

Re: EE Student, Edit, Proposal Masters, Help (concepts of functional programming, symbolic programming and MATLAB)

Source: <http://sci.tech-archive.net/Archive/sci.math.symbolic/2005-03/0098.html>

From: Joachim Durchholz (jo_at_durchholz.org)

Date: 03/09/05

Date: Wed, 09 Mar 2005 14:29:37 +0100

John Creighton wrote:

> *John Creighton <JohnCreighton_@hotmail.com> wrote:*

>

>> *Expressions describe how data is evaluated mathematically in terms of*

>> *data (e.g. variables) and functions (e.g plus). Expressions can be*

>> *represented with strings but are usually represented as a list.*

>

> *"No, expressions are not lists. A list is a particular type of*

> *expression."*

>

> *Do you have a reference for this? I wonder if the definition of a list*

> *is agreed upon in the context of computer science.*

It is, though in some contexts it's used in a more specific way than in others.

In general, a list is a container of elements of some common type that can be iterated linearly and that has an unspecified number of elements. (Arrays are the same but have a specific number of elements.) Also, accessing the Nth element of a list is an O(N) operation.

> *In MATLAB a list*

> *separates function input arguments when calling a function and*

> *function output arguments when the function returns its values. A list*

> *has the property that each element is of indeterminate size.*

This is a specialty of Matlab lists. Or, rather, of dynamically-typed languages.

> *A cell*

> *array in MATLAB is also a data structure which maps an n tuple of*

> *integers to elements of indeterminate size. The natural way to*

> *represent such a data structure is with an array of pointers and when*

> *the array is one dimensional it has the same properties of a list.*

Agreed, but that's not specific to lists but a property of data structures with elements of varying size. I.e. this is done not only in lists and arrays but in records as well.

- > *Perhaps it would be best to see how list is defined in the language*
- > *lisp given lisp was the second programming language and stands for*
- > *list processing.*

Lisp uses a definition that explicitly relies on implementation details, so it's a bit misleading.

And, no, Lisp isn't the last word on what's a list and what isn't, even if it's the oldest modern language.

- > *By the very sound of its name I would of took the word expression to*
- > *be something which can be evaluated.*

You have been misled. A list is simply a data structure.

Interpreters often use lists to store expressions that are awaiting evaluation, but that's just an implementation detail. In practice, it's usually an "association list", a mapping from names to expressions, which is often implemented as a list in Lisp but as some tree structure in any other language (Lisp as defined in the '50ies isn't particularly efficient at working with datastructures other than lists, mostly because everything is forced into double-pointer list cells even if it doesn't make sense. Modern Lisp dialects tend to have data structures beyond lists and can do reasonably well, though they still tend to call the thing an "association list" even if it isn't a list. It's a traditional name *g*)

- >>*The*
- >>*MATLAB equivalent of a list is a one dimensional cell array. In an*
- >>*expression, the first element of the list is the function or*
- >>*operation. The subsequent elements are the arguments. Arguments can*
- >>*be*
- >>*any data structure including expressions. When the arguments of the*
- >>*functions in an expression are expressions it becomes natural to*
- >>*represent the expression as a tree. This representation is known as*
- >>*the operation tree. Unlike the flow of procedural programming*
- >>*languages, in functional programming language the outer function is*
- >>*evaluated first and the inner function is only evaluated if called by*
- >>*the outer function. This is referred to as lazy evaluation.*
- >
- >
- > *"Maple does not use lazy evaluation. When an expression is evaluated,*
- > *it is ordinarily evaluated from the inside out: by default, a function*
- > *evaluates its arguments first. There are some exceptions."*
- >
- > *I do know that you can explore the operation tree with the functions:*
- > *op*

- > *type*
- > *whattype*
- >
- > *It would seem to me that since expressions are stored in an operation*
- > *tree that to get to the innermost argument you must start from the*
- > *outside and work inward.*

That's what lazy evaluation does, at least in theory, but making this even reasonably efficient requires a good deal of coding.

Most tree evaluation algorithms work recursively, along the lines of

```
eval (tree-node) is
  case tree-node.operation of
  add:
    eval (tree-node.child [1]) + eval (tree-node.child [2])
  negate:
    - eval (tree-node.child [1])
  if-then-else:
    if eval (tree-node.child [1]) then
      eval (tree-node.child [2])
    else
      eval (tree-node.child [3])
    end
  ... other operations go here
  end
end
```

This is easy to code and reasonably efficient, and that's why most language use strict evaluation.

Note that only one of the "then" and "else" part of an if-then-else expression is evaluated. If both parts were evaluated, it would be impossible to write a recursive function that terminates.

- > *Perhaps expressions are by default*
- > *automatically evaluated by maple before being passed to a function.*
- > *And perhaps there are ways to prevent this. Given a simply command*
- > *like collect or expand it would seem like a waste of computer cycles*
- > *to reevaluate the expression before the function which simplifies the*
- > *expression is carried out.*

I don't know how Matlab does this, but usually expressions are evaluated whenever they are used. It's the responsibility of the programmer/user to assign results to intermediate variables to prevent multiple evaluation of the same expression.

Many compilers do "common subexpression elimination", but this optimization is usually limited to the scope of a single function. And it's not motivated to make programmer-written code faster (programmers tend to factor out common expressions anyway), it's to make compiler-generated code faster, particularly code that accesses arrays.

- > *I am interested to here more on this do you*
- > *have any good references? Anyway, I still think a natural way to*
- > *represent an operation tree is with an array of pointers where the*
- > *first pointer points to the operation and subsequent pointers point to*
- > *the elements.*

It's usually a list, because during parsing a function's parameter list, the parser doesn't know how many additional subexpressions there will be. Binary operators are then done in lists as well, simply for uniformity (uniformity is a programmer's friend because variation means that there are many more cases to test for all the code that uses the data structure).

Regards,
Jo