The Secret of Learning M

Gaining a Major Edge in Job Effectiveness

by Erik Zoltán

The Challenge

Today it seems everyone is looking for M programmers, and many non-programmers frequently need to read M source code. There aren't enough experienced people to go around, so many organizations are investing more heavily in M training. Not all of the people being trained have programmed before. But instruction is critical: highly-trained professionals are highly productive, gaining a major edge in job effectiveness. I have been teaching the M language for six years, and there's no great mystery to developing (or evaluating) effective instruction techniques.

M has always had a reputation for being ***easy to learn***. This is generally true—it has a relatively small number of highly-flexible language elements, so there's much less to memorize than in C, for example. Some concepts, such as the rigid language syntax and the ultra-flexible sparse array data structure, are admittedly counter-intuitive to programmers coming to M from other languages. However, the only truly difficult aspect of the M language is indirection, one of its most powerful and advanced features.

But if you think that ***teaching*** M is easy, then think again. Many of the people learning M today have little programming experience: they have to learn M and programming ***in one week!*** My best teaching methods have been developed after my usual approach failed to get the job done for a specific topic on a certain day. That's the most generative classroom moment, when surprising new innovations become possible.

The Grandmother Principle

Teaching *anything* is a tricky business: when you thoroughly understand something, it becomes more difficult to talk to those with difficulty comprehending it. What seems simple to the teacher is often unclear to some students. Many people who are not professional instructors underestimate the difficulties involved.

I like to use the "Grandmother Principle." **If I can't imagine my grandmother understanding a concept, then I haven't made it simple enough.**

Here are three examples of what I'm talking about. As you read the following sections, try to evaluate them according to the Grandmother Principle. If you can suggest any improvements, I'd love to hear about them!

How to Teach M Syntax

Question: What's the real difference between the following two lines of M code? (Notice that the second line has *two* spaces between the **I** and the **X**.)

```
I X W X K X Q
I  X W X K X Q
```

Answer: Although the two lines look nearly identical to the novice, an experienced M guru may not even notice the similarity, at first. In reality, the two lines have very little in common, as the following re-statements make clear.

```
if X write X kill X quit
if  xecute W xecute K xecute Q
```

As you may know, M is extremely touchy about the number of spaces; one extra space can change everything that comes after. (Exactly one space separates a command from its argument. If there are two spaces, the next thing is another command.) After years of teaching M Programming, I can assure you that for many people *this concept takes a lot of getting used to*.

Teaching Solution: Famous Harvard psychologist Jerome Bruner has always said that any concept can be taught to any audience (even small children) in some intellectually honest way, as long as one can find the right way in which to present it.

This *oversimplified* diagram (see figure 1)makes it easy to tell the difference between commands and arguments in the preceding examples. One space separates the command from its argument, while two spaces separate one command from the next. Using this picture, it's easy to understand the syntax of simple, one-argument examples that don't contain labels, blocks, post-conditionals, or comments. (Those are separate concepts.)

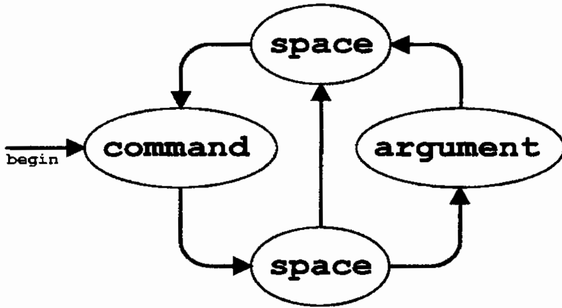

Fig. 1

This diagram, which represents one simple topic, adheres to the Grandmother Principle. It is my belief that no more than 200 such topics must be mastered in order to become an advanced M programming professional. (Most programmers know those 200 ideas intuitively, without utilizing such concrete diagrams.) But expertise has always been mysterious: no one has managed to identify all of these underlying concepts, although in six years of teaching M I have probably isolated about half of them.

How to Teach Tree-Walking

Tree-walking (formally known as sparse array traversal) is another stumbling block for many M students. The **$ORDER** and **$QUERY** functions are a constant source of confusion for some programmers. I suspect this is partly because M's brilliant sparse array data structure is so unusual, requiring many of us to "unlearn" array concepts gleaned from other languages.

The following **SET** commands create a simple local array:

```
>S ARR(0)=27,ARR(3,5)="hello"
>S ARR(3,6)=-1,ARR("B",7)=""
>S ARR("B")="trumpet"
```

In order to understand tree-walking, there are two important ways to visualize this array. The first is a list showing the nodes that contain data and their associated values. In most M systems, such lists are produced by the **ZWRITE** (or argumentless **WRITE**) command for locals and by the **%G** (or **%g**) routine for globals.

```
>ZWRITE
ARR(0)       = 27
ARR(3,5)     = hello
ARR(3,6)     = -1
ARR("B")     = trumpet
ARR("B",7)   =
```

Fig. 2

This way of visualizing the array is helpful in understanding the **$QUERY** function, which simply returns the next entry in the list after the one you specify as its argument. Here are two simple examples, using the array node listing as a reference:

```
>W $Q(ARR(0))       ┌─ARR(0)
ARR(3,5)            └►ARR(3,5)
>W $Q(ARR(3,6))     ┌─ARR(3,6)
ARR("B")            └►ARR("B")
                      ARR("B",7)
```

Fig. 3

Of course there's much more to it: $QUERY generally requires the use of *indirection*, which is by far the most difficult M language concept to master (all M books except the ANSI Standard contain some false information about indirection.)

It is even more useful to visualize M arrays as tree structures. In the following picture of the same local array, filled circles are array nodes containing values, while open circles are undefined "virtual" nodes. This kind of diagram makes **$DATA** and **$ORDER** much easier to understand.

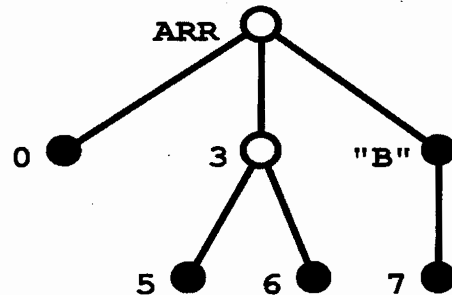

Fig. 4

Note that the number of filled-in circles is the same as the number of lines in the earlier ZWRITE listing. Using this tree, it's easy to define the $ORDER function as "jumping to the right."

$ORDER will return the next subscript to the right, under the same immediate parent, or "" if there is none.

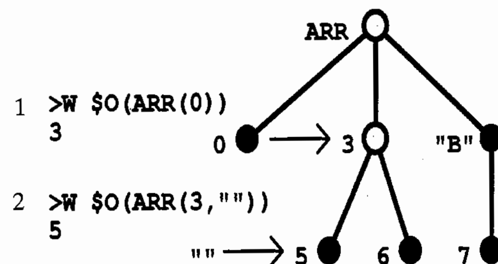

Fig. 5

Figure 5 shows two $ORDER examples, using the tree diagram as a reference.

As with $QUERY, there's still a lot more to it. For example, you still have to be able to create **FOR** loops that use $ORDER correctly in real-world global data structures. But once you know the underlying concepts, even the most sophisticated treewalking becomes understandable.

Given that these array concepts would not be introduced without some prior supporting topics, I believe that they also adhere to the Grandmother Principle.

How to Teach Null Strings

A number of functions and operators are very simple in design, *except for* special behaviors related to the null string. The null string is defined as an empty string of length **0**: it contains *no* characters. The following language elements are complicated somewhat by the null string:

$ASCII function;
$EXTRACT function;
READ command;
[(contains) operator;
] (follows) operator.

Actually, these null string behaviors are remarkably consistent in M. A common thread runs through them. The following diagram of the string "**Objects**" makes these behaviors predictable:

Characters

	""	""	"O"	"b"	"j"	"e"	"c"	"t"	"s"	""	""	
...	-1	0	1	2	3	4	5	6	8	9	10	...
	-1	-1	79	98	106	101	99	116	115	-1	-1	

ASCII Codes

Fig. 6

At the outset, it is important to explain to students that this diagram is a *behavioral model*: M will *act as if* strings looked this way, even though the system internals are different. M can't *really* store strings this way, because then each string would be infinitely long in both directions, and illegal byte values of **-1** would have to be stored in most of the locations.

The diagram has two parts; one shows the characters stored in each position of the string, the other ASCII code numbers. Character-oriented language elements (e.g. **$EXTRACT** and [) use the top part of the diagram, while ASCII code-oriented language elements (e.g. **$ASCII** and]) use the bottom part. Here are some of the behaviors explained by this way of thinking about strings:

- **$A("Objects",6)** equals **116**;
- **$A("Objects",0)** or **$A("Objects",10)** both equal **-1**;
- **$E("Objects",4)** equals **"e"**;
- **$E("Objects",-1)** or **$E("Objects",9)** both equal **""**;
- **R X:2** sets **X** to **""** if no string can be read within 2 seconds;
- **R *X:0** sets **X** to **-1** if no character can be read immediately;
- **"Objects"["bjec"** and **"Objects"["b"** are both true, but **"Objects"["B"** is false;
- **"Objects"[""** is true;
- in fact, **Y[X** is always true if **X=""**;
- **X]""** is always true unless **X** equals **""**;
- **"ABC"]"AB"** is true because the third character of **"AB"** has an ASCII code of **-1**.

Fig. 7

This is a general concept that applies under a variety of circumstances. The string diagram is presented early in the student's M learning experience; several different strings are then diagrammed by the student. The instructor then refers to the diagram whenever a null string behavior requires an explanation. Thus, the Grandmother Principle is not violated.

Summary

Each teaching model discussed above has been effective in practice and was developed after years of using a more complicated approach. There are many different kinds of good teachers and teaching methods—every teacher has unique gifts, and imposing any "best" method is only stifling in the long run. The following principles represent the most difficult philosophy of teaching that I can envision and the most effective I have found to date.

1. Slower is Faster

The first topics you cover should pave the way for those to come. The more time you spend on these initial concepts, and the more thoroughly they are covered, the easier the more advanced ideas will be to grasp, and the more the students will learn. Be patient, and sell the students on being patient themselves.

2. It's Not the Student's Fault

If a student has difficulty grasping a concept, it is not because they are to blame; it's because your teaching method needs to be refined or overhauled. The students with the most to

teach you are those who have trouble learning from you. Approach each student on their own terms, as though your personal mission in life were to make them understand. They will learn more and will greatly appreciate your level of commitment.

3. This is Not the Way

No matter how cleverly you present the concepts you are covering, know that it is not the best way to do it. Some day, if you keep this in mind, you will stumble onto a better way and you'll exclaim, "Aha! Why didn't I think of that before?"

4. Repetition is Key

Once you have an elegantly simple, impossible-to-misunderstand way of thinking about a concept, drive it home with a series of equally simple examples, illustrating every important nuance in detail. (Space limitations have prevented me from doing so in this article.) Don't assume the students understand, make sure they do. As with foreign languages, each concept must be *overlearned* in order to become second-nature to the student. Use unusual, occasionally humorous examples to keep it interesting throughout.

5. Insist on Active Students

Classroom discussion should be a dialogue, not a lecture. Encourage participation, and praise students for answering correctly or asking incisive questions. Create an atmosphere of emotional warmth, and never invalidate a student. Stress that "there's no such thing as a stupid question," and systematically brainwash yourself into truly believing that statement. Give frequent hands-on exercises, so the students can apply each important concept while they are learning it. Stress those concepts they will apply in practice; interesting nuances or theoretical concepts can be picked up later.

6. Be Self-Secure

Think on your feet. Work "without a wire," shunning detailed notes and improvising spontaneous examples whenever possible to keep things fresh. Base your approach on the specific needs of each class. Use humor whenever you make mistakes. You're not expected to be perfect.

Don't feel threatened by your students. The best way to achieve this is to know your topic thoroughly. Read M routines like newspaper pages. And read one page from the ANSI Standard every day: it's not as hard as it looks at first, and it contains much knowledge not available elsewhere. (But DON'T make your students read it!)

7. Never Stop Raising the Bar

Even a teaching approach that gets the job done and receives extremely high marks from students is never "good enough." Students are good at knowing what doesn't work for them, but they aren't experts in the field. Sometimes a great teacher can get high marks for an ineffective approach, but working harder ultimately pays big dividends for teacher and student alike. And besides, teaching the same topics over and over can only be kept interesting by trying new things and setting new challenges. **M**

Erik Zoltán has been teaching M programming courses for over 6 years at ESI Technology Corp., 5 Commonwealth Road, Natick, MA.

PROFESSIONAL OPPORTUNITIES

Kaiser Permanente Mid-Atlantic States Region, part of the nation's largest, prepaid health care system, is seeking M[UMPS] professionals to join its expanding I.T.S. department in Silver Spring, MD. Several exciting application programming positions are currently available:

M[UMPS] PROGRAMMERS
with 2-3 years' experience

Our M[UMPS] programmers enjoy current technology, competitive pay, modern facilities and easy commuting access from the I-95 Calverton Exit. Join our growing team of dedicated and highly skilled information technology professionals!

We offer an excellent compensation package that includes retirement, health, dental and life insurance. For immediate consideration, please fax your resume to: (301) 816-7425 or mail to:

Kaiser Permanente
Box 6500
Rockville, MD 20849

24 hour JOBLINE 1-800-326-4005.
Visit our website! http://www.kaiseronline.org

EEO/AA

KAISER PERMANENTE