Pico Lisp
A Radical Approach to Application Development

Software Lab. Alexander Burger
Bahnhofstr. 24a, D-86462 Langweid
abu@software-lab.de (www.software-lab.de)
+49-(0)821-9907090

March 31, 2004

Abstract

Criteria for productive application development are considered (yet again), and a point is made why we regard Lisp as the *only* language suited for that task. Pico Lisp is presented as a successful example, used in commercial applications for many years, and adapted to this task (arguably) better than any other Lisp.

1 Introduction

I am working as a consultant and free software developer. During the past twenty years my partners and I worked on projects as diverse as pre-press image processing, computer aided design, simulations, and various financial and business applications.

For almost all these projects we used Lisp.

My daily job is to listen to customer requests, to analyze business processes, and develop software according to those needs.

Typically – in business applications like ERP or CRM – this is a process of permanent change. At the beginning of a new project, neither the developer nor the customer know for sure what ist needed, and how exactly the final product should look like.

It will be found by an iterative process (some call it "extreme programming"): The customer evaluates each new version, then we discuss further strategies. It is not uncommon that unanticipated requirements may cause large parts of the project to be rewritten. This does not necessarily imply bad planning, because the process I describe here *is* the planning. In an ideal world, software development is *only* planning – the time spent for actually writing code should converge towards zero.

We need a programming language which lets us directly express what we want the program to do, in a pragmatic and flexible way. And we believe that everything should be as simple as possible, so that the programmer is able to understand at any time what is going on under the hood.

Over the years, the Pico Lisp [1] system evolved from a minimalist Lisp implementation to a dedicated application server. Please note that we are not talking of a rapid prototyping tool. At each development step, the result is always a fully functional program, not a prototype, growing towards the (possibly final) production version. Instead, you may call it a power tool for the professional programmer, who likes to keep in control of his evironment, and wants to express his application logic and data structures in a concise notation.

First we want to introduce Pico Lisp, explain why Pico differs in its lower levels quite radically from other Lisps or development systems, and then show its benefit at the higher levels.

2 A Radical Approach

The (Common-) Lisp community will probably not be enthusiastic about Pico Lisp, because it disposes of several traditional Lisp beliefs and dogmas. Some are just myths, but they can cause Lisp to become too complicated, heavy and slow. The practical experience with Pico Lisp proves that a lightweight and fast Lisp is optimal for many kinds of productive application development.

2.1 Myth 1: Lisp Needs a Compiler

This is in fact the most significant myth. If you listen to Lisp discussion groups, the compiler plays a central role. You might get the impression that it is almost a synonym for the execution environment. People worry about what the compiler does to their code, and how effective it is. If your Lisp program appears to be slow, you are supposed to get a better compiler.

The idea of an *interpreted* Lisp is regarded as an old misconception. A modern Lisp needs a compiler; the interpreter is just a useful add-on, and mainly an interactive debugging aid. It is too slow and bloated for executing production level programs.

We believe that the opposite is true. For one thing (and not just from a philosophical point of view) a compiled Lisp program is no longer Lisp at all. It breaks the fundamental rule of "formal equivalence of code and data". The resulting code does not consist of S-Expressions, and cannot be handled by Lisp. The source language (Lisp) was transformed to another language (machine code), with inevitable incompatibilities between different virtual machines.

Practically, a compiler complicates the whole system. Features like multiple binding strategies, typed variables and macros were introduced to satisfy the needs of compilers. The system gets bloated, because it also has to support the interpreter and thus two rather different architectures.

But is it worth the effort? Sure, there is some gain in raw execution speed, and compiler construction is interesting for academic work. But we claim that in daily life a well-designed "interpreter" can often outperform a compiled system.

You understand that we are not really talking about "interpretation". A Lisp system immediately converts all input to internal pointer structures called "S-Expressions". True "interpretation" would deal with one-dimensional character codes, considerably slowing down the execution process. Lisp, however, "evaluates" the S-Expressions by quickly following these pointer structures. There are no searches or lookups involved, so nothing is really "interpreted". But out of habit we'll stick to that term.

A Lisp program as an S-Expression forms a tree of executable nodes. The code in these nodes is typically written in optimized C or assembly, so the task of the interpreter is simply to pass control from one node to the other. Because many of those built-in lisp functions are very powerful and do a lot of processing, most of the time is spent in the nodes. The tree itself functions as a kind of glue.

A Lisp compiler will remove some of that glue, and replace some nodes with primitive or flow functionality directly with machine code. But because most of the time is spent in built-in functions anyway, the improvements will not be as dramatic as for example in a Java byte code compiler, where each node (a byte code) has just a comparatively

primitive functionality.

Of course, the compilation itself is also quite time-consuming. An application server often executes single-pass Lisp source files on the fly, and immediately discards the code when it is done. In these cases, either the inherently slower interpreter of a compiler-based Lisp system, or the additional time spent by the compiler will noticeable degrade the overall performance.

Pico Lisp's internal structures were designed for convenient interpretation from the beginning. Though it is completely written in C, and was not specially optimized for speed, a lack of performance was never an issue: The first commercial production system written in Pico Lisp was an image processing, retouch, and page layout program for the printing and pre-press industry, in 1988, on a Mac II with a 12 MHz CPU and 8 MB of RAM. No Lisp compiler, of course, just the low level pixel manipulations and bezier routines were written in C. Even then, on a hardware hundreds of times slower than today's, nobody complained about the performance.

Just out of interest I installed CLisp the other day, and compared it with Pico Lisp for some simple benchmarks. Of course, the results are not meant to reflect the usefulness of either system as an application server, but they give a rough indication about the relative performances.

First I tried the simple recursive fibonacci function:

```
(defun fibo (N)
   (if (< N 2)
      1
      (+
         (fibo (- N 1))
         (fibo (- N 2)) ) ) )
```

When called as `(fibo 30)`, I get the following execution times (on a 266 MHz Pentium-I notebook):

Pico (interpreted)	12 sec
CLisp interpreted	37 sec
CLisp compiled	7 sec

The CLisp interpreter is about three times slower, the compiler roughly twice as fast as Pico Lisp.

However, the fibonacci function is not a good example of a typical Lisp program. It consists only of primitive flow and arithmetic functions, is easy for a compiler to optimize, and might be written directly in `C` if it were time-critical (in that case, it would take only 0.2 sec).

Therefore, I went to the other extreme, with a function doing extensive list processings:

```
(defun tst ()
   (mapcar
      (lambda (X)
         (cons
            (car X)
            (reverse (delete (car X) (cdr X))) ) )
      '((a b c a b c) (b c d b c d) (c d e c d e) (d e f d e f)) ) )
```

When called[1] one million times, I got:

[1] For a Pico Lisp version, replace `defun` with `de` and `lambda` with `quote`

Pico (interpreted)	31 sec
CLisp interpreted	196 sec
CLisp compiled	80 sec

Now the CLisp interpreter is more than six times slower, but to my surprise the compiled code is still 2.58 times slower than Pico Lisp.

Perhaps CLisp comes with a particularly slow Lisp compiler? And probably the code can be sped up using some tricks. But still these results leave a lot of doubt whether the overhead for a compiler can be justified. Fiddling around with compiler optimization is the last thing I want to do when I'm concerned about the application logic, and when the user anyway doesn't notice any delays.

2.2 Myth 2: Lisp Needs Plenty of Data Types

The fibonacci function in the above benchmark can probably be sped up by declaring the variable N to some integer. But this shows how much Lisp got influenced by the demands of compiler support. A compiler can produce more efficient code when data types are statically decleared. Common Lisp supports a whole zoo of types, including various integer, fixed/floating point, or rational number types, characters, strings, symbols, structs, hash tables, and vectored types in addition to lists.

Pico Lisp, on the other hand, supports only three built-in data types – numbers, symbols and lists – and can get along with them remarably well. A Lisp system works faster with fewer data types, because fewer options have to be checked at runtime. There may be some cost for less efficient memory usage, but fewer types can also save space because fewer tag bits are required.

The main reason for using only three data types is simplicity, an advantage which by far outweighs the speed and space considerations.

At the lowest level, in fact, Pico Lisp consists of only a single data type, the `cell`, which is used internally to construct numbers, symbols and lists. A small number or a minimal symbol occupies only a single cell in memory, growing dynamically on demand. This memory model also allows for efficient garbage collection and completely avoids fragmentation (as would be the case, for example, with vectors).

At the higher levels it is always possible to emulate other data types using these three primitive types. So we emulate trees by lists, and strings, classes and objects by symbols. As long as we observe no performance problems why should we make it more complicated?

2.3 Myth 3: Dynamic Binding is Bad

Pico Lisp employs a straightforward implementation of dynamic, shallow binding: The content of a symbol's value cell is saved when a lambda body or binding environment is entered, then set to its new value. Upon return, the original value is restored. As a result, the current value of a symbol is determined dynamically by the history and state of execution, not by static inspection of the lexical environment.

For an interpreted system, this is probably the simplest and fastest strategy. Looking up the value for a symbol does not require any searches (just access to the value cell), and all symbols (local or global) are treated uniformly. A compiler, on the other hand, can produce more efficient code for lexical bindings, so compiled Lisps usually complicate things by supporting several several types of binding strategies.

Dynamic binding is a very powerful mechanism. The current value of any symbol can be accessed from any place, both the symbol and its value are always the "real thing", physically existent, not something that it just "looks like" (as is the case with lexical

binding, and – to some degree in Pico Lisp – the use of transient symbols (see below)).

Unfortunately, power is not available without danger. The programmer must be familiar with the underlying principles to use their advantages and avoid their pitfalls. As long as we stick to the conventions recommended by Pico Lisp, the riscs can be minimized, however.

We can see two types of situation where the results of a computation involving dynamic binding may come out unexpected for the programmer:

- A symbol is bound to *itself* and we try to modify the symbol's value
- The *funarg* problem, when a symbol's value got dynamically modified by pass-through code that is invisible in the current source environment.

Both situations are defused when we resort to transient symbols in such cases.

Transient symbols are symbols in Pico Lisp which look like strings (and are often used as such[2]), and which are interned only temporarily during execution of a single source file (or just a part of it). Thus, they have a lexical scope, comparable to `static` identifiers in `C` programs, though their behavior is still completely dynamic, because they are normal symbols in all other respects.

So the rules are simple: Whenever a function has to modify the value of a passed-in symbol, or to evaluate (directly or indirectly) a passed-in Lisp expression, its parameters should be written as transient symbols. Actual experience shows, however, that these cases are rare in the top levels of application development, and occur mostly in the support libraries and system tools.

2.4 Myth 4: Property Lists are Bad

Properties are a nice, clean way to associate information with symbols in addition to the value/function cell. They are extremely flexible, because the amount and type of data is not statically fixed.

Most people seem to believe that property *lists* are too ancient and primitive to be used today. More advanced data structures should be used instead. While this is true in some cases, depending on the total number of properties in a symbol, the break-even point might be higher than expected.

Previous versions of Pico Lisp experimented with hash tables and self-adjusting binary trees to store properties, but we found the plain list simply to be more effective. We must take into account the net effect of the total system, and the overhead both for maintaining many internal data types (see above) and more complicated lookup algorithms is often larger than that involved with simple linear lookup. And when we are also concerned about memory efficiency, the advantages of property lisps are clearly winning.

Pico Lisp implements properties in lists of key-value pairs. The only concession to speed optimization is a last-recently-used scheme, accelerating repeated accesses a little, but we have no concrete evidency whether this was actually necessary.

Another argument against properties is their alleged global scope. This is true to the same extend as an item in a `C`-structure, or an instance variable in a Java object, is global.

A property in a *global symbol* is global, of course, but in typical application programming properties are stored in anonymous symbols, objects or database entities, all of which are accessible only in a well-defined context. Therefore, a property named `''color''` can be used with a certain meaning in one context, and with a completely different meaning in another, without any interference.

[2] perhaps an unfortunate design decision

3 The Application Server

On top of that simple Pico Lisp machine we developed a vertically structured application server. It unifies a database engine (based on Pico's implementation of persistent objects as a first class data type) and an abstracted GUI (generating, for example, HTML and generic Java Applets).

The crucial element in that unified system is a Lisp-based markup language, which is used to implement the individual application modules.

Whenever a new database view, an editor mask, a document, a report or some other service is requested from the application server, a Lisp source file is `load`ed, and executed on the fly. This is similar to a request for an URL, followed by sending a HTML file, in a traditional web server. The Lisp expressions evaluated in that course, however, usually have the side effect of building and handling an interactive user interface.

These Lisp expressions describe the layout of GUI components, their behavior in response to user actions, and their connection and interaction with database objects. In short, they contain the complete specification of that application module. To make this possible, we found it important to strictly adhere to the *Locality Principle*, and to use the mechanisms of "Prefix Classes" and "Relation Maintenance Daemons" (the latter two are described elsewhere, see [2]).

3.1 Locality Principle

As we said, business application development is a process of permanent change. The Locality Principle proved to be of great help for the maintenance of such projects. It demands that all relevant information concerning a single module should be kept together in a single place. It allows for the programmer to keep a single focus of attendance.

This sounds quite obvious, of course, but opposed to this, current software design methologies recommend to encapsulate behavior and data, and hide them from the rest of the application. This usually results in a system where the application logic is written in some place (source file), but the functions, classes and methods implementing the functionality are defined somewhere else. To be sure, in general this is a good recommendation, but it gives a lot of problems in a permanently changing environment: Contexts switches and modifications have to be done simultaneously in several places. If a feature is obsolete, some modules may become obsolete too, but we often forget to remove them.

So we think that the optimal way is to build an abstracted library of functions, classes, and methods – as general-purpose as possible, and virtually constant over time and between individual applications – and to use that to build a tailored markup language with high expressive power to actually write the applications.

That language should have a compact syntax and allow the description of all static and dynamic aspects of the application. Locally, in one single place. Without need to define behavior in separate class files.

3.2 Lisp

And this is the main reason why we said in the beginning that Lisp is the *only* language suitable for us. Only Lisp allows a uniform treatment of code and data, and this is the foundation of the Pico Lisp application programming model. It makes heavy use of functional bodies and evaluatable expressions, mixed freely with static data and passed around – and stored in – the internal runtime data structures.

To our knowledge, this is not possible with any other programming language, at least not

with similar simplicity and elegance. To a certain degree it might be done in scripting languages, using interpretable text strings, but only rather limited and clumsy. And – as described above – compiler-dependent Lisp systems might be a bit too heavy and inflexible. In order for all these data structures and code fragments to work together smoothly, Pico Lisp's dynamic shallow binding strategy is of great advantage, because expressions can be evaluated without the need of binding environment setup.

Another reason is Lisp's ability to directly manipulate complex data strucures like symbols and nested lists, without having to explicitly declare, allocate, initialize, or de-allocate them. This also contributes to the compactness and readability of the code and gives expressive power to the programmer, letting him do things in-line – at the snap of a finger – where other languages would require him to program a separate module.

Additionally, as Pico Lisp makes no formal distinction between database objects and internal symbols, all these advantages apply to database programming as well, resulting in a direct linkage of GUI and database operations in the same local scope, using identical syntax.

4 Conclusion

The Lisp community seems to suffer from a paranoia of "unefficient" Lisp. This is probably due to the fact that for decades they had to defend their language against claims like "Lisp is slow" and "Lisp is bloated".

Partly, this used to be true. But on today's hardware raw execution speed doesn't matter for many practical applications. And in those cases where it *does*, just coding a few critical functions in `C` usually solves the problem.

Now let's turn our focus to more practical aspects. Some people might be surprised how small and fast a supposedly "ancient" Lisp system can be. So we should be careful not to make Lisp really "bloated" by overloading the core language with more and more features, but dare to employ *simple* solutions which give their full flexibility to the programmer.

Pico Lisp can be seen as a proof of "Less may be More".

References

[1] Pico Lisp Download, http://www.software-lab.de/down.html

[2] A Unifying Language for Database And User Interface Development, A.Burger 2002, http://www.software-lab.de/dbui.html