Data Types, Functions, and Programming

Daniel R. Grayson* and Michael E. Stillman**

In this chapter we present an introduction to the structure of *Macaulay 2* commands and the writing of functions in the *Macaulay 2* language. For further details see the *Macaulay 2* manual distributed with the program [1].

1 Basic Data Types

The basic data types of *Macaulay 2* include numbers of various types (integers, rational numbers, floating point numbers, complex numbers), lists (basic lists, and three types of visible lists, depending on the delimiter used), hash tables, strings of characters (both 1-dimensional and 2-dimensional), Boolean values (true and false), symbols, and functions. Higher level types useful in mathematics are derived from these basic types using facilities provided in the *Macaulay 2* language. Except for the simplest types (integers and Boolean values), *Macaulay 2* normally displays the type of the output value on a second labeled output line.

Symbols have a name which consists of letters, digits, or apostrophes, the first of which is a letter. Values can be assigned to symbols and recalled later.

```
i1 : w

o1 = w

o1 : Symbol

i2 : w = 2^100

o2 = 1267650600228229401496703205376

i3 : w

o3 = 1267650600228229401496703205376
```

Multiple values can be assigned in parallel.

```
i4 : (w,w') = (33,44)

o4 = (33, 44)

o4 : Sequence

i5 : w

o5 = 33

i6 : w'

o6 = 44
```

* Supported by NSF grant DMS 99-70085.
** Supported by NSF grant 99-70348.

Comments are initiated by -- and extend to the end of the line.

```
i7 : (w,w') = (33,   -- this is a comment
               44)

o7 = (33, 44)

o7 : Sequence
```

Strings of characters are delimited by quotation marks.

```
i8 : w = "abcdefghij"

o8 = abcdefghij
```

They may be joined horizontally to make longer strings, or vertically to make a two-dimensional version called a *net*.

```
i9 : w | w

o9 = abcdefghijabcdefghij

i10 : w || w

o10 = abcdefghij
      abcdefghij
```

Nets are used in the preparation of two dimensional output for polynomials.

Floating point numbers are distinguished from integers by the presence of a decimal point, and rational numbers are entered as fractions.

```
i11 : 2^100

o11 = 1267650600228229401496703205376

i12 : 2.^100

o12 = 1.26765 10^30

o12 : RR

i13 : (36 + 1/8)^6

      582622237229761
o13 = ---------------
           262144

o13 : QQ
```

Parentheses, braces, and brackets are used as delimiters for the three types of *visible lists*: lists, sequences, and arrays.

```
i14 : x1 = {1,a}

o14 = {1, a}

o14 : List

i15 : x2 = (2,b)

o15 = (2, b)

o15 : Sequence

i16 : x3 = [3,c,d,e]

o16 = [3, c, d, e]

o16 : Array
```

Even though they use braces, lists should not be confused with sets, which will be treated later. A double period can be used to construct a sequence of consecutive elements in various contexts.

```
i17 : 1 .. 6

o17 = (1, 2, 3, 4, 5, 6)

o17 : Sequence

i18 : a .. f

o18 = (a, b, c, d, e, f)

o18 : Sequence
```

Lists can be nested.

```
i19 : xx = {x1,x2,x3}

o19 = {{1, a}, (2, b), [3, c, d, e]}

o19 : List
```

The number of entries in a list is provided by `#`.

```
i20 : #xx

o20 = 3
```

The entries in a list are numbered starting with 0, and can be recovered with `#` used as a binary operator.

```
i21 : xx#0

o21 = {1, a}

o21 : List

i22 : xx#0#1

o22 = a

o22 : Symbol
```

We can join visible lists and *append* or *prepend* an element to a visible list. The output will be the same type of visible list that was provided in the input: a list, a sequence, or an array; if the arguments are various types of lists, the output will be same type as the first argument.

```
i23 : join(x1,x2,x3)

o23 = {1, a, 2, b, 3, c, d, e}

o23 : List


i24 : append(x3,f)

o24 = [3, c, d, e, f]

o24 : Array
```

```
i25 : prepend(f,x3)

o25 = [f, 3, c, d, e]

o25 : Array
```

Use `sum` or `product` to produce the sum or product of all the elements in a list.

```
i26 : sum {1,2,3,4}

o26 = 10


i27 : product {1,2,3,4}

o27 = 24
```

2 Control Structures

Commands for later execution are encapsulated in *functions*. A function is created using the operator `->` to separate the parameter or sequence of parameters from the code to be executed later. Let's try an elementary example of a function with two arguments.

```
i28 : f = (x,y) -> 1000 * x + y

o28 = f

o28 : Function
```

The parameters `x` and `y` are symbols that will acquire a value later when the function is executed. They are *local* in the sense that they are completely different from any symbols with the same name that occur elsewhere. Additional local variables for use within the body of a function can be created by assigning a value to them with `:=` (first time only). We illustrate this by rewriting the function above.

```
i29 : f = (x,y) -> (z := 1000 * x; z + y)

o29 = f

o29 : Function
```

Let's apply the function to some arguments.

```
i30 : f(3,7)

o30 = 3007
```

The sequence of arguments can be assembled first, and then passed to the function.

```
i31 : s = (3,7)

o31 = (3, 7)

o31 : Sequence
```

```
i32 : f s

o32 = 3007
```

As above, functions receiving one argument may be called without parentheses.

```
i33 : sin 2.1

o33 = 0.863209

o33 : RR
```

A compact notation for functions makes it convenient to apply them without naming them first. For example, we may use `apply` to apply a function to every element of a list and to collect the results into a list.

```
i34 : apply(1 .. 10, i -> i^3)

o34 = (1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)

o34 : Sequence
```

The function `scan` will do the same thing, but discard the results.

```
i35 : scan(1 .. 5, print)
1
2
3
4
5
```

Use `if ... then ... else ...` to perform alternative actions based on the truth of a condition.

```
i36 : apply(1 .. 10, i -> if even i then 1000*i else i)

o36 = (1, 2000, 3, 4000, 5, 6000, 7, 8000, 9, 10000)

o36 : Sequence
```

A function can be terminated prematurely with `return`.

```
i37 : apply(1 .. 10, i -> (if even i then return 1000*i; -i))

o37 = (-1, 2000, -3, 4000, -5, 6000, -7, 8000, -9, 10000)

o37 : Sequence
```

Loops in a program can be implemented with `while ... do ...`.

```
i38 : i = 1; while i < 50 do (print i; i = 2*i)
1
2
4
8
16
32
```

Another way to implement loops is with `for` and `do` or `list`, with optional clauses introduced by the keywords `from`, `to`, and `when`.

```
i40 : for i from 1 to 10 list i^3

o40 = {1, 8, 27, 64, 125, 216, 343, 512, 729, 1000}

o40 : List
```

```
i41 : for i from 1 to 4 do print i
1
2
3
4
```

A loop can be terminated prematurely with `break`, which accepts an optional value to return as the value of the loop expression.

```
i42 : for i from 2 to 100 do if not isPrime i then break i

o42 = 4
```

If no value needs to be returned, the condition for continuing can be provided with the keyword `when`; iteration continues only as long as the predicate following the keyword returns `true`.

```
i43 : for i from 2 to 100 when isPrime i do print i
2
3
```

3 Input and Output

The function `print` can be used to display something on the screen.

```
i44 : print 2^100
1267650600228229401496703205376
```

For example, it could be used to display the elements of a list on separate lines.

```
i45 : (1 .. 5) / print;
1
2
3
4
5
```

The operator `<<` can be used to display something on the screen, without the newline character.

```
i46 : << 2^100
1267650600228229401496703205376
o46 = stdio

o46 : File

  --  the standard input output file
```

Notice the value returned is a *file*. A *file* in *Macaulay 2* is a data type that represents a channel through which data can be passed, as input, as output, or in both directions. The file `stdio` encountered above corresponds to your shell window or terminal, and is used for two-way communication between the program and the user. A file may correspond to what one usually calls a file, i.e., a sequence of data bytes associated with a given name and stored on your disk drive. A file may also correspond to a *socket*, a channel for communication with other programs over the network.

Files can be used with the binary form of the operator `<<` to display something else on the same line.

```
i47 : << "the value is : " << 2^100
the value is : 1267650600228229401496703205376
o47 = stdio

o47 : File

  --  the standard input output file
```

Using `endl` to represent the new line character or character sequence, we can produce multiple lines of output.

```
i48 : << "A = " << 2^100 << endl << "B = " << 2^200 << endl;
A = 1267650600228229401496703205376
B = 1606938044258990275541962092341162602522202993782792835301376
```

We can send the same output to a disk file named `foo`, but we must remember to close it with `close`.

```
i49 : "foo" << "A = " << 2^100 << endl << close

o49 = foo

o49 : File
```

The contents of the file can be recovered as a string with `get`.

```
i50 : get "foo"

o50 = A = 1267650600228229401496703205376
```

If the file contains valid *Macaulay 2* commands, as it does in this case, we can execute those commands with `load`.

```
i51 : load "foo"
```

We can verify that the command took effect by evaluating `A`.

```
i52 : A

o52 = 1267650600228229401496703205376
```

Alternatively, if we want to see those commands and the output they produce, we may use `input`.

```
i53 : input "foo"

i54 : A = 1267650600228229401496703205376

o54 = 1267650600228229401496703205376

i55 :
```

Let's set up a ring for computation in *Macaulay 2*.

```
i56 : R = QQ[x,y,z]

o56 = R

o56 : PolynomialRing


i57 : f = (x+y)^3

       3      2         2    3
o57 = x  + 3x y + 3x*y  + y

o57 : R
```

Printing, and printing to files, works for polynomials, too.

```
i58 : "foo" << f << close;
```

The two-dimensional output is readable by humans, but is not easy to convert back into a polynomial.

```
i59 : get "foo"

o59 =  3     2        2    3
      x  + 3x y + 3x*y  + y
```

Use `toString` to create a 1-dimensional form of the polynomial that can be stored in a file in a format readable by *Macaulay 2* and by other symbolic algebra programs, such as *Mathematica* or *Maple*.

```
i60 : toString f

o60 = x^3+3*x^2*y+3*x*y^2+y^3
```

Send it to the file.

```
i61 : "foo" << toString f << close;
```

Get it back.

```
i62 : get "foo"

o62 = x^3+3*x^2*y+3*x*y^2+y^3
```

Convert the string back to a polynomial with `value`, using `oo` to recover the value of the expression on the previous line.

```
i63 : value oo

       3     2        2    3
o63 = x  + 3x y + 3x*y  + y

o63 : R
```

The same thing works for matrices, and a little more detail is provided by `toExternalString`, if needed.

```
i64 : vars R

o64 = | x y z |

               1       3
o64 : Matrix R  <--- R

i65 : toString vars R

o65 = matrix {{x, y, z}}

i66 : toExternalString vars R

o66 = map(R^{{0}}, R^{{-1}, {-1}, {-1}}, {{x, y, z}})
```

4 Hash Tables

Recall how one sets up a quotient ring for computation in *Macaulay 2*.

```
i67 : R = QQ[x,y,z]/(x^3-y)

o67 = R

o67 : QuotientRing

i68 : (x+y)^4

        2 2       3    4           2
o68 = 6x y  + 4x*y  + y  + x*y + 4y

o68 : R
```

How does *Macaulay 2* represent a ring like R in the computer? To answer that, first think about what sort of information needs to be retained about R. We may need to remember the coefficient ring of R, the names of the variables in R, the monoid of monomials in the variables, the degrees of the variables, the characteristic of the ring, whether the ring is commutative, the ideal modulo which we are working, and so on. We also may need to remember various bits of code: the code for performing the basic arithmetic operations, such as addition and multiplication, on elements of R; the code for preparing a readable representation of an element of R, either 2-dimensional (with superscripts above the line and subscripts below), or 1-dimensional. Finally, we may want to remember certain things that take a lot of time to compute, such as the Gröbner basis of the ideal.

A *hash table* is, by definition, a way of representing (in the computer) a function whose domain is a finite set. In *Macaulay 2*, hash tables are extremely flexible: the elements of the domain (or *keys*) and the elements of the range (or *values*) of the function may be any of the other objects represented in the computer. It's easy to come up with uses for functions whose domain is finite: for example, a monomial can be represented by the function that associates to a variable its nonzero exponent; a polynomial can be represented by a function that associates to a monomial its nonzero coefficient; a set can be represented by any function with that set as its domain; a (sparse) matrix can be represented as a function from pairs of natural numbers to the corresponding nonzero entry.

Let's create a hash table and name it.

```
i69 : f = new HashTable from { a=>444, Daniel=>555, {c,d}=>{1,2,3,4}}

o69 = HashTable{{c, d} => {1, 2, 3, 4}}
                a => 444
                Daniel => 555

o69 : HashTable
```

The operator `=>` is used to represent a key-value pair. We can use the operator `#` to recover the value from the key.

```
i70 : f#Daniel

o70 = 555

i71 : f#{c,d}

o71 = {1, 2, 3, 4}
```

```
o71 : List
```

If the key is a symbol, we can use the operator `.` instead; this is convenient if the symbol has a value that we want to ignore.

```
i72 : Daniel = a

o72 = a

o72 : Symbol

i73 : f.Daniel

o73 = 555
```

We can use `#?` to test whether a given key occurs in the hash table.

```
i74 : f#?a

o74 = true

i75 : f#?c

o75 = false
```

Finite sets are implemented in *Macaulay 2* as hash tables: the elements of the set are stored as the keys in the hash table, with the accompanying values all being 1. (Multisets are implemented by using values larger than 1, and are called *tallies*.)

```
i76 : x = set{1,a,{4,5},a}

o76 = Set {{4, 5}, 1, a}

o76 : Set


i77 : x#?a

o77 = true

i78 : peek x

o78 = Set{{4, 5} => 1}
          1 => 1
          a => 1

i79 : y = tally{1,a,{4,5},a}

o79 = Tally{{4, 5} => 1}
            1 => 1
            a => 2

o79 : Tally

i80 : y#a

o80 = 2
```

We might use `tally` to tally how often a function attains its various possible values. For example, how often does an integer have 3 prime factors? Or 4? Use `factor` to factor an integer.

```
i81 : factor 60

        2
o81 = 2 3*5

o81 : Product
```

Then use # to get the number of factors.

```
i82 : # factor 60

o82 = 3
```

Use `apply` to list some values of the function.

```
i83 : apply(2 .. 1000, i -> # factor i)

o83 = (1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2,  ···

o83 : Sequence
```

Finally, use `tally` to summarize the results.

```
i84 : tally oo

o84 = Tally{1 => 193}
            2 => 508
            3 => 275
            4 => 23

o84 : Tally
```

Hash tables turn out to be convenient entities for storing odd bits and pieces of information about something in a way that's easy to think about and use. In *Macaulay 2*, rings are represented as hash tables, as are ideals, matrices, modules, chain complexes, and so on. For example, although it isn't a documented feature, the key `ideal` is used to preserve the ideal that was used above to define the quotient ring `R`, as part of the information stored in `R`.

```
i85 : R.ideal

             3
o85 = ideal(x  - y)

o85 : Ideal of QQ [x, y, z]
```

The preferred and documented way for a user to recover this information is with the function `ideal`.

```
i86 : ideal R

             3
o86 = ideal(x  - y)

o86 : Ideal of QQ [x, y, z]
```

Users who want to introduce a new high-level mathematical concept to *Macaulay 2* may learn about hash tables by referring to the *Macaulay 2* manual [1].

5 Methods

You may use the `code` command to locate the source code for a given function, at least if it is one of those functions written in the *Macaulay 2* language. For example, here is the code for `demark`, which may be used to put commas between strings in a list.

```
i87 : code demark

o87 = -- ../../../m2/fold.m2:23
      demark = (s,v) -> concatenate between(s,v)
```

The code for tensoring a ring map with a module can be displayed in this way.

```
i88 : code(symbol **, RingMap, Module)

o88 = -- ../../../m2/ringmap.m2:294-298
      RingMap ** Module := Module => (f,M) -> (
           R := source f;
           S := target f;
           if R =!= ring M then error "expected module over source ring";
           cokernel f(presentation M));
```

The code implementing the `ideal` function when applied to a quotient ring can be displayed as follows.

```
i89 : code(ideal, QuotientRing)

o89 = -- ../../../m2/quotring.m2:7
      ideal QuotientRing := R -> R.ideal
```

Notice that it uses the key `ideal` to extract the information from the ring's hash table, as you might have guessed from the previous discussion. The bit of code displayed above may be called a *method* as a way of indicating that several methods for dealing with various types of arguments are attached to the function named `ideal`. New such *method functions* may be created with the function `method`. Let's illustrate that with an example: we'll write a function called `denom` which should produce the denominator of a rational number. When applied to an integer, it should return 1. First we create the method function.

```
i90 : denom = method();
```

Then we tell it what to do with an argument from the class `QQ` of rational numbers.

```
i91 : denom QQ := x -> denominator x;
```

And also what to do with an argument from the class `ZZ` of integers.

```
i92 : denom ZZ := x -> 1;
```

Let's test it.

```
i93 : denom(5/3)

o93 = 3

i94 : denom 5

o94 = 1
```

6 Pointers to the Source Code

A substantial part of *Macaulay 2* is written in the same language provided to the users. A good way to learn more about the *Macaulay 2* language is to peruse the source code that comes with the system in the directory `Macaulay2/m2`. Use the `code` function, as described in the previous section, for locating the bit of code you wish to view.

The source code for the interpreter of the *Macaulay 2* language is in the directory `Macaulay2/d`. It is written in another language designed to be mostly type-safe, which is translated into `C` by the translator whose own `C` source code is in the directory `Macaulay2/c`. Here is a sample line of code from the file `Macaulay2/d/tokens.d`, which shows how the translator provides for allocation and initialization of dynamic data structures.

```
globalFrame := Frame(dummyFrame,globalScope.seqno,Sequence(nullE));
```

And here is the `C` code produced by the translator.

```
tokens_Frame tokens_globalFrame;
tokens_Frame tmp__23;
Sequence tmp__24;
tmp__24 = (Sequence) GC_MALLOC(sizeof(struct S259_)+(1-1)*sizeof(Expr));
if (0 == tmp__24) outofmem();
tmp__24->len_ = 1;
tmp__24->array_[0] = tokens_nullE;
tmp__23 = (tokens_Frame) GC_MALLOC(sizeof(struct S260_));
if (0 == tmp__23) outofmem();
tmp__23->next = tokens_dummyFrame;
tmp__23->scopenum = tokens_globalScope->seqno;
tmp__23->values = tmp__24;
tokens_globalFrame = tmp__23;
```

The core algebraic algorithms constitute the *engine* of *Macaulay 2* and are written in `C++`, with the source files in the directory `Macaulay2/e`. In the current version of the program, the interface between the interpreter and the core algorithms consists of a single two-directional stream of bytes. The manual that comes with the system [1] describes the engine communication protocol used in that interface.

References

1. Daniel R. Grayson and Michael E. Stillman: *Macaulay 2*, a software system for research in algebraic geometry and commutative algebra. Available in source code form and compiled for various architectures, with documentation, at http://www.math.uiuc.edu/Macaulay2/.

Index