transpose1(mat,(fone_Dptr)XLARGE(res));
    }
  else
    {
      res = mksimple_fmat(C_2_L_integer(hd),C_2_L_integer(vd));
      ftwo_D_transpose(mat,(ftwo_Dptr)XLARGE(res));
    }
  return res;
}
```

After we check if the type is what we expect, we extract the pointer to the C structure and we get the size of the matrix. We check the number of rows to decide which version of `ftwo_D_transpose` we call. The allocation is done inside the `if` because we need to allocate different structures in each case.

The other interesting thing about `transp_fmat` is that `ftwo_D_transpose` scans the array in two different ways: by incrementing the pointer starting from the beginning of the array and by explicit array references.

```
int ftwo_D_transpose(mat,res)
     ftwo_Dptr mat, res;
{
  int i,j;
  int imin,jmin,imax,jmax;
  float *ff1, **fr;

  imin = mat->vmin;
  imax = mat->vmax;
  jmin = mat->hmin;
  jmax = mat->hmax;

  ff1 = F2DFIRST(mat);
  fr  = res->fdata;

  for (i=imin; i<=imax; i++)
    for (j=jmin; j<=jmax; j++)
      fr[j][i] = *(ff1++);
  return 0;
}
```

One of the two vectors has to be scanned column by column, so we have to use array referencing for at least one of them.

Table of Contents