Empirical Modelling and the use of LISP

1003773

Abstract

This paper attempts to explore the advantages and disadvantages of the current code writing and macro techniques available within EDEN, and the use of LISP's macros to improve the ability to create advanced models and teach with definitive notation.

1 Introduction and motivation.

Within Empirical Modelling, we currently see prevalent use of macro-assisted coding to create models. For example, within the PjawnsMartin2003 model, we seen that much of the code is highly repetitive, such as within the board.eden script, where we have copies of *"viewport pjawns11 circle 11"*, or lines such as *"p11 = circle({500,500},400)"*, among others, which are all copied 64 times (one for each square on the pjawns board). These large blocks of near-identical code plagues vast amounts of EDEN code, and a cursory look at previous projects done in EDEN such as the majority on `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/` shows that the use of this style of scripting within EDEN is the prevalent way to create large numbers of dependencies.

This prevalent use of repeated code is essentially opposed to most ideas of what constitutes good coding practice. Whilst this may be fine in the use of small models, if scaled up to even slightly larger projects, it is easy to see how this would become unmaintainable. If a single change was required on the general form of one of these blocks, the whole section would have to be re-written, which alone is quite an irritating task, to say nothing of if we were re-factoring many different blocks around.

2 Current solutions.

At present, there are a few main ways in which this problem is/can theoretically be handled. The first of which is the use of separate code to write this section of code. As an example, the above viewport code could be written as

```
for(i = 1; i <= BOARDWIDTH; i++){
for(j = 1; j <= BOARDHEIGHT; j++){
writeln("viewport pjawns"+str(i)+str(j)+" circle "+str(i)+str(j));
}
}
```

This has the unhelpful side effect of having to be altered, copied, and pasted in to the right section whenever you wish to change the code, but is at least a step in the right direction towards maintainability.

Moving closer to re-usability, another way of writing large numbers of similar bits of code is through the *"execute"* function in EDEN. This function takes any string it is given, and attempts to run it as an EDEN script. To use it to create large sections of related EDEN definitions, we could alter the above code, and instead of outputting to the standard output, simply append to a string, which we then execute afterwards, which will then create all the dependencies we require.

This dynamic creation of dependencies is what we require, however, doing so feels as though you are fighting with the language to create your model, rather than the language assisting in its' creation.

3 Why LISP is a candidate for adding to the project.

There are a number of reasons why I suggest using a modified version of LISP as a solution to the problem. The first of these is the similarity of LISP's macros to our best "in-language" solution to the dependency deluge. LISP has always had this code-writing-code functionality built into its core, and is indeed the whole reason the language has not moved away from its' infamous parentheses since 1958.

The second of these is because LISP's functional form allows it to be easily used in an on-the-fly-interpreter. This is a key point in its role in Empirical Modelling, as it allows for a very fast feedback loop between the code-writer and the results of the code. Due to this immediacy of response, this also allows the writer to learn more quickly from the statements they input, a main objective of the Empirical Modelling project.

The final, and most important, reason I wish to alter LISP to be a definitive notation, is due to its' homoiconicity: its' ability to represent its' own code as data. This could allow us to look at at Empirical Modelling from a more general perspective: The ability to view and modify the code itself as an observable. Not only that, but due to LISP's ability to alter its' own language, the language itself could also be viewed as an observable, whilst within an application created in this language.

If we allow the code of the model itself to become an observable for the agents within the system, then we can allow for easier access for learners to interact with, and learn from, a model of software. This idea has already been touched on with the EMPE environment, in which a developer/teacher can explicitly state code which a student may then run and observe the results.

4 How LISP should be modified to fit the Empirical Modelling project.

There are a number of alterations we need to make to the standard LISP framework in order to allow dependencies to work as we would expect. The first of these is adding a special form to the language that allows us to create dependencies. This will created so that the form *(defact dependant action)* will add a pointer from every symbol in the *dependants* form to the *action* symbol, that will cause the form referenced by *action* to be evaluated whenever the value of any symbol in *dependants* changes.

The second change that is needed is to alter the *set* and/or *define* special forms, so that they re-evaluate the dependant form(s) of the variable being set.

5 What was achieved.

At the current stage of development of the interpreter (from this point onwards referred to as LIDEN (LIsp with DEfinitive Notations)), we have managed to create a working (but highly buggy), version of a LISP interpreter, with access to several basic LISP constructs and the above *defact*. The interpreter is not complete, however, as many things such as lexical scope are either not working or not correctly implemented at this current time. The current environment does allow us to get a glimpse of what may be possible in the future with this idea.

6 Examples of how LIDEN may be used to develop software.

In most programming languages, we must use the language as-is, without alteration. However, with LIDEN as our environment, we can expand the language indefinitely whilst working with it. This allows us to change the language to suit how we may view the code, rather than as the language creator viewed it. As a simple example, suppose we had some code
(if (canSee) lotsOfGoingOutsideCode),
but our own vision of what this code means we should instead like to write
(whenSunComesUp stayOutsideUntilDark),
as our experience of the code suggests that canSee is a dependant on if the sun is up, and the lotsOfGoingOutsideCode recurses often enough that we, the programmer, view this whole section of code as a loop.

In any language that does not have macro support, it is essentially impossible to allow for and adapt to this (in the view of the author, admittedly rather odd) view/experience of the program we are creating, as any user-made functions in other languages will evaluate the arguments (stayOutsideUntilDark) of a function (whenSunComesUp) before passing them, which would be incorrect in the case of a lazy-if, where the argument is not evaluated until we have branched. However, within LIDEN, such an alteration/abstraction/adaptation would be relatively easy to create. We can simply write
(defmacro whenSunComesUp (someCode) '(if (canSee) someCode)).

All of the above is possible in regular LISP, but with the addition of *defact/defdep*, we can also have lotsOfGoingOutsideCode change at run-time and just use a dependency between it and stayOutsideUntilDark. For example, if lotsOfGoingOutsideCode was a list that other functions changed based on what they feel you should do outside (i.e. *parent* functions attempting to modify your behaviour), we would be able to still use the stayOutsideUntilDark alias without manually updating it, as we would have to do with LISP.

This may seem like a silly example, but it illustrates what is possible when the code itself becomes an variable that we may observe and alter at run-time.

7 Further work and applications.

7.1 Creating a more usable interpreter.

Obviously the interpreter currently has various bugs and issues, however, once those are fixed, there are still several things I would wish to add to LIDEN, such as to easily allow for dependencies similar to EDEN's
identifier is expression
Adding EDEN's dependencies can in fact be accomplished without modifying the interpreter, thanks to LISP's power in creating new syntax. To do this, we can create a function within LISP that returns a (1-deep) list of all tokens within a list. Then, after removing the tokens which appear after *let* or *define* tokens, we can create the action
(defact (list of tokens) originalexpression).

7.2 Self-Extensionability.

I would also like to add the ability to execute EDEN statements in LIDEN. This would be relatively simple to achieve, thanks to EDEN's *execute* function. This would allow us to add new functionality (such as access to the work done in Donald and Scout), to LIDEN within the environment of the interpreter itself.

7.3 Integrating the interpreter with the tkeden environment.

The final thing I believe would improve LIDEN is to re-integrate the interpreter back into tkeden. At present, the interpreter runs in a very basic text window, created so that there was an easy way to test LIDEN. With more work, however, I feel it could be listed as another language in tkeden, sharing its observables and dependencies with the other definitive notations.