(Ed.) Thorsten Jolitz

PicoLisp Works

– References, Tutorials, Articles, Essays –

Version 1.0
August 23, 2012

"Perfection is attained not
when there is nothing left to add
but when there is nothing left to take away"
(Antoine de Saint-Exupéry)

Preface

PicoLisp Works is a compilation of (almost) all available information about the technological gem *PicoLisp* - a programming language and environment that definitely deserves wider attention.

Built on the unique characteristics of Lisp (almost no syntax, code is equivalent to data), PicoLisp combines powerful abstractions with simplicity and purity.

In a software world that is driven by hypes and desillusions, a language like PicoLisp almost appears as timeless as mathematics. With its roots in the very beginning of programming language development (Lisp was, together with Fortran, among the very first of its kind), PicoLisp may well represent the future too – as a candidate for being the "hundred-year language", that all programming languages finally converge into.

As Paul Graham puts it in his famous essay [1]:

> *The hundred-year language could, in principle, be designed today, and such a language, if it existed, might be good to program in today.*

This book consists of references, tutorials, articles and essays about PicoLisp. The reader should consider all documents written by *Alexander Burger*, the creator of PicoLisp, as the "official" references for the language. While the community tutorials and articles might be very helpful and a great source of information, they are just that - documents written by members of the PicoLisp community at various stages of "PicoLisp Enlightment". When in doubt about substantive or style questions, always refer to the offcial docs as last instance.

One of the official articles, *A Unifying Language for Database And User Interface Development*, has been written as early as 2002 and is therefore slightly

[1] Paul Graham: "The Hundred-Year Language" `http://paulgraham.com/hundred.html`, 2003

out of date in some technical details. It describes the old *Java-Applet GUI* that is not supported anymore. However, since the conceptual reasoning in this article is still valid and of fundamental importance, it was included in this book anyway.

PicoLisp Works is accompanied by a second volume, *PicoLisp by Example*, with more than 600 PicoLisp solutions to a wide range of programming tasks as well as the full PicoLisp function reference. Both volumes are freely available as *pdf* files, e.g. on *Scribd* [2]. They are published under the *GNU Free Documentation Licence*, their source code is available in public *Github* repositories [3] [4].

Berlin, August 2012 *Thorsten Jolitz*

[2] scribd.com
[3] https://github.com/tj64/picolisp-works
[4] https://github.com/tj64/picolisp-by-example

Contents

20 A PicoLisp Tutorial

21 PicoLisp Application Development

Part VIII Ersatz PicoLisp

List of Contributors

Alexander Burger
Germany
abu@software-lab.de

José Ignacio Romero
Argentina
jir@2.71828.com.ar

Thorsten Jolitz
Germany
tjolitz@gmail.com

Henrik Sarvell
Sweden
hsarvell@gmail.com

Part I

Philosophy and Concepts of PicoLisp

1

A Radical Approach to Application Development

Alexander Burger

abu@software-lab.de

Summary. 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.

1.1 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 is needed, nor how exactly the final product should look.

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.

1.2 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.

1.2.1 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 writ-

ten 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.

1.2.2 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 declared. 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?

1.2.3 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 risks 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] perhaps an unfortunate design decision

1.2.4 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.

1.3 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]).

1.3.1 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: Context 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.

1.3.2 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.

1.4 Conclusion

The Lisp community seems to suffer from a paranoia of "inefficient" 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

2

A Unifying Language for Database And User Interface Development

Alexander Burger

abu@software-lab.de

Summary. A language framework is presented which closes the semantic gap between database, application logic, and user interface. We introduce the concepts of Prefix Classes and Relation Maintenance Daemon Objects to suggest a unified development style reducing software development time and cost.

2.1 Introduction

Ever since computers were used in commercial applications, there was a growing demand to reduce software development time and cost.

Business models are quite different from each other. Off-the-shelf software usually does not fit the individual needs for enterprise workflow data processing, real world modeling, and multimedia applications. For that reason, many companies are forced to develop their own solutions. Software is a significant cost factor.

Due to the competitive nature of business, the redundancy inherent in such individual developments can hardly be avoided. Concerning the development costs, however, there seems to be a lot of room for improvements.

Typically, these projects implement some kind of database and a set of application programs. The task of these programs is to manipulate the data in the database, implement the general application logic, and provide for some kind of user interface.

We observe a large semantic gap between the database structure, and the application logic with its user interface. The current standard database language is `SQL`, which operates on the table level, while general-purpose programming languages like `Cobol`, `C/C++` or `Java` are used to implement the application logic and user interface.

In numerous attempts to remedy the situation, object-oriented paradigms are applied to both ends, by extending the relational database to object-oriented databases, and by building user interface frameworks in object-oriented languages.

There's no doubt that object-oriented databases provide for more intuitivity and productivity, and modern graphical user interfaces (GUI) cannot not be imagined without those frameworks.

But the semantic gap does not appear to diminish significantly. Databases and user interfaces are separate worlds: Existing class libraries are concerned about visual effects and event handling, but not about application logic and database maintenance. It is the programmer's responsibility to write glue code that displays data in corresponding GUI fields, detects modifications by the user, validates them, writes changes back to the database, and does other housekeeping.

This paper introduces a language and programming environment that closes the semantic gap, by unifying database and user interface into a single application server framework.

2.2 Traditional DB and GUI Development

The mainstream database format today is still the relational model, which packs the application data into two-dimensional tables, and relies on the application program logic to correctly access these data and to maintain their integrity via proper `SQL` statements.

A single data record on the application level (like, for example, a customer or an article) - which is presented to the user within a single GUI window - is usually spread out over several tables in the database.

The application program has to `select` these data, explicitly supplying knowledge about the relations between the tables, the data types and sizes, then have them copied to variables in the application scope, and move them to the GUI window. (Note: There are tendencies to "hide" this code into the methods of an object-relational database. This serves well for better structuring the application program design, but does not decrease the actual amount of programming work. The correct `select`-statements still have to be written, and a change in the relations or table structures may require individual modifications.)

The GUI components have access to these variables in the application scope; they are notified after a successful `select` to display these data.

Now comes the most tedious part. The user interface cannot simply wait until a user has done all his changes to the data record and hits some "submit"-button. It is necessary to provide immediate feedback on field entry, field exit, and often on every key stroke or mouse click. Depending on the context and the internal state of the application, individual GUI components or program features have to be enabled or disabled.

When the focus leaves a GUI component, its data have to be validated (possibly displaying error messages to the user), certain side-effects carried out (which might influence other GUI components), and modifications written to the database. A simple change in a single text field can cause an `update` to several database tables.

All these things involve `SQL` statements which again must contain extensive knowledge about the database structure. And they cannot easily be abstracted into reusable DB- or GUI-classes, because the requirements tend to differ for each view on the data record and each combination of GUI components. Thus, they have to be written individually for each GUI window.

2.3 A Unified Approach

We use the `PicoLisp` interpreter to build a vertical, unified solution to these problems. It allows to describe a direct mapping between application structures and database objects, so that the underlying machine can handle most of the above issues automatically.

The solution is "vertical", because it extends from the virtual machine level up into the application and GUI levels. And it "unifies", in a consistent way, the `Lisp` interpreter with

- a flexible object architecture
- an object-oriented database
- relation maintenance daemon objects
- a Prolog-equivalent query language. (Note: "Unified" is also a pun here, as it is fundamental to the Prolog terminology.)
- and a user interface strategy

by using the same syntax and philosophy all along the way. We will describe its details in the following sections, and give some practical examples.

It turned out that certain requirements for the virtual machine are not met by existing languages like `C++` and `Java`. These requirements include dynamic I/O of persistent objects, and a garbage collector handling these objects accordingly.

PicoLisp was chosen as the base language, because its interpreter is simple and completely written in C. This makes it easy to incorporate the necessary extensions to the virtual machine. Besides this, two other features (of Lisp in general)

- dynamic data types and structures
- and formal equivalence of code and data

are considered essential. They are both needed for the intended descriptive syntax of the resulting application development system.

2.3.1 Object Architecture

PicoLisp employs a very simple object architecture. It uses *symbols* for the implementation of both classes and objects. There are many ways to implement symbols in Lisp [1]; in PicoLisp each symbol has a value cell, a property list, and a name.

For a symbol representing an *object*, the value cell holds a list of the object's classes, the property list holds the object's attributes (instance variables), and the name is usually empty (anonymous symbol).

For a symbol representing a *class*, the value cell holds an association list with the class' methods, concatenated with a list of the class' superclasses, the property list may hold some class attribues (class variables), and the symbol's name is the name of the class.

When a *message* (also a symbol) is sent to an object, that object's list of classes - and recursively those classes' superclasses - is searched from left to right (in a depth-first manner) for that message, and the corresponding *method* body is executed. (In effect, this is a multiple-inheritance late-binding strategy.)

In that way, a class +MyCls can be defined to inherit from three classes +Cls1, +Cls2 and +Cls3 (by convention, class names start with a '+'):

```
(class +MyCls +Cls1 +Cls2 +Cls3)
```

Then, an object can be created with

```
(new '(+MyCls))
```

or another object with equivalent behavior:

```
(new '(+Cls1 +Cls2 +Cls3))
```

In both cases the resulting objects will inherit method definitions from `+Cls1`, `+Cls2` and `+Cls3`. Because of the depth-first and left-to-right search order, however, methods in the class hierarchy of `+Cls1` will override methods with the same name anywhere in the class hierarchies of `+Cls2` and `+Cls3`.

This is a "horizontal" inheritance, as opposed to - and in addition to - the normal "vertical" inheritance. `+Cls1` and `+Cls2` can surgically alter the behavior of `+Cls3`, in a very fine-grained manner. Thus - as `+Cls3` will typically define the general behavior - `+Cls1` and `+Cls2` are called *Prefix Classes* of `+Cls3`.

This object architecture is used throughout the whole system, including the DB and GUI. And prefix classes are an essential part of it: The expressive power of `Lisp`'s equivalence of code and data is augmented in combination with prefix classes.

2.3.2 Database

The `PicoLisp` database is built upon that object architecture.

On the lowest level, a database is a collection of persistent objects. `PicoLisp` supports persistent symbols, called *external* symbols, as a first-class data type. These symbols are known to - and handled in special ways by - the interpreter.

External symbols are stored in a file, in linked blocks of fixed size, where each symbols's starting block address is computable from the symbols's name.

A read or write access to an external symbol's value or properties causes that symbol to be automatically fetched from the database file. At the end of any transaction, modified symbols are written back (`commit`) or reverted (`rollback`). The garbage collector knows about the state of external symbols, and purges currently unused symbols from memory (Note: These symbols are only temporarily removed from memory, not from the database. The latter is done separately by a DB-level garbage collector.).

On the higher levels, these external symbols are organized into class hierarchies, reflecting the application's organizational structures.

As opposed to simple two-dimensional tables, they form arbitrary data structures like lists, stacks, trees and graphs.

The connections (relations) between objects in the database are not established by index lookup, but by explicit inclusion. That is, an object referring to another object can explicitly hold (i.e. contain a pointer to) that object, and can get access to it very rapidly. There is no explicit `select` operation, everything is simply available when it is needed.

2.3.3 Relation Daemons

Generally, instances of persistent database objects are called "Entities". In our system, there is an additional separate class hierarchy of "Relations": Instances of these classes we call "relation maintenance daemon objects".

Relation daemons are a kind of *metadata*, controlling the interactions between entities, and maintaining database integrity. They are the concrete realization of an abstract Entity/Relationship.

Like other classes, relation classes can be extended and refined, and in combination with proper prefix classes a fine-grained description of the application's structure can be produced.

Besides some primitive relation classes, like `+Number`, `+String` or `+Date`, there are

- relations between entities, like `+Link` (uni-directional link), `+Joint` (bi-directional link) or `+Hook` (object-local index root)
- relations that bundle other relations into a single unit (`+Bag`)
- a `+List` prefix class
- prefix classes that maintain index trees, like `+Key` (unique index), `+Ref` (non-unique index) or `+Idx` (full text index)
- prefix classes which in turn modify index class behavior, like `+Sn` (soundex algorithm for tolerant searches) [2]
- a `+Need` prefix class, for existence checks
- a `+Dep` prefix class controlling dependencies between other relations

A defining function `rel` is provided, which is used in the context of an entity class to assign relation daemon objects to that class:

```
(rel attr (+Cls ..) Arg ..)
```

`rel` is called with an attribute name `attr`, a list of relation classes `(+Cls ..)` and - depending on these classes - other optional arguments. Basically, this function simply does a

```
(new '(+Cls ..) Arg ..)
```

and assigns the resulting daemon object to the `attr`-property of the current entity class.

Relation Prefix Usage

For example, a simple entity "Person" can be defined, having just a "name" attribute:

```
(class +Person +Entity) # class '+Person'
(rel name (+String))    # relation 'name'
```

If the `name` relation needs a unique index, it is written as:

```
(rel name (+Key +String))
```

And - for an extended example of prefix classes - if `name` should be a mandatory list of names, each with an index using the soundex algorithm:

```
(rel name (+Need +List +Sn +Idx +String))
```

This demonstrates the power of combined prefix classes, which allow to define complex object behavior "on the fly", without the need to leave the current programming focus. This is even more important in user interface programming (see section 2.3.5 on the following page).

Entity Linkage

A uni-directional link to another entity might be, for example, the person's address:

```
(rel adr (+Link) (+Address))
```

This assumes that some entity class `+Address` is defined:

```
(class +Address +Entity)
```

In reality, the relation will probably be bi-directional, with several persons living at some address:

```
(class +Person +Entity)
(rel adr (+Joint) prs (+Address))

(class +Address +Entity)
(rel prs (+List +Joint) adr (+Person))
```

This says: The `adr`-attribute of `+Person` (a `+Joint`) is an address (connected to the `prs`-attribute of `+Address`), while the `prs`-attribute of `+Address` (a `+List +Joint`) is a list of persons (connected to the `adr`-attribute of each person).

So, when a person's `adr` is assigned to some address, the `+Joint` relation daemon will take care of updating the list of persons in that address, and vice versa.

2.3.4 Query Language

For extensive searches in the database, as they are needed for reports or user-specified queries, a `Prolog` engine was incorporated into `PicoLisp`. `Prolog` is similar to `SQL`, due to its declarative nature, but much more powerful because of its rule-deriving and backtracking capabilities.

`Prolog` is easy to implement in `Lisp` [3]. We extended the basic inference machine to iterate also through facts in the database, and the basic search/backtracking algorithm to a self-optimizing parallel search through multiple index trees.

The details of the query language are beyond the scope of this paper. It is mentioned here because our production system would be incomplete without it.

2.3.5 GUI Integration

The connection between database objects and GUI components is established with only two prefix classes: `+E/R` and `+Obj`.

Normally, GUI components are created at runtime with the `gui` function, e.g.

```
(gui '(+TextField) "Text" 8)
(gui '(+NumField) "Number" 8)
```

for a text field and a numeric field, each 8 characters wide.

`+E/R` (Entity/Relation)

The `+E/R` prefix connects the GUI field with a given relation of an entity:

```
(gui '(+E/R +NumField) '(n . Obj) "Number" 8)
```

The specification `(n . Obj)` is passed as an additional argument. It indicates that this numeric field is "connected" to the `n`-property of the database object `Obj`.

Nothing else has to be done by the programmer. The field will automatically display the value of `n` from the database, and modifications entered by the user into this field will be written automatically to the database value of `n`, in the object `Obj`.

`+Obj` (Object)

The `+Obj` prefix extends the GUI field types, from primitives like numbers or strings, to database objects.

That is, a GUI field can "contain" a database object, just like a text field contains a string, and a numeric field contains a number. An object can be "set" into (assigned to) the field, and retrieving the value of the field will result in that object.

The field cannot, of course, directly display the complete database object. But it can show a typical attribute of that object, e.g. the object's name in a text field, or the object's ID number in a numeric field:

```
(gui '(+Obj +NumField) '(id +Cls) "ID" 8)
```

The `+Obj` prefix extends the capabilities of the basic field type (here a `+NumField`), in such a way that

- a proper attribute value is displayed when an object is assigned to the field
- the field is set to the corresponding object when a legal attribute value is entered
- the field performs keyboard auto-expansion to legal attribute values directly out of the database
- the field can display a choice list for matching attribute values
- the fields opens an editor for that object when it is double-clicked (HyperLink)

As a consequence, when `+E/R` and `+Obj` are combined

```
(gui '(+E/R +Obj +NumField) '(x . Obj) '(id +Cls) "Number" 8)
```

they effectively establish the GUI for an entity linkage (`+Link`, `+Joint` etc.), as described in section 2.3.3 on page 19.

2.4 An Example

Putting it all together, the total effect of the described concepts can best be explained with the help of a detailed and complete example.

Assuming a simple family database, we represent a network of family relationships. For each person, we have attribues for name, sex, date of birth, and we want to have links to father, mother, husband/wife and children.

For a better understanding, we first present the traditional, relational representation (Note that we have to introduce a unique ID number for each record. Also, in such a tabular representation, it is more convenient to store the ID of father and mother (instead of the children).):

ID	nm	sex	dat	mate	pa	ma
1	John	M	22jan1954	2		
2	Mary	F	01feb1958	1		
3	Thomas	M	26jan1988		1	2
4	Claudia	F	15jul1989		1	2
5	Michael	M	03jan1992		1	2

When viewing the these family members as a graph, we get the following object structure:

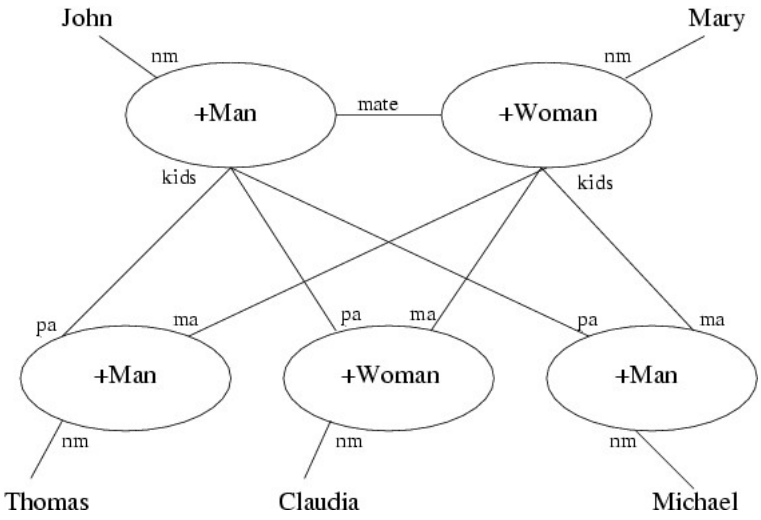


Fig. 2.1. Family Members

In the figure, we omitted the date of birth attribute.

The bi-directional relations, connecting the parents to the children and to each other, lend themselves to the `+Joint` entity linkage, resulting in the following definition for a person:

```
(class +Person +Entity)
(rel nm   (+Key +String))          # Name
(rel pa   (+Joint) kids (+Man))    # Father
(rel ma   (+Joint) kids (+Woman))  # Mother
(rel mate (+Joint) mate (+Person)) # Partner
(rel dat  (+Date))                 # born
```

From this base class, we can derive two classes `+Man` and `+Woman`:

```
(class +Man +Person)
(rel kids (+List +Joint) pa (+Person))

(class +Woman +Person)
(rel kids (+List +Joint) ma (+Person))
```

To produce a corresponding GUI

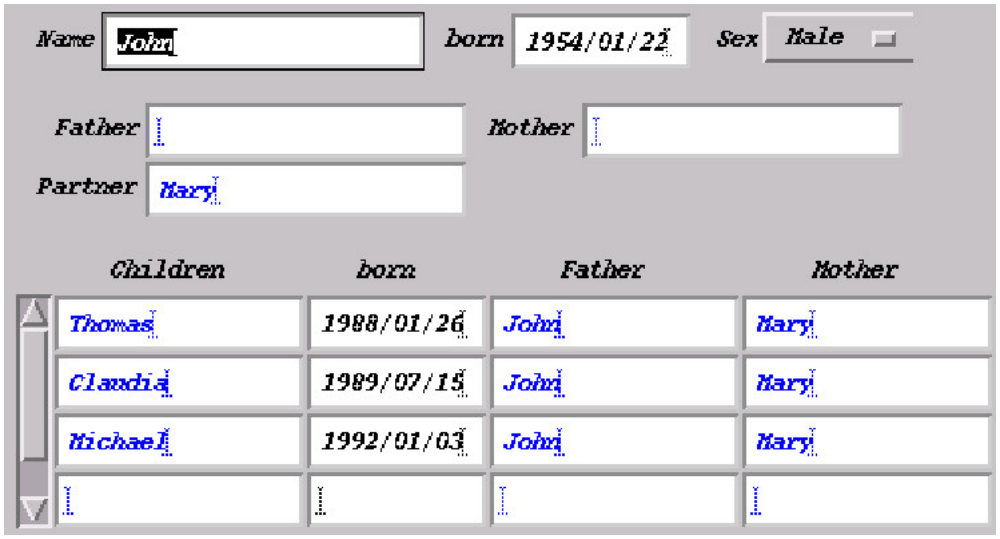


Fig. 2.2. GUI

allowing to view and edit the family members in the database, the following code is sufficient:

```
(row
   (gui '(+E/R +TextField) '(nm : home obj) "Name" 20)
   (gui '(+E/R +DateField) '(dat : home obj) "born" 10)
   (gui '(+ClassField)
      '(: home obj) "Sex"
      '(("Male" +Man) ("Female" +Woman)) ) )
(----)
(row
   (gui '(+E/R +Obj +TextField)
      '(pa : home obj) '(nm +Man)
      "Father" 20 )
   (gui '(+E/R +Obj +TextField)
      '(ma : home obj) '(nm +Woman)
      "Mother" 20 ) )
(gui '(+E/R +Obj +TextField)
   '(mate : home obj) '(nm +Person)
   "Partner" 20 )
(---- T)
```

```
(gui '(+E/R +Chart)
   '(kids : home obj)
   4 '("Children" "born" "Father" "Mother")
   (quote
      (gui '(+Obj +TextField) '(nm +Person) "" 15)
      (gui '(+Skip +Lock +DateField) "" 10)
      (gui '(+ObjView +TextField) '(: nm) "" 15)
      (gui '(+ObjView +TextField) '(: nm) "" 15) ) )
```

The above block of code will produce exactly the layout and functionality of the example GUI display. Without explaining all details here, suffice it to say that the `row` function arranges the components horizontally (while otherwise the default is vertically), `(----)` groups components into separate panels, and a `+Chart` creates an array of its argument components.

The point is that this is the *complete program*, not just some important details. It specifies the whole database and GUI application.

2.5 Discussion

The previous sections and the example show that application programming does not need to involve any concerns about database access (select, insert, update etc.) and database integrity maintenance.

The advantages are derived from the use of prefix classes and relation daemons. They allow to specify the complete program behavior and appearance in a single place of definition, and in a very concise form. Typically, the names of prefix classes are simply chained together, and intermix freely with `Lisp`'s formal indifference of code and data.

This removes the need of maintaining separate resource files, class and data declarations, and program code.

2.6 Conclusion

Achieving low program development costs is an old claim. It has been stated for manyfold methodologies and paradigms.

The system described in this paper has proven its practical value in commercial applications during several years. Among them are sales, accounting, report, and logistics applications. Research projects include graphics/animation and speech synthesis systems.

This shows that it is possible to employ the concepts of prefix classes and relation maintenance daemons successfully to commercial application development.

We observed a significant decrease in program development time. What used to be the major part of a project's development effort showed to reduce to about 5 percent.

2.7 Download

The `PicoLisp` system can be downloaded from the PicoLisp Download Page [4].

References

1. John Allen: "Anatomy of Lisp", McGraw-Hill, 1978
2. Donald E. Knuth: "The Art of Computer Programming", Vol.3, Addison-Vesley, 1973, p. 392
3. J. A. Campbell: Implementations of Prolog, Ellis Horwood Limited, 1984
4. Pico Lisp Download, http://www.software-lab.de/down.html

Part II

PicoLisp References

3

The PicoLisp Reference

Alexander Burger

abu@software-lab.de

Summary. This document describes the concepts, data types, and kernel functions of the PicoLisp system.

This is *not* a Lisp tutorial. For an introduction to Lisp, a traditional Lisp book (like [1]) is recommended. Note, however, that there are significant differences between PicoLisp and Maclisp (and even greater differences to Common Lisp).

Please take a look at the *PicoLisp Tutorial* for an explanation of some aspects of PicoLisp, and scan through the list of *Frequently Asked Questions (FAQ)*.

3.1 Introduction

PicoLisp is the result of a language design study, trying to answer the question "What is a minimal but useful architecture for a virtual machine?". Because opinions differ about what is meant by "minimal" and "useful", there are many answers to that question, and people might consider other solutions more "minimal" or more "useful". But from a practical point of view, PicoLisp has proven to be a valuable answer to that question.

First of all, PicoLisp is a virtual machine architecture, and then a programming language. It was designed in a "bottom up" way, and "bottom up" is also the most natural way to understand and to use it: *Form Follows Function.*

PicoLisp has been used in several commercial and research programming projects since 1988. Its internal structures are simple enough, allowing an experienced programmer always to fully understand what's going on under the hood, and its language features, efficiency and extensibility make it suitable for almost any practical programming task.

In a nutshell, emphasis was put on four design objectives. The PicoLisp system should be

Simple The internal data structure should be as simple as possible. Only one single data structure is used to build all higher level constructs.

Unlimited There are no limits imposed upon the language due to limitations of the virtual machine architecture. That is, there is no upper bound in symbol name length, number digit counts, stack depth, or data structure and buffer sizes, except for the total memory size of the host machine.

Dynamic Behavinor should be as dynamic as possible ("run"-time vs. "compile"-time). All decisions are delayed until runtime where possible. This involves matters like memory management, dynamic symbol binding, and late method binding.

Practical PicoLisp is not just a toy of theoretical value. It is in use since 1988 in actual application development, research and production.

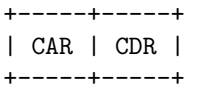
3.2 The PicoLisp Machine

An important point in the PicoLisp philosophy is the knowledge about the architecture and data structures of the internal machinery. The high-level constructs of the programming language directly map to that machinery, making the whole system both understandable and predictable.

This is similar to assembly language programming, where the programmer has complete control over the machine.

3.2.1 The Cell

The PicoLisp virtual machine is both simpler and more powerful than most current (hardware) processors. At the lowest level, it is constructed from a single data structure called "cell":



A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology. These words can represent either a numeric value (scalar) or the address of another cell (pointer). All higher level data structures are built out of cells.

The type information of higher level data is contained in the pointers to these data. Assuming the implementation on a byte-addressed physical machine, and a pointer size of typically 4 bytes, each cell has a size of 8 bytes.

Therefore, the pointer to a cell must point to an 8-byte boundary, and its bit-representation will look like:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx000
```

(the '`x`' means "don't care"). For the individual data types, the pointer is adjusted to point to other parts of a cell, in effect setting some of the lower three bits to non-zero values. These bits are then used by the interpreter to determine the data type.

In any case, bit(0) - the least significant of these bits - is reserved as a mark bit for garbage collection.

Initially, all cells in the memory are unused (free), and linked together to form a "free list". To create higher level data types at runtime, cells are taken from that free list, and returned by the garbage collector when they are no longer needed. All memory management is done via that free list; there are no additional buffers, string spaces or special memory areas, with two exceptions:

- A certain fixed area of memory is set aside to contain the executable code and global variables of the interpreter itself, and
- a standard push down stack for return addresses and temporary storage. Both are not directly accessible by the programmer).

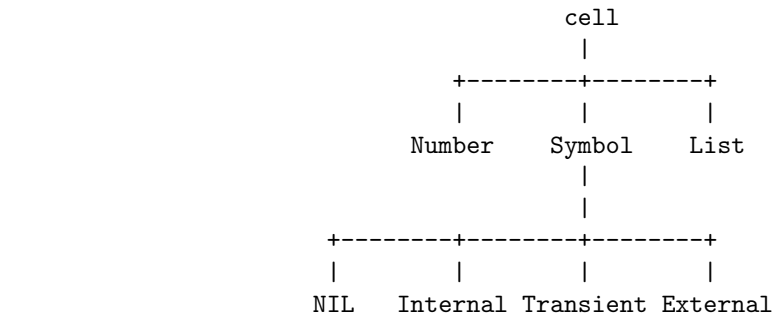
3.2.2 Data Types

On the virtual machine level, PicoLisp supports

- three base data types: Numbers, Symbols and Cons Pairs (Lists),
- the three scope variations of symbols: Internal, Transient and External, and
- the special symbol `NIL`.

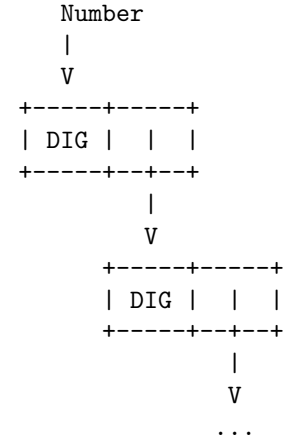
They are all built from the single cell data structure, and all runtime data cannot consist of any other types than these three.

The following diagram shows the complete data type hierarchy, consisting of the three base types and the symbol variations:



Numbers

A number can represent a signed integral value of arbitrary size. The CARs of one or more cells hold the number's "digits" (each in the machine's word size), to store the number's binary representation.



The first cell holds the least significant digit. The least significant bit of that digit represents the sign.

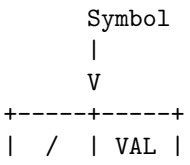
The pointer to a number points into the middle of the CAR, with an offset of 2 from the cell's start address. Therefore, the bit pattern of a number will be:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx010
```

Thus, a number is recognized by the interpreter when bit(1) is non-zero.

Symbols

A symbol is more complex than a number. Each symbol has a value, and optionally a name and an arbitrary number of properties. The CDR of a symbol cell is also called VAL, and the CAR points to the symbol's tail. As a minimum, a symbol consists of a single cell, and has no name or properties:



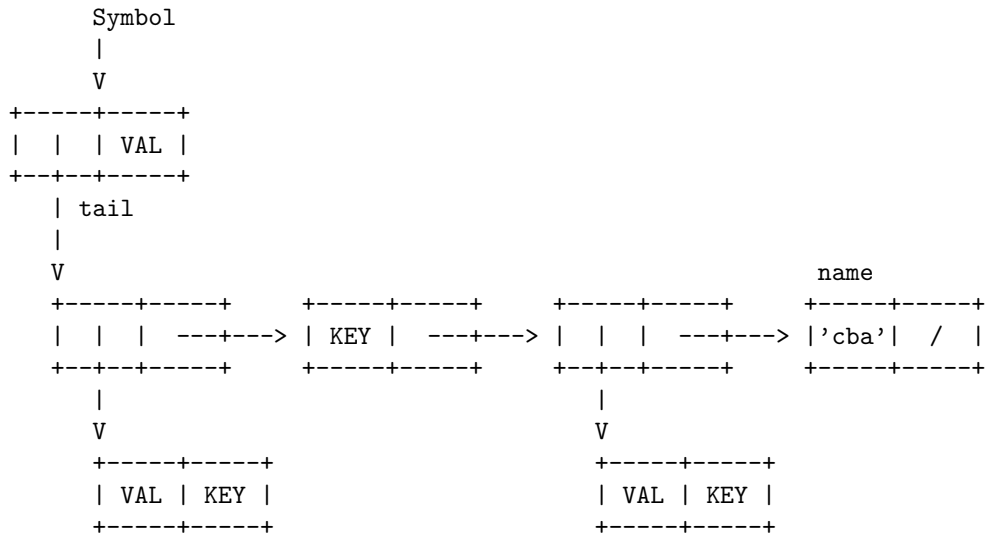
That is, the symbol's tail is empty (points to `NIL`, as indicated by the `/` character).

The pointer to a symbol points to the CDR of the cell, with an offset of 4 from the cell's start address. Therefore, the bit pattern of a symbol will be:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx100
```

Thus, a symbol is recognized by the interpreter when bit(2) is non-zero.

A property is a key-value pair, represented as a cell in the symbol's tail. This is called a "property list". The property list may be terminated by a number representing the symbol's name. In the following example, a symbol with the name `''abc''` has three properties: A KEY/VAL cell, a cell with only a KEY, and another KEY/VAL cell.



Each property in a symbol's tail is either a symbol (like the single KEY above, then it represents the boolean value `T`), or a cell with the property key in its CDR and the property value in its CAR. In both cases, the key should be a symbol, because searches in the property list are performed using pointer comparisons.

The name of a symbol is stored as a number at the end of the tail. It contains the characters of the name in UTF–8 encoding, using between one and three 8-bit-bytes per character. The first byte of the first character is stored in the lowest 8 bits of the number.

All symbols have the above structure, but depending on scope and accessibility there are actually four types of symbols: `NIL`, *internal*, *transient* and *external* symbols.

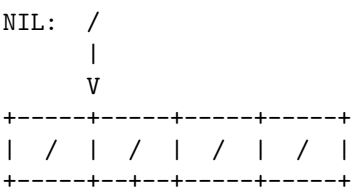
NIL

`NIL` is a special symbol which exists exactly once in the whole system. It is used

- as an end-of-list marker
- to represent the empty list
- to represent the boolean value "false"
- to represent the absolute minimum
- to represent a string of length zero

- to represent the value "Not a Number"
- as the root of all class hierarchies

For that, `NIL` has a special structure:



The reason for that structure is `NIL`'s dual nature both as a symbol and as a list:

- As a symbol, it should give `NIL` for its VAL, and be without properties
- For the empty list, `NIL` should give `NIL` both for its CAR and for its CDR

These requirements are fulfilled by the above structure.

Internal Symbols

Internal Symbols are all those "normal" symbols, as they are used for function definitions and variable names. They are "interned" into an index structure, so that it is possible to find an internal symbol by searching for its name.

There cannot be two different internal symbols with the same name.

Initially, a new internal symbol's VAL is `NIL`.

Transient Symbols

Transient symbols are only interned into a index structure for a certain time (e.g. while reading the current source file), and are released after that. That means, a transient symbol cannot be accessed then by its name, and there may be several transient symbols in the system having the same name.

Transient symbols are used

- as text strings
- as identifiers with a limited access scope (like, for example, `static` identifiers in the C language family)
- as anonymous, dynamically created objects (without a name)

Initially, a new transient symbol's VAL is that symbol itself.

A transient symbol without a name can be created with the `box` or `new` functions.

External Symbols

External symbols reside in a database file (or a similar resources, see `*Ext`), and are loaded into memory - and written back to the file - dynamically as needed, and transparently to the programmer. They are kept in memory ("cached") as long as they are accessible ("referred to") from other parts of the program, or when they were modified but not yet written to the database file (by `commit`).

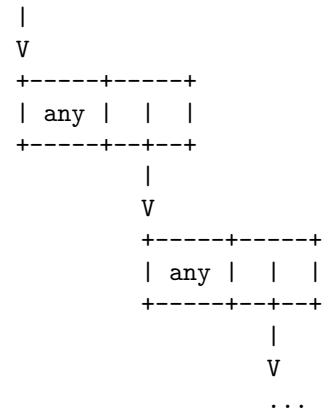
The interpreter recognizes external symbols internally by an additional tag bit in the tail structure.

There cannot be two different external symbols with the same name. External symbols are maintained in index structures while they are loaded into memory, and have their external location (disk file and block offset) directly coded into their names (more details *here*).

Initially, a new external symbol's VAL is `NIL`, unless otherwise specified at creation time.

Lists

A list is a sequence of one or more cells, holding numbers, symbols, or lists.



Lists are used in PicoLisp to emulate composite data structures like arrays, trees, stacks or queues.

In contrast to lists, numbers and symbols are collectively called "Atoms".

Typically, the CDR of each cell in a list points to the following cell, except for the last cell which points to `NIL`. If, however, the CDR of the last cell points to an atom, that cell is called a "dotted pair" (because of its I/O syntax with a dot '`.`' between the two values).

3.2.3 Memory Management

The PicoLisp interpreter has complete knowledge of all data in the system, due to the type information associated with every pointer. Therefore, an efficient garbage collector mechanism can easily be implemented. PicoLisp employs a simple but fast mark-and-sweep garbage collector.

As the collection process is very fast (in the order of milliseconds per megabyte), it was not necessary to develop more complicated, time-consuming and error-prone garbage collection algorithms (e.g. incremental collection). A compacting garbage collector is also not necessary, because the single cell data type cannot cause heap fragmentation.

3.3 Programming Environment

Lisp was chosen as the programming language, because of its clear and simple structure.

In some previous versions, a Forth-like syntax was also implemented on top of a similar virtual machine (Lifo). Though that language was more flexible and expressive, the traditional Lisp syntax proved easier to handle, and the virtual machine can be kept considerably simpler. PicoLisp inherits the major advantages of classical Lisp systems like

- Dynamic data types and structures
- Formal equivalence of code and data
- Functional programming style
- An interactive environment

In the following, some concepts and peculiarities of the PicoLisp language and environment are described.

3.3.1 Installation

PicoLisp supports two installation strategies: Local and Global.

Normally, if you didn't build PicoLisp yourself but installed it with your operating system's package manager, you will have a global installation. This allows system-wide access to the executable and library/documentation files.

To get a local installation, you can directly download the PicoLisp tarball, and follow the instructions in the INSTALL file.

A local installation will not interfere in any way with the world outside its directory. There is no need to touch any system locations, and you don't have to be root to install it. Many different versions - or local modifications - of PicoLisp can co-exist on a single machine.

Note that you are still free to have local installations along with a global installation, and invoke them explicitly as desired.

Most examples in the following apply to a global installation.

3.3.2 Invocation

When PicoLisp is invoked from the command line, an arbitrary number of arguments may follow the command name.

By default, each argument is the name of a file to be executed by the interpreter. If, however, the argument's first character is a hyphen '`-`', then the rest of that argument is taken as a Lisp function call (without the surrounding parentheses), and a hyphen by itself as an argument stops evaluation of the rest of the command line (it may be processed later using the `argv` and `opt` functions). This whole mechanism corresponds to calling `(load T)`.

A special case is if the last argument is a single '`+`'. This will switch on debug mode (the `*Dbg` global variable) and discard the '`+`'.

As a convention, PicoLisp source files have the extension ' `.l` '.

Note that the PicoLisp executable itself does not expect or accept any command line flags or options (except the '`+`', see above). They are reserved for application programs.

The simplest and shortest invocation of PicoLisp does nothing, and exits immediately by calling `bye`:

```
$ picolisp -bye
$
```

In interactive mode, the PicoLisp interpreter (see `load`) will also exit when `Ctrl-D` is entered:

```
$ picolisp
: $                          # Typed Ctrl-D
```

To start up the standard PicoLisp environment, several files should be loaded. The most commonly used things are in "lib.l" and in a bunch of other files, which are in turn loaded by "ext.l". Thus, a typical call would be:

```
$ picolisp lib.l ext.l
```

The recommended way, however, is to call the "pil" shell script, which includes "lib.l" and "ext.l". Given that your current project is loaded by some file "myProject.l" and your startup function is `main`, your invocation would look like:

```
$ pil myProject.l -main
```

For interactive development it is recommended to enable debugging mode, to get the vi-style command line editor, single-stepping, tracing and other debugging utilities.

```
$ pil myProject.l -main +
```

This is - in a local installation - equivalent to

```
$ ./dbg myProject.l -main
```

or

```
$ ./pil myProject.l -main +
```

In any case, the directory part of the first file name supplied (normally, the path to "lib.l" as called by 'pil' or 'dbg') is remembered internally as the *PicoLisp Home Directory*. This path is later automatically substituted for any leading ' `@` ' character in file name arguments to I/O functions (see `path`).

3.3.3 Input/Output

In Lisp, each internal data structure has a well-defined external representation in human-readable format. All kinds of data can be written to a file, and restored later to their original form by reading that file.

In normal operation, the PicoLisp interpreter continuously executes an infinite "read-eval-print loop". It reads one expression at a time, evaluates it, and prints the result to the console. Any input into the system, like data structures and function definitions, is done in a consistent way no matter whether it is entered at the console or read from a file.

Comments can be embedded in the input stream with the hash `#` character. Everything up to the end of that line will be ignored by the reader.

```
: (* 1 2 3)  # This is a comment
-> 6
```

A comment spanning several lines may be enclosed between `#{` and `}#`.

Here is the I/O syntax for the individual PicoLisp data types (numbers, symbols and lists) and for read-macros:

Numbers

A number consists of an arbitrary number of digits ('`0`' through '`9`'), optionally preceded by a sign character ('`+`' or '`-`'). Legal number input is:

```
: 7
-> 7
: -12345678901245678901234567890
-> -12345678901245678901234567890
```

Fixpoint numbers can be input by embedding a decimal point '`.`', and setting the global variable `*Scl` appropriately:

```
: *Scl
-> 0

: 123.45
-> 123
: 456.78
-> 457

: (setq *Scl 3)
-> 3
: 123.45
-> 123450
: 456.78
-> 456780
```

Thus, fixpoint input simply scales the number to an integer value corresponding to the number of digits in `*Scl`.

Formatted output of scaled fixpoint values can be done with the `format` and `round` functions:

```
: (format 1234567890 2)
-> "12345678.90"
: (format 1234567890 2 "." ",")
-> "12,345,678.90"
```

Symbols

The reader is able to recognize the individual symbol types from their syntactic form. A symbol name should - of course - not look like a legal number (see above).

In general, symbol names are case-sensitive. `car` is not the same as CAR.

NIL

Besides for standard normal form, `NIL` is also recognized as `()`, `[]` or `""`.

```
: NIL
-> NIL
: ()
-> NIL
: ""
-> NIL
```

Output will always appear as `NIL`.

Internal Symbols

Internal symbol names can consist of any printable (non-whitespace) character, except for the following meta characters:

```
"  '  (  )  ,  [  ]  `  ~ { }
```

It is possible, though, to include these special characters into symbol names by escaping them with a backslash '\'.

The dot '`.`' has a dual nature. It is a meta character when standing alone, denoting a *dotted pair*, but can otherwise be used in symbol names.

As a rule, anything not recognized by the reader as another data type will be returned as an internal symbol.

Transient Symbols

A transient symbol is anything surrounded by double quotes '`""`'. With that, it looks - and can be used - like a string constant in other languages. However, it is a real symbol, and may be assigned a value or a function definition, and properties.

Initially, a transient symbol's value is that symbol itself, so that it does not need to be quoted for evaluation:

```
: "This is a string"
-> "This is a string"
```

However, care must be taken when assigning a value to a transient symbol. This may cause unexpected behavior:

```
: (setq "This is a string" 12345)
-> 12345
: "This is a string"
-> 12345
```

The name of a transient symbol can contain any character except the null-byte. A double quote character can be escaped with a backslash '\', and a backslash itself has to be escaped with another backslash. Control characters can be written with a preceding hat '^' character.

```
: "We^Ird\\Str\"ing"
-> "We^Ird\\Str\"ing"
: (chop @)
-> ("W" "e" "^I" "r" "d" "\\" "S" "t" "r" "\"" "i" "n" "g")
```

The index for transient symbols is cleared automatically before and after `loading` a source file, or it can be reset explicitly with the `====` function. With that mechanism, it is possible to create symbolsb with a local access scope, not accessible from other parts of the program.

A special case of transient symbols are *anonymous symbols*. These are symbols without name (see `box`, `box?` or `new`). They print as a dollar sign (`$`) followed by a decimal digit string (actually their machine address).

External Symbols

External symbol names are surrounded by braces (‘`{`’ and ‘`}`’). The characters of the symbol's name itself identify the physical location of the external object. This is

- in the 32-bit version: The number of the database file, and - separated by a hyphen - the starting block in the database file. Both numbers are encoded in base–64 notation (characters ‘`0`’ through ‘`9`’, ‘`:`’, ‘`;`’, ‘`A`’ through ‘`Z`’ and ‘`a`’ through ‘`z`’).
- in the 64-bit version: The number of the database file minus 1 in “hax” notation (i.e. hexadecimal/alpha notation, where `@` is zero, ‘`A`’ is 1 and `O` is 15 (from “alpha” to “omega”)), immediately followed (without a hyphen) the starting block in octal (‘`0`’ through ‘`7`’).

In both cases, the database file (and possibly the hypen) are omitted for the first (default) file.

Lists

Lists are surrounded by parentheses (`(` and `)`).

`(A)` is a list consisting of a single cell, with the symbol `A` in its CAR, and `NIL` in its CDR.

`(A B C)` is a list consisting of three cells, with the symbols `A`, `B` and `C` respectively in their CAR, and `NIL` in the last cell's CDR.

`(A . B)` is a “dotted pair”, a list consisting of a single cell, with the symbol `A` in its CAR, and `B` in its CDR.

PicoLisp has built-in support for reading and printing simple circular lists. If the dot in a dotted-pair notation is immediately followed by a closing parenthesis, it indicates that the CDR of the last cell points back to the beginning of that list.

```
: (let L '(a b c) (conc L L))
-> (a b c .)
: (cdr '(a b c .))
-> (b c a .)
: (cddddr '(a b c .))
-> (b c a .)
```

A similar result can be achieved with the function `circ`. Such lists must be used with care, because many functions won't terminate or will crash when given such a list.

Read-Macros

Read-macros in PicoLisp are special forms that are recognized by the reader, and modify its behavior. Note that they take effect immediately while `read`ing an expression, and are not seen by the `eval` in the main loop.

The most prominent read-macro in Lisp is the single quote character `'` which expands to a call of the `quote` function. Note that the single quote character is also printed instead of the full function name.

```
: '(a b c)
-> (a b c)
: '(quote . a)
-> 'a
: (cons 'quote 'a)   # (quote . a)
-> 'a
: (list 'quote 'a)   # (quote a)
-> '(a)
```

A comma `,` will cause the reader to collect the following data item into an `idx` tree in the global variable `*Uni`, and to return a previously inserted equal item if present. This makes it possible to create a unique list of references to data which do normally not follow the rules of pointer equality. If the value of `*Uni` is `T`, the comma read macro mechanism is disabled.

A single backquote character ` will cause the reader to evaluate the following expression, and return the result.

```
: '(a `(+ 1 2 3) z)
-> (a 6 z)
```

A tilde character ~ inside a list will cause the reader to evaluate the following expression, and (destructively) splice the result into the list.

```
: '(a b c ~(list 'd 'e 'f) g h i)
-> (a b c d e f g h i)
```

When a tilde character is used to separate two symbol names (without surrounding whitespace), the first is taken as a namespace to look up the second (64-bit version only).

```
: 'libA~foo  # Look up 'foo' in namespace 'libA'
-> "foo"     # "foo" is not interned in the current namespace
```

Reading `libA~foo` is equivalent to switching the current namespace to `libA` (with `symbols`), reading the symbol `foo`, and then switching back to the original namespace.

Brackets ('`[`' and '`]`') can be used as super parentheses. A closing bracket will match the innermost opening bracket, or all currently open parentheses.

```
: '(a (b (c (d]
-> (a (b (c (d))))
: '(a (b [c (d]))
-> (a (b (c (d))))
```

Finally, reading the sequence '' will result in a new anonymous symbol with value `NIL`, equivalent to a call to `box` without arguments.

```
: '({} {} {})
-> ($134599965 $134599967 $134599969)
: (mapcar val @)
-> (NIL NIL NIL)
```

3.3.4 Evaluation

PicoLisp tries to evaluate any expression encountered in the read-eval-print loop. Basically, it does so by applying the following three rules:

- A number evaluates to itself.

- A symbol evaluates to its value (VAL).
- A list is evaluated as a function call, with the CAR as the function and the CDR the arguments to that function. These arguments are in turn evaluated according to these three rules.

```
: 1234
-> 1234        # Number evaluates to itself
: *Pid
-> 22972       # Symbol evaluates to its VAL
: (+ 1 2 3)
-> 6           # List is evaluated as a function call
```

For the third rule, however, things get a bit more involved. First - as a special case - if the CAR of the list is a number, the whole list is returned as it is:

```
: (1 2 3 4 5 6)
-> (1 2 3 4 5 6)
```

This is not really a function call but just a convenience to avoid having to quote simple data lists.

Otherwise, if the CAR is a symbol or a list, PicoLisp tries to obtain an executable function from that, by either using the symbol's value, or by evaluating the list.

What is an executable function? Or, said in another way, what can be applied to a list of arguments, to result in a function call? A legal function in PicoLisp is either a

number When a number is used as a function, it is simply taken as a pointer to executable code that will be called with the list of (unevaluated) arguments as its single parameter. It is up to that code to evaluate the arguments, or not. Some functions do not evaluate their arguments (e.g. `quote`) or evaluate only some of their arguments (e.g. `setq`).

or a

lambda expression A lambda expression is a list, whose CAR is either a symbol or a list of symbols, and whose CDR is a list of expressions. Note: In contrast to other Lisp implementations, the symbol LAMBDA itself does not exist in PicoLisp but is implied from context.

A few examples should help to understand the practical consequences of these rules. In the most common case, the CAR will be a symbol defined as a function, like the `*` in:

```
: (* 1 2 3)    # Call the function '*'
-> 6
```

Inspecting the VAL of `*` gives

```
: *            # Get the VAL of the symbol '*'
-> 67318096
```

The VAL of `*` is a number. In fact, it is the numeric representation of a C-function pointer, i.e. a pointer to executable code. This is the case for all built-in functions of PicoLisp.

Other functions in turn are written as Lisp expressions:

```
: (de foo (X Y)            # Define the function 'foo'
   (* (+ X Y) (+ X Y)) )
-> foo
: (foo 2 3)                # Call the function 'foo'
-> 25
: foo                      # Get the VAL of the symbol 'foo'
-> ((X Y) (* (+ X Y) (+ X Y)))
```

The VAL of `foo` is a list. It is the list that was assigned to `foo` with the `de` function. It would be perfectly legal to use `setq` instead of `de`:

```
: (setq foo '((X Y) (* (+ X Y) (+ X Y))))
-> ((X Y) (* (+ X Y) (+ X Y)))
: (foo 2 3)
-> 25
```

If the VAL of `foo` were another symbol, that symbol's VAL would be used instead to search for an executable function.

As we said above, if the CAR of the evaluated expression is not a symbol but a list, that list is evaluated to obtain an executable function.

```
: ((intern (pack "c" "a" "r")) (1 2 3))
-> 1
```

Here, the `intern` function returns the symbol `car` whose VAL is used then. It is also legal, though quite dangerous, to use the code-pointer directly:

```
: *
-> 67318096
: ((* 2 33659048) 1 2 3)
-> 6
: ((quote . 67318096) 1 2 3)
-> 6
: ((quote . 1234) (1 2 3))
Segmentation fault
```

When an executable function is defined in Lisp itself, we call it a *lambda expression*. A lambda expression always has a list of executable expressions as its CDR. The CAR, however, must be a either a list of symbols, or a single symbol, and it controls the evaluation of the arguments to the executable function according to the following rules:

When the CAR is a list of symbols For each of these symbols an argument is evaluated, then the symbols are bound simultaneously to the results. The body of the lambda expression is executed, then the VAL's of the symbols are restored to their original values. This is the most common case, a fixed number of arguments is passed to the function.

Otherwise, when the CAR is the symbol `@` All arguments are evaluated and the results kept internally in a list. The body of the lambda expression is executed, and the evaluated arguments can be accessed sequentially with the `args`, `next`, `arg` and `rest` functions. This allows to define functions with a variable number of evaluated arguments.

Otherwise, when the CAR is a single symbol The symbol is bound to the whole unevaluated argument list. The body of the lambda expression is executed, then the symbol is restored to its original value. This allows to define functions with unevaluated arguments. Any kind of interpretation and evaluation of the argument list can be done inside the expression body.

In all cases, the return value is the result of the last expression in the body.

```
: (de foo (X Y Z)                  # CAR is a list of symbols
   (list X Y Z) )                  # Return a list of all arguments
-> foo
: (foo (+ 1 2) (+ 3 4) (+ 5 6))
-> (3 7 11)                        # all arguments are evaluated
```

```
: (de foo X                          # CAR is a single symbol
   X )                               # Return the argument
-> foo
: (foo (+ 1 2) (+ 3 4) (+ 5 6))
-> ((+ 1 2) (+ 3 4) (+ 5 6))        # the whole unevaluated list is returned
```

```
   : (de foo @                          # CAR is the symbol '@'
      (list (next) (next) (next)) )     # Return the first three arguments
   -> foo
   : (foo (+ 1 2) (+ 3 4) (+ 5 6))
   -> (3 7 11)                          # all arguments are evaluated
```

Note that these forms can also be combined. For example, to evaluate only the first two arguments, bind the results to `X` and `Y`, and bind all other arguments (unevaluated) to `Z`:

```
: (de foo (X Y . Z)                 # CAR is a list with a dotted-pair tail
   (list X Y Z) )                   # Return a list of all arguments
-> foo
: (foo (+ 1 2) (+ 3 4) (+ 5 6))
-> (3 7 ((+ 5 6)))                  # Only the first two arguments are evaluated
```

Or, a single argument followed by a variable number of arguments:

```
   : (de foo (X . @)                    # CAR is a dotted-pair with '@'
      (println X)                       # print the first evaluated argument
      (while (args)                     # while there are more arguments
         (println (next)) ) )           # print the next one
   -> foo
   : (foo (+ 1 2) (+ 3 4) (+ 5 6))
   3                                    # X
   7                                    # next argument
   11                                   # and the last argument
   -> 11
```

In general, if more than the expected number of arguments is supplied to a function, these extra arguments will be ignored. Missing arguments default to `NIL`.

3.3.5 Coroutines

Coroutines are independent execution contexts. They may have multiple entry and exit points, and preserve their environment between invocations.

They are available only in the 64-bit version.

A coroutine is identified by a tag. This tag can be passed to other functions, and (re)invoked as needed. In this regard coroutines are similar to "continuations" in other languages.

When the tag goes out of scope while it is not actively running, the coroutine will be garabage collected. In cases where this is desired, using a *transient* symbol for the tag is recommended.

A coroutine is created by calling `co`. Its `prg` body will be executed, and unless `yield` is called at some point, the coroutine will "fall off" at the end and disappear.

When `yield` is called, control is either transferred back to the caller, or to some other - explicitly specified, and already running - coroutine.

A coroutine is stopped and disposed when

- execution falls off the end
- some other (co)routine calls `co` with that tag but without a `prg` body
- a `throw` into another (co)routine environment is executed
- an error occurred, and *error handling* was entered

In the current implementation, not more than 64 coroutines can exist at the same time. Reentrant coroutines are not supported, a coroutine cannot resume itself directly or indirectly.

3.3.6 Interrupt

During the evaluation of an expression, the PicoLisp interpreter can be interrupted at any time by hitting `Ctrl-C`. It will then enter the breakpoint routine, as if `!` were called.

Hitting ENTER at that point will continue evaluation, while `(quit)` will abort evaluation and return the interpreter to the top level. See also `debug`, `e`, `^` and `*Dbg`

Other interrupts may be handled by `alarm`, `sigio`, `*Hup` and `*Sig[12]`.

3.3.7 Error Handling

When a runtime error occurs, execution is stopped and an error handler is entered.

The error handler resets the I/O channels to the console, and displays the location (if possible) and the reason of the error, followed by an error message. That message is also stored in the global `*Msg`, and the location of the error in `^`. If the VAL of the global `*Err` is non-`NIL` it is executed as a `prg` body. If the standard input is from a terminal, a read-eval-print loop (with a question mark ‘ `?` ’ as prompt) is entered (the loop is exited when an empty line is input). Then all pending `finally` expressions are executed, all variable bindings restored, and all files closed. If the standard input is not from a terminal, the interpreter terminates. Otherwise it is reset to its top-level state.

```
: (de foo (A B) (badFoo A B))       # 'foo' calls an undefined symbol
-> foo
: (foo 3 4)                         # Call 'foo'
!? (badFoo A B)                     # Error handler entered
badFoo -- Undefined
? A                                 # Inspect 'A'
-> 3
? B                                 # Inspect 'B'
-> 4
?                                   # Empty line: Exit
:
```

Errors can be caught with `catch`, if a list of substrings of possible error messages is supplied for the first argument. In such a case, the matching substring (or the whole error message if the substring is `NIL`) is returned.

An arbitrary error can be thrown explicitly with `quit`.

3.3.8 @ Result

In certain situations, the result of the last evaluation is stored in the VAL of the symbol `@`. This can be very convenient, because it often makes the assignment to temporary variables unnecessary.

This happens in two - only superficially similar - situations:

load In read-eval loops, the last three results which were printed at the console are available in `@@@`, `@@` and `@`, in that order (i.e the latest result is in `@`).

```
: (+ 1 2 3)
-> 6
: (/ 128 4)
-> 32
: (- @ @@)        # Subtract the last two results
-> 26
```

Flow functions Flow- and logic-functions store the result of their controlling expression - respectively non-`NIL` results of their conditional expression - in `@`.

```
: (while (read) (println 'got: @))
abc             # User input
got: abc        # print result
123             # User input
got: 123        # print result
NIL
-> 123

: (setq L (1 2 3 4 5 1 2 3 4 5))
-> (1 2 3 4 5 1 2 3 4 5)
: (and (member 3 L) (member 3 (cdr @)) (set @ 999))
-> 999
: L
-> (1 2 3 4 5 1 2 999 4 5)
```

Functions with controlling expressions are *case*, *prog1*, *prog2*, and the bodies of `*Run` tasks.

Functions with conditional expressions are *and*, *cond*, *do*, *for*, *if*, *if2*, *ifn*, *loop*, *nand*, *nond*, *nor*, *not*, *or*, *state*, *unless*, *until*, *when* and *while*.

`@` is generally local to functions and methods, its value is automatically saved upon function entry and restored at exit.

3.3.9 Comparing

In PicoLisp, it is legal to compare data items of arbitrary type. Any two items are either

Identical They are the same memory object (pointer equality). For example, two internal symbols with the same name are identical. In the 64-bit version, also short numbers (up to 60 bits plus sign) are pointer-equal.

Equal They are equal in every respect (structure equality), but need not to be identical. Examples are numbers with the same value, transient symbols with the same name or lists with equal elements.

Or they have a well-defined ordinal relationship Numbers are comparable by their numeric value, strings by their name, and lists recursively by their elements (if the CAR's are equal, their CDR's are compared). For differing types, the following rule applies: Numbers are less than symbols, and symbols are less than lists. As special cases, `NIL` is always less than anything else, and `T` is always greater than anything else.

To demonstrate this, `sort` a list of mixed data types:

```
: (sort '("abc" T (d e f) NIL 123 DEF))
-> (NIL 123 DEF "abc" (d e f) T)
```

See also `max`, `min`, `rank`, `<`, `= >` etc.

3.3.10 OO Concepts

PicoLisp comes with built-in object oriented extensions. There seems to be a common agreement upon three criteria for object orientation:

Encapsulation Code and data are encapsulated into objects, giving them both a behavior and a state. Objects communicate by sending and receiving messages.

Inheritance Objects are organized into classes. The behavior of an object is inherited from its class(es) and superclass(es).

Polymorphism Objects of different classes may behave differently in response to the same message. For that, classes may define different methods for each message.

PicoLisp implements both objects and classes with symbols. Object-local data are stored in the symbol's property list, while the code (methods) and links to the superclasses are stored in the symbol's VAL (encapsulation).

In fact, there is no formal difference between objects and classes (except that objects usually are anonymous symbols containing mostly local data, while classes are named internal symbols with an emphasis on method definitions). At any time, a class may be assigned its own local data (class variables), and any object can receive individual method definitions in addition to (or overriding) those inherited from its (super)classes.

PicoLisp supports multiple inheritance. The VAL of each object is a (possibly empty) association list of message symbols and method bodies, concatenated

with a list of classes. When a message is sent to an object, it is searched in the object's own method list, and then (with a left-to-right depth-first search) in the tree of its classes and superclasses. The first method found is executed and the search stops. The search may be explicitly continued with the `extra` and `super` functions.

Thus, which method is actually executed when a message is sent to an object depends on the classes that the object is currently linked to (polymorphism). As the method search is fully dynamic (late binding), an object's type (i.e. its classes and method definitions) can be changed even at runtime!

While a method body is being executed, the global variable `This` is set to the current object, allowing the use of the short-cut property functions , `:` and `::`.

3.3.11 Database

On the lowest level, a PicoLisp database is just a collection of *external symbols*. They reside in a database file, and are dynamically swapped in and out of memory. Only one database can be open at a time (`pool`).

In addition, further external symbols can be specified to originate from arbitrary sources via the `*Ext` mechanism.

Whenever an external symbol's value or property list is accessed, it will be automatically fetched into memory, and can then be used like any other symbol. Modifications will be written to disk only when `commit` is called. Alternatively, all modifications since the last call to `commit` can be discarded by calling `rollback`.

Transactions

In the typical case there will be multiple processes operating on the same database. These processes should be all children of the same parent process, which takes care of synchronizing read/write operations and heap contents. Then a database transaction is normally initiated by calling `(dbSync)`, and closed by calling `(commit 'upd)`. Short transactions, involving only a single DB operation, are available in functions like `new!` and methods like `put!>` (by convention with an exclamation mark), which implicitly call `(dbSync)` and `(commit 'upd)` themselves.

A transaction proceeds through five phases:

1. `dbSync` waits to get a `lock` on the root object `*DB`. Other processes continue reading and writing meanwhile.

2. `dbSync` calls `sync` to synchronize with changes from other processes. We hold the shared lock, but other processes may continue reading.
3. We make modifications to the internal state of external symbols with `put>, set>, lose>` etc. We - and also other processes - can still read the DB.
4. We call `(commit 'upd)`. `commit` obtains an exclusive lock (no more read operations by other processes), writes an optional transaction log, and then all modified symbols. As `upd` is passed to 'commit', other processes synchronize with these changes.
5. Finally, all locks are released by 'commit'

Entities / Relations

The symbols in a database can be used to store arbitrary information structures. In typical use, some symbols represent nodes of search trees, by holding keys, values, and links to subtrees in their VAL's. Such a search tree in the database is called index.

For the most part, other symbols in the database are objects derived from the `+Entity` class.

Entities depend on objects of the `+relation` class hierarchy. Relation-objects manage the property values of entities, they define the application database model and are responsible for the integrity of mutual object references and index trees.

Relations are stored as properties in the entity classes, their methods are invoked as daemons whenever property values in an entity are changed. When defining an `+Entity` class, relations are defined - in addition to the method definitions of a normal class - with the `rel` function. Predefined relation classes include

- Primitive types like `+Symbol` Symbolic data `+String` Strings (just a general case of symbols) `+Number` Integers and fixpoint numbers `+Date` Calendar date values, represented by a number `+Time` Time-of-the-day values, represented by a number `+Blob` "Binary large objects" stored in separate files
- Object-to-object relations `+Link` A reference to some other entity `+Hook` A reference to an entity holding object-local index trees `+Joint` A bidirectional reference to some other entity
- Container prefix classes like `+List` A list of any of the other primitive or object relation types `+Bag` A list containing a mixture of any of the other types

- Index prefix classes `+Ref` An index with other primitives or entities as key `+Key` A unique index with other primitives or entities as key `+Idx` A full-text index, typically for strings `+Sn` Tolerant index, using a modified Soundex-Algorithm
- Booleans `+Bool` `T` or `NIL`
- And a catch-all class `+Any` Not specified, may be any of the above relations

3.3.12 Pilog (PicoLisp Prolog)

A declarative language is built on top of PicoLisp, that has the semantics of Prolog, but uses the syntax of Lisp.

For an explanation of Prolog's declarative programming style, an introduction like [2] is recommended.

Facts and rules can be declared with the `be` function. For example, a Prolog fact '`likes(john,mary).`' is written in Pilog as:

```
(be likes (John Mary))
```

and a rule '`likes(john,X) :- likes(X,wine), likes(X,food).`' is in Pilog:

```
(be likes (John @X) (likes @X wine) (likes @X food))
```

As in Prolog, the difference between facts and rules is that the latter ones have conditions, and usually contain variables.

A variable in Pilog is any symbol starting with an at-mark character (' `@` '). The symbol `@` itself can be used as an anonymous variable: It will match during unification, but will not be bound to the matched values.

The *cut* operator of Prolog (usually written as an exclamation mark (`!`)) is the symbol `T` in Pilog.

An interactive query can be done with the `?` function:

```
(? (likes John @X))
```

This will print all solutions, waiting for user input after each line. If a non-empty line (not just a ENTER key, but for example a dot (`.`) followed by ENTER) is typed, it will terminate.

Pilog can be called from Lisp and vice versa:

- The interface from Lisp is via the functions `goal` (prepare a query from Lisp data) and `prove` (return an association list of successful bindings), and the application level functions `pilog` and `solve`.
- When the CAR of a Pilog clause is a Pilog variable, the CDR is executed as a Lisp expression and the result unified with that variable.
- Within such a Lisp expression in a Pilog clause, the current bindings of Pilog variables can be accessed with the `->` function.

3.3.13 Naming Conventions

It was necessary to introduce - and adhere to - a set of conventions for PicoLisp symbol names. Because all (internal) symbols have a global scope (there are no packages or name spaces), and each symbol can only have either a value or function definition, it would otherwise be very easy to introduce name conflicts. Besides this, source code readability is increased when the scope of a symbol is indicated by its name.

These conventions are not hard-coded into the language, but should be so into the head of the programmer. Here are the most commonly used ones:

- Global variables start with an asterisk "`*`"
- Functions and other global symbols start with a lower case letter
- Locally bound symbols start with an upper case letter
- Local functions start with an underscore "`_`"
- Classes start with a plus-sign "`+`", where the first letter
  - is in lower case for abstract classes
  - and in upper case for normal classes
- Methods end with a right arrow "`>`"
- Class variables may be indicated by an upper case letter

For historical reasons, the global constant symbols `T` and `NIL` do not obey these rules, and are written in upper case.

For example, a local variable could easily overshadow a function definition:

```
: (de max-speed (car)
   (.. (get car 'speeds) ..) )
-> max-speed
```

Inside the body of `max-speed` (and all other functions called during that execution) the kernel function `car` is redefined to some other value, and will surely crash if something like `(car Lst)` is executed. Instead, it is safe to write:

```
: (de max-speed (Car)            # 'Car' with upper case first letter
   (.. (get Car 'speeds) ..) )
-> max-speed
```

Note that there are also some strict naming rules (as opposed to the voluntary conventions) that are required by the corresponding kernel functionalities, like:

- Transient symbols are enclosed in double quotes (see *Transient Symbols*)
- External symbols are enclosed in braces (see *External Symbols*)
- Pattern-Wildcards start with an at-mark ' `@` ' (see *match* and *fill*)
- Symbols referring to a shared library contain a colon ' `lib:sym` '

With that, the last of the above conventions (local functions start with an underscore) is not really necessary, because true local scope can be enforced with transient symbols.

3.3.14 Breaking Traditions

PicoLisp does not try very hard to be compatible with traditional Lisp systems. If you are used to some other Lisp dialects, you may notice the following differences:

Case Sensitivity PicoLisp distinguishes between upper case and lower case characters in symbol names. Thus, `CAR` and `car` are different symbols, which was not the case in traditional Lisp systems.

QUOTE In traditional Lisp, the `QUOTE` function returns its *first* unevaluated argument. In PicoLisp, on the other hand, `quote` returns *all* (unevaluated) argument(s).

LAMBDA The `LAMBDA` function, in some way at the heart of traditional Lisp, is completely missing (and `quote` is used instead).

PROG The `PROG` function of traditional Lisp, with its GOTO and ENTER functionality, is also missing. PicoLisp's `prog` function is just a simple sequencer (as `PROGN` in some Lisps).

Function/Value In PicoLisp, a symbol cannot have a value *and* a function definition at the same time. Though this is a disadvantage at first sight, it allows a completely uniform handling of functional data.

3.3.15 Bugs

The names of the symbols `T` and `NIL` violate the *naming conventions*. They are global symbols, and should therefore start with an asterisk "`*`". It is too easy to bind them to some other value by mistake:

```
(de foo (R S T)
   ...
```

However, `lint` will issue a warning in such a case.

References

1. Winston/Horn: "Lisp", Addison-Wesley, 1981
2. Clocksin/Mellish: "Programming in Prolog", Springer-Verlag, 1981

4

The Equivalence of Code and Data

Alexander Burger

abu@software-lab.de

Summary. This article demonstrates how the *equivalence of code and data*, as a powerful abstraction instrument, supports the "locality principle of structure, content and behavior" in PicoLisp.

4.1 The Equivalence of Code and Data

Why does PicoLisp make so much fuss about that equivalence of code and data? Answer: Because it is such a powerful abstraction instrument! It supports the "locality principle of structure, content and behavior".

Let me try to explain this using a GUI example, though the same principle also applies to most other libraries and applications. Take a simple GUI page:

```
(action
   (html 0 "Increment" "lib.css" NIL
      (form NIL
         (gui '(+Var +NumField) '*Num 9)
         (gui '(+JS +Button) "+" '(inc '*Num)) ) ) )
```

This page shows two components, a numeric input field and a button labeled "+". You can either enter a value into that field, or increment its value with the button.

The page has a *structure*, namely a HTML body containing a form, which in turn contains the two components. It has *content*, the numeric value shown in the field. And it has *behavior*, as the numeric field handles string conversion and error checking, and the button increments the number.

Note that a fourth attribute of the page, the *layout and style*, is not discussed here. It is maintained separately in the CSS file.

The three attributes structure, content and behavior, however, cannot be separated. This may be contrary to some schools of thought, which advocate a separation of structure and content, and behavior anyway. But obviously the HTML structure also holds content like the "Increment" page title or the "+" button label. And this content might also be subject to behavior, for example the button label may change dynamically:

```
(gui '(+JS +Button)
   (if *VerboseDisplay "Increment" "+")
   '(inc '*Num) )
```

Or the button may prompt a confirmation dialog to the user:

```
(gui '(+JS +Button) "+"
   '(ask "Increment value?"
      (inc '*Num) ) )
```

The equivalence of code and data makes the development of functions like `html`, `form`, `gui` or `ask` particularly simple. The list

```
(ask "Increment value?" (inc '*Num))
```

is *data* for the `gui` function. `gui` constructs a component of the class `+Button`, and simply stows away that list into the new component. Later, when the button is pressed, it is executed. It is just that simple.

`gui` itself doesn't understand anything about buttons and what they do with these arguments. It is all data. From the view of the button, however, that list is a piece of code, to be executed when the button is pressed.

Later, when `ask` is executed, the situation is similar. A dialog is shown which displays the question "Increment value?", and a "Yes" and a "No" button. The "Yes" button receives the expression `(inc '*Num)`, to increment the number when pressed.

The example can be modified, to operate on database entities instead. Instead of connecting the numeric field to a variable (with the `+Var` prefix class), we connect it to the `count` property of the database object, and instead of incrementing the variable, the button increments that object's count attribute:

```
(gui '(+E/R +NumField) '(count : home obj) 9)
(gui '(+JS +Button) "+" '(inc!> (: home obj) 'count))
```

No need to write some separate database "select" or "update" functionality! It is all in a single place.

This shows how easily any kind of application logic can be implemented using this style. In a typical application hundreds or thousands of such components are put together, so it is essential that this can be done with as little noise and redundancy as possible.

This becomes especially evident if you compare this style with the GUI and application strategies of other languages, for example in the RosettaCode tasks GUI component interaction[1] and GUI enabling/disabling of controls[2].

In other environments, you usually end up defining constants, components and functions in separate locations. This is the opposite of what PicoLisp advocates, the "locality principle of structure, content and behavior".

[1] http://rosettacode.org/wiki/GUI_component_interaction#PicoLisp
[2] http://rosettacode.org/wiki/GUI_enabling/disabling_of_controls#PicoLisp

5

First Class Environments

Alexander Burger

abu@software-lab.de

Summary. This article discusses the advantages and technical details of *first class environments* in PicoLisp. First, the differences between *dynamic binding* and *lexical scoping* are highlighted, then *first class data types* and *first class environments* are discussed.

5.1 Dynamic Binding vs Lexical Scoping

PicoLisp uses dynamic binding for symbolic variables. This means that the value of a symbol is determined by the current runtime context, not by the lexical context in the source file.

This has advantages in practical programming. It allows you to write independent code fragments as **data** which can be passed to other parts of the program to be later executed as **code** in that context.

For example, the major part of the PicoLisp GUI framework consists of code snippets, which are installed as event handlers, button actions or update managers in GUI components.

In a lexically scoped language, you can't separate code from its environment. A call like `(foo X Y)` has no meaning by itself (unless `X` and `Y` are treated as "free" variables, i.e. dynamically, not lexically). Instead, you must write a `let` or `lambda` expression, or pass around full function definitions as closures. All this is much more noisy (consider the typical case of dozens of such fragments in a single GUI page).

PicoLisp tries to be as dynamic as possible. This involves also (and much more important than just variable bindings) things like I/O-channels, 'make[1]' environments, the place of the currently executed method in the inheritance

[1] http://software-lab.de/doc/refM.html#make

hierarchy (for 'super[2]' and 'extra[3]') or the current 'This[4]' object. A running code fragment can always assume to have access to these environments.

This article is only concerned about variable bindings.

5.2 First Class Data Type

In programming languages, a first class data type can be created and manipulated separately within the language, independent from its intended purpose.

For a **function**, for example, the intended purpose is to *call* it. If it is a first class function, it can be build, passed around etc., independent from that. The two tasks (creating and calling the function) can be handled independently within the language.

For an **environment**, the intended purpose is to

1. activate it
2. execute one or more expressions in it
3. deactivate it, possibly restoring the previous environment

If it is a first class environment it, can also be created and manipulated within the language.

5.3 First Class Environments

To make such an environment useful, it is important that the three tasks (creation, activation/deactivation and execution) can be controlled independently.

Let us consider the expression `(foo X Y)` again. Assume we want to execute it in an environment where `X` is 3 and `Y` is 4. The direct way is

```
(let (X 3 Y 4)
   (foo X Y) )
```

This handles the three tasks all at once. It creates and activates the environment, immediately executes the code, and then deactivates the environment.

[2] http://software-lab.de/doc/refS.html#super
[3] http://software-lab.de/doc/refE.html#extra
[4] http://software-lab.de/doc/refT.html#This

If the expression `(foo X Y)` is supplied from the application code, stand-alone in one part of the application, and is supposed to run in some other part of the application in an environment active there at that moment, it is also no problem. It can simply be executed:

```
(foo X Y)
```

If, however, that application environment supposed to be manipulated separately, for example because it is passed over across a HTTP transaction or appears in an RPC call, then the

- creation of that environment is typically done in the application
- activation of that environment is done in the GUI framework
- execution of the expression(s) is done in the application
- deactivation is done in the GUI framework again

5.3.1 Creation

The environment is represented by a list of symbol-value pairs. It can be created with the direct, explicit lisp operations, or more conveniently with the 'env[5]' function.

If the environment is intended as a subset of the current environment, i.e. if you simply want to create it with the current values of `X` and `Y`, you may write

```
: (setq Env (env '(X Y)))
-> ((Y . 4) (X . 3))
```

Otherwise, you may pass explit values

```
: (setq Env (env 'X 3 'Y 4))
-> ((Y . 4) (X . 3))
```

5.3.2 Activation

The simplest way to activate an environment would be to to iterate over the list and 'set[6]' each symbol to its value. This is normally not recommended, because it would not restore the previous environment when done.

[5] http://software-lab.de/doc/refE.html#env
[6] http://software-lab.de/doc/refS.html#set

In most cases either 'bind[7]' or 'job[8]' are used. The main difference between these two functions is that job modifies the environment destructively, to store modified values before restoring the previous environment.

```
: (bind Env (* X Y))
-> 12

: (job Env (* X Y))
-> 12
```

If the environment is to be modified by the expression, use 'job':

```
: Env
-> ((Y . 4) (X . 3))

: (job Env (* X (inc 'Y)))
-> 15

: Env
-> ((Y . 5) (X . 3))
```

These examples don't show the separation of activation and execution, as this cannot be simply done in an isolated example. For a real-world use case, take a look at the top-level GUI function in "@lib/form.l". The relevant part can be reduced to

```
(with "*App"                           # Point 'This' to the current form
   (job (: env)                        # Activate environment of that form
      (<post> ...
         (<hidden> ...)
         ...
         (if *PRG
            (let gui
               ...
               (htPrin "Prg") )      # Execute the GUI code
            ...
            (let gui
               ...
               (htPrin "Prg") )      # Execute the GUI code
```

[7] http://software-lab.de/doc/refB.html#bind
[8] http://software-lab.de/doc/refJ.html#job

The GUI code which actually builds the page is in `"Prg"`. When it runs, `This` and the desired environment (from the form's `env` property) are properly activated.

A similar case can be found more down in "@lib/form.l" in 'postGui' to handle HTTP POST and XMLHttpRequest events.

6

Even small details make a difference!

Alexander Burger

abu@software-lab.de

Summary. This article discusses how it's unique implementation of the `quote` function makes PicoLisp more efficient than other Lisp's in terms of both space and time.

6.1 Even small details make a difference!

In Lisp, the single quote character plays a central role. It is actually a *read macro*, which expands to the built-in function `quote`. Its purpose is to inhibit the evaluation of the following argument, and `quote` is one of the most often called functions.

But it is remarkable that almost all Lisp implementations handle this in a somewhat suboptimal way. They define `quote` to return its *first* argument, without evaluating it. So far so good. But when you think about that for a moment, you may ask: "Why only the first argument?"

This is inefficient, both in terms of space and time, and has no benefit at all. Yet almost all Lisp implementations stick to it.

PicoLisp breaks with that bad habit, and defines `quote` to return *all* arguments, without evaluating them.

Let's look at some examples: Quoted expressions like

```
'a
'(a b)
'(a . x)
```

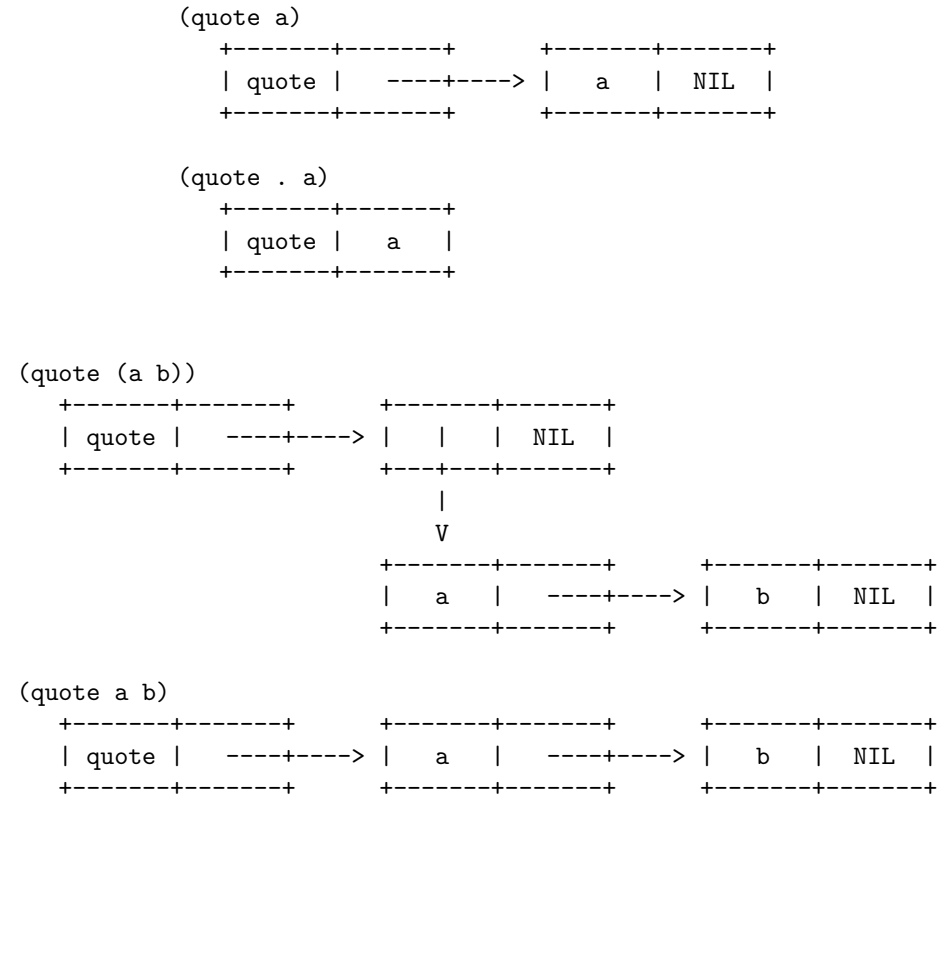
expand in traditional Lisps to

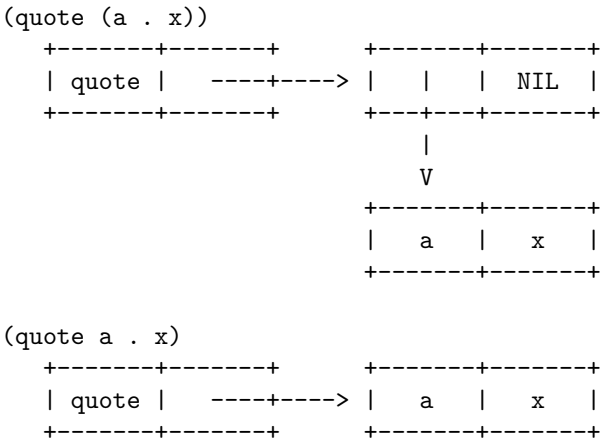
```
(quote a)
(quote (a b))
(quote (a . x))
```

while in PicoLisp, they expand to

```
(quote . a)
(quote a b)
(quote a . x)
```

This makes quite a difference, as it uses only half the space (two cells in traditional Lisp, but only one in PicoLisp):





The runtime code for a traditional `quote` has more work to do. It needs two pointer dereferences to get the data, while PicoLisp needs only one.

For that reason, `quote` is the shortest of all functions in PicoLisp, being

```
any doQuote(any x) {return cdr(x);}
```

in `C`, or

```
(code 'doQuote 2)
   ld E (E CDR)  # Get CDR
   ret
```

in `asm`.

7

The Dual Nature of NIL

Alexander Burger

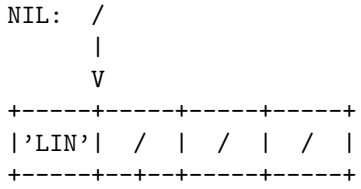
abu@software-lab.de

Summary. In this article, the dual nature of `NIL` as symbol and cons pair, a basic design requirement of PicoLisp, is discussed.

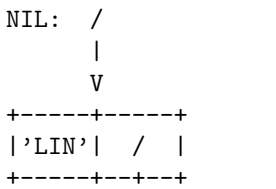
7.1 The Dual Nature of NIL

`NIL` is a very fundamental piece of data. It has a dual nature, being both a symbol *and* a cons pair of the form `(NIL . NIL)`. From the beginning, this has been a basic design requirement of PicoLisp.

As shown in “doc64/structures“ (similar also in ”doc/structures“):

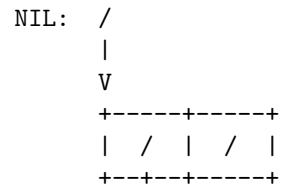


Physically, the pointer to `NIL` (shown with the 'V' in the above diagram) is a true symbol, as it points at an offset of a pointer size into the memory heap. Taken as such, `NIL` is a normal symbol,



having a tail with the characters 'N', 'I' and 'L' (the name), and a value which in turn points to `NIL` (symbolized with the '/').

However, when this pointer is boldly used as a cell pointer (by ignoring the pointer tags caused by the pointer size offset)



then it is a normal cons pair, with `NIL` in its CAR and `NIL` in its CDR, which just happens to be misaligned in memory (i.e. at the place of a symbol).

This structure has great advantages. Any proper list (ending with `NIL`) becomes sort of "infinite", allowing to take the CDR as often as possible and still obtain `NIL` again and again.

Therefore, a function doesn't need to check whether it actually received an argument or not. It can simply take the next argument with CDR from the argument list, and doesn't see any difference between `(foo NIL)` and `(foo)`. This makes interpretation both simpler and faster.

8

Array Abstinence

Alexander Burger

abu@software-lab.de

Summary. The differences between lists and arrays as well as the question if arrays are really needed are discussed, followed by relative performance considerations.

8.1 Introduction

Some people are unhappy with the situation that PicoLisp has no built-in array (or vector) data type. As described in the reference manual, the PicoLisp Machine[1] supports exactly three data types[2]: Numbers, Symbols and Lists.

But what about arrays? After all, any other language supports arrays! We are so very much used to them.

If we look at a typical Fortran, C or Java program, we find almost every second statement to be something like (e.g. in Java):

```
for (i = 0; i < Arr.length; ++i)
   doSomething(Arr[i]);
```

It is nearly impossible to survive in such languages without arrays.

8.2 What is an Array?

An array is a wonderful data structure.

[1] http://software-lab.de/doc/ref.html#vm
[2] http://software-lab.de/doc/ref.html#data

It implements a mapping of numeric keys (integers) to arbitrary data. Given a number, the associated data can be found in O(1) time. And the keys themselves do not take up any memory space, they exist implicit in the data offsets form the array's start address.

8.3 Lists

PicoLisp has lists.

A list can simulate an array, albeit less efficiently. It takes up twice the space, and indexed access to its data takes O(N/2) time on the average.

But lists have a lot of advantages, and are also more efficient than arrays in many other cases. We do not need to discuss these advantages here in detail, any Lisp programmer will know them.

As PicoLisp strives for simplicity, there is no room for an additional data type - also for hard physical reasons, as the number of precious tag bits is limited. To fully support an array data type, all those powerful built-in list manipulation functions would need array counterparts, and the programmer would permanently need to make design decisions whether to use lists or arrays for the problem at hand.

If there is a choice between arrays **or** lists, lists will clearly win.

8.4 Are Arrays *Really* Needed?

My short answer would be "No". The advantages in efficiency do not outweigh the disadvantages of increased complexity.

I would even say that the proponents of arrays in PicoLisp misunderstand some aspects of PicoLisp programming.

The predominance of arrays in other languages, as shown with the above example

```
for (i = 0; i < Arr.length; ++i)
   doSomething(Arr[i]);
```

does not carry over to PicoLisp. Such constructs are simply not used. Instead, you have a large number of mapping functions at your disposal:

```
(mapc doSomething Arr)
```

In fact, it is the lack of functionality for the direct manipulation of compound data, which requires the excess indexing into arrays in other languages.

The mistake is mixing up two separate concepts:

1. Sequential access to certain pieces of data
2. Mapping integer keys to data items

What arrays are *really* needed for is point (2): You have an integer, and want to get the corresponding piece of data.

Point (1), however, can and should be handled directly and elegantly with mapping and other access functions.

So back to point (2). *When* in our programming life do we really need to map integers to data?

These cases are surprisingly rare. We have to keep in mind that for arrays only the mapping of continuous integer ranges makes sense. Most practical tasks of mapping numbers to other data involve sparse or non-linear input data, or non-integers, or no numeric keys at all.

Sure, there are typical cases like image rasterization. But when did you the last time implement Bresenham's line algorithm in *application* programming? Instead, you resort to a library, or write it in C or even assembly if a library is not available.

In other cases - like two-dimensional maps - there are better ways. Look for example how the board in the PicoLisp chess program (and other games and many rosetta code solutions) is implemented with direct connection attributes (north, west etc.) between the fields instead of integer arithmetics for array indexes.

For more flexible (not only integer, or even numeric) key mapping, built-in mechanisms like association lists, symbol properties or binary 'idx[3]' trees can be used. Agreed, these mechanisms are less efficient than simple integer-indexed arrays, but the difference is not very dramatic.

8.5 Relative Performance Consideration

What would be the performance penalty for using lists instead of arrays, if we would really need a mapping of a continuous integer range to other data?

Let's assume an integer array of length 100, initialized to increasing values:

[3] http://software-lab.de/doc/refI.html#idx

```
int Arr[100];

for (i = 0; i < 100; ++i)
   Arr[i] = i;
```

In PicoLisp, this is equivalent to

```
(setq List (range 1 100))
```

Now let's increment each element:

```
for (i = 0; i < 100; ++i)
   ++Arr[i];
```

The PicoLisp version would be

```
(map inc List)
```

but let's assume for a moment we have no mapping function, and insist on indexed access instead:

```
(for N 100
   (inc (nth List N)) )
```

'nth[4]' is PicoLisp's idea of an indexed array access.

Let's compare that to an analog O(1) access, by repeatedly accessing the first list element:

```
(for N 100
   (inc (nth List 1)) )
```

The time difference will be the relative overhead for the O(N/2) access to indexed list elements.

To get measurable timing results, we do each test ten thousand times. First the accesses to all list elements:

[4] http://software-lab.de/doc/refN.html#nth

```
: (bench
   (do 10000
      (for N 100
         (inc (nth List N)) ) ) )
0.282 sec
```

Then the "simulation of an array data type", by accessing only the first element:

```
: (bench
   (do 10000
      (for N 100
         (inc (nth List 1)) ) ) )
0.121 sec
```

We can see, the results are in the same order of magnitude. The difference of 161 milliseconds was the time actually spent in list traversals.

In fact, the greatest part of execution time is taken by the interpreter overhead anyway. Compare this to a full-list increment using 'map[5]':

```
: (bench
   (do 10000
      (map inc List) ) )
0.076 sec
```

Of course the difference gets bigger if the list gets longer, but typically there will be also much more processing of the data than simple increments.

Regarding all that, it should become clear that wasting tag bits and efforts for such a data type and its associated functions is not a good idea.

[5] http://software-lab.de/doc/refM.html#map

9

Coroutines

Alexander Burger

abu@software-lab.de

Summary. This article introduces application examples for *coroutines*, comparing their use and relative efficiency with a possible solution via *generators*.

9.1 Introduction

With picoLisp-3.0.3, the 64-bit version of PicoLisp has support for coroutines[1]. This article tries to show their basic usage.

Assume we need all Pythagorean triples (i.e. all numbers A, B and C, such that $A + B = C$ with elements between 1 and N. A straightforward way to print them is:

```
(de pythag (N)
   (for A N
      (for B (range A N)
         (for C (range B N)
            (when (= (+ (* A A) (* B B)) (* C C))
               (println (list A B C)) ) ) ) ) )
```

We get:

[1] http://software-lab.de/doc/ref.html#coroutines

```
: (pythag 20)
(3 4 5)
(5 12 13)
(6 8 10)
(8 15 17)
(9 12 15)
(12 16 20)
```

However, just printing the triples is not very useful if they were needed in various situations for further processing. The lispy way for a general tool is passing a *function* to `pythag` and decide later what to do with the data:

```
(de pythag (N Fun)
   (for A N
      (for B (range A N)
         (for C (range B N)
            (when (= (+ (* A A) (* B B)) (* C C))
               (Fun (list A B C)) ) ) ) ) )
```

(note that for a truly general tool, we would write `Fun` and the local variables as transient symbols to avoid conflicts)

Now we can print them again

```
: (pythag 20 println)
(3 4 5)
(5 12 13)
(6 8 10)
(8 15 17)
(9 12 15)
(12 16 20)
```

collect them into a list if we like

```
: (make (pythag 20 link))
-> ((3 4 5) (5 12 13) (6 8 10) (8 15 17) (9 12 15) (12 16 20))
```

or do with them whatever we like.

Still, this method has its limits. What if we need to pass a very large number for `N`, and we want to access the values one by one, perhaps in the course of some other involved calculation? Pre-generating the list of all values might not be feasible.

9.2 Using a Generator

One way to solve this problem is a generator. This function returns the next value each time it is called:

```
(de pythag (N)
   (job '((A . 1) (B . 1) (C . 0))
      (loop
         (when (> (inc 'C) N)
            (when (> (inc 'B) N)
               (setq B (inc 'A)) )
            (setq C B) )
         (T (> A N))
         (T (= (+ (* A A) (* B B)) (* C C))
            (list A B C) ) ) ) )

: (pythag 20)
-> (3 4 5)
: (pythag 20)
-> (5 12 13)
: (pythag 20)
-> (6 8 10)
...
```

Now we can call it whenever we need a new value. The function encapsulates the state of its local variables in a job[2] environment.

A major disadvantage, however, is that it does not reflect the flow of control. The three nested `for` loops above had to be unfolded and programmed manually. This is hard to read, even for this simple case, and may be difficult or impossible to program in more complicated cases.

9.3 Using a Coroutine

A coroutine preserves the local environment, as well as the state of control of a function. It may have multiple exit points, and continue execution where it left off the last time.

This is done via two new functions: co[3] and yield[4].

[2] http://software-lab.de/doc/refJ.html#job
[3] http://software-lab.de/doc/refC.html#co
[4] http://software-lab.de/doc/refY.html#yield

```
(de pythag (N)
   (co 'pythag
      (for A N
         (for B (range A N)
            (for C (range B N)
               (when (= (+ (* A A) (* B B)) (* C C))
                  (yield (list A B C)) ) ) ) ) ) )

: (pythag 20)
-> (3 4 5)
: (pythag 20)
-> (5 12 13)
: (pythag 20)
-> (6 8 10)
...
```

So this is a generator equivalent to the one above, but with cleanly nested `for` loops.

The function `co` is called with a **tag** argument, and an executable **body** (a *prg*), similar to catch[5]. When called the first time, a new coroutine for that tag is created, and the body gets executed. If called again later, and an existing coroutine for that tag is found, execution will continue at the point where it left off last time with `yield`.

`yield` stops executing the coroutine's body, and immediately returns to the caller (or to some other coroutine if desired). When a coroutine is resumed with `co`, it will continue at the point of the last call to `yield`. If it is resumed by a call to `yield` in another coroutine, the return value of the first `yield` is the value given as an argument to the second `yield`.

9.4 Efficiency

In terms of memory usage, coroutines are rather expensive, because each of them requires its own stack segment. For that reason (and also due to internal structures) the maximum number of coroutines in the system is limited to 64.

To my surprise, however, coroutines are quite efficient in terms of runtime overhead. Measuring the context switch to and from an empty coroutine (an endless loop with just a (yield))

[5] http://software-lab.de/doc/refC.html#catch

```
: (bench (do 1000000 (co 'bench (loop (yield)))))
0.380 sec
-> NIL
```

shows that it needs just 0.38 / 2 = 0.19 microseconds per switch operation.

This is in the same order of magnitude of a normal function call:

```
: (bench (do 1000000 ((quote (X Y) (+ X Y)) 3 4)))
0.162 sec
-> 7
```

Comparing the implementations of `pythag` above - as a generator using `job` and a coroutine - shows that the coroutine version is about 10 percent faster.

9.5 Inspecting and Stopping Coroutines

The function stack[6] can be used to see which coroutines are currently running (in addition to its primary task of setting or returning the current stack segment size).

Let's say we called `pythag` like in the example above, and then started two more coroutines as

```
: (co "routine1" (yield 1))
-> 1
: (co "routine2" (yield 2))
-> 2
```

Now there must be three coroutines running. `stack` returns them:

```
: (stack)
-> ("routine2" "routine1" pythag . 4)
```

When `co` is called with only a `tag`, but without a body, the corresponding coroutine is stopped.

[6] http://software-lab.de/doc/refS.html#stack

```
: (co "routine1")
-> T
: (co "routine2")
-> T
: (co 'pythag)
-> T
: (stack)
-> 4
```

Now all coroutines are gone, and `stack` returns only the remaining segment size (in megabytes).

9.6 A Tree Example

A typical example of a situation where a coroutine can simplify things a lot, is a recursive algorithm.

For testing, let's generate a balanced binary tree:

```
(balance '*Tree (range 1 15))
```

As you may know, you can display it with the view[7] function

```
: (view *Tree T)
         15
      14
         13
   12
         11
      10
         9
8
         7
      6
         5
   4
         3
      2
         1
```

It is easy to traverse such a tree, e.g. to print its nodes:

[7] http://software-lab.de/doc/refV.html#view

```
: (de printNodes (Tree)
   (when Tree
      (printNodes (cadr Tree))
      (println (car Tree))
      (printNodes (cddr Tree)) ) )
-> printNodes

: (printNodes *Tree)
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

But what if you want to get the nodes returned, one by one, as in a generator function? For example, to compare the nodes of two trees and see of they are equal? If you think about it, you'll see that it is not trivial.

With a coroutine, it is straightforward. We can write a function that returns another node upon each call:

```
(de nextLeaf (Rt Tree)
   (co Rt
      (recur (Tree)
         (when Tree
            (recurse (cadr Tree))
            (yield (car Tree))
            (recurse (cddr Tree)) ) ) ) )
```

With that, we can write a function to compare two trees

```
(de cmpTrees (Tree1 Tree2)
   (prog1
      (use (Node1 Node2)
         (loop
            (setq
               Node1 (nextLeaf "rt1" Tree1)
               Node2 (nextLeaf "rt2" Tree2) )
            (T (nor Node1 Node2) T)
            (NIL (= Node1 Node2)) ) )
      (co "rt1")
      (co "rt2") ) )
```

The last two calls to `co` stop the two coroutines, independent of the result.

10

Transient Namespaces

Alexander Burger

abu@software-lab.de

Summary. This articles is about using use a transient symbol (instead of an internal symbol) for a *temporary* namespace.

10.1 Introduction

A few days ago I hit upon an interesting concept: Transient Namespaces.

With Transient Namespaces I do not mean namespaces for transient symbols (there is always only one single namespace for transient symbols at any moment), but to use a transient symbol (instead of an internal symbol) for a *temporary* namespace.

It turns out that this makes quite some sense.

10.2 Using transient symbols

Last week I added an example for the Common-Lisp-style `DO*` function to macros[1]

```
(de do* "Args"
   (bind (mapcar car (car "Args"))
      (for "A" (car "Args")
         (set (car "A") (eval (cadr "A"))) )
      (until (eval (caadr "Args"))
         (run (cddr "Args"))
         (for "A" (car "Args")
```

[1] http://software-lab.de/doc/faq.html#macros

```
            (and (cddr "A") (set (car "A") (run @))) ) )
      (run (cdadr "Args")) ) )
```

It can be used as

```
: (do* ((A 123) (B 'abc) (C 1 (inc C)))
   ((> C 7) (pack A B))
   (println C) )
1
2
3
4
5
6
7
-> "123abc"
```

As you see, the above implementation of `do*` uses *transient* symbols for `"A"` and `"Args"`. This is the recommended way, to avoid symbol conflicts ("captures" in CL-terminology).

10.3 Using internal symbols

An implementation with *internal* symbols

```
(de do* Args
   (bind (mapcar car (car Args))
      (for A (car Args)
         (set (car A) (eval (cadr A))) )
      (until (eval (caadr Args))
         (run (cddr Args))
         (for A (car Args)
            (and (cddr A) (set (car A) (run @))) ) )
      (run (cdadr Args)) ) )
```

looks better, but gives a wrong result: Because the symbol `A` is captured, the function returns `"abc"` instead of `"123abc"`.

So it is wise to use transient symbols, as in the first version.

10.4 Using transient namespaces

10.4.1 Implementation

However, as PicoLisp has namespaces for internal symbols (in the 64-bit version since 3.0.7.9, and in Ersatz PicoLisp since 3.1.0.10), there is a third way:

Define `do*` in a separate namespace, and keep `A` and `Args` locally:

```
(symbols 'flow 'pico)

(local Args A)

(de pico~do* Args
   (bind (mapcar car (car Args))
      (for A (car Args)
         (set (car A) (eval (cadr A))) )
      (until (eval (caadr Args))
         (run (cddr Args))
         (for A (car Args)
            (and (cddr A) (set (car A) (run @))) ) )
      (run (cdadr Args)) ) )
```

The 'symbols[2]' call in `(symbols 'flow 'pico)` creates a new namespace `flow` as a copy of the pico[3] namespace, and activates it.

`(local Args A)` ensures that `Args` and `A` are local[4] to the new namespace.

`pico do*` refers to the existing or newly created symbol `do*` in the `pico` namespace.

Now, back in the `pico` namespace (e.g. after hitting EOF), `do*` is available

```
: (pp 'do*)
(de do* "Args"
   (bind (mapcar car (car "Args"))
      (for "A" (car "Args")
         (set (car "A") (eval (cadr "A"))) )
      (until (eval (caadr "Args"))
         (run (cddr "Args"))
         (for "A" (car "Args")
            (and (cddr "A") (set (car "A") (run @))) ) )
      (run (cdadr "Args")) ) )
-> do*
```

[2] http://software-lab.de/doc/refS.html#symbols
[3] http://software-lab.de/doc/refP.html#pico
[4] http://software-lab.de/doc/refL.html#local

As you see, `"Args"` and `"A"` are transient symbols relative to the `pico` namespace, as the symbols `Args` and `A` are internal to the `flow` namespace.

10.4.2 Drawback

There is just one serious drawback to this approach. The call

```
(symbols 'flow 'pico)
```

creates the new namespace `flow` as a copy of `pico`. For a typical namespace, this eats up about 25 kB of memory. If you get into the habit of creating a lot of such namespaces, it could become quite wasteful.

Here **Transient Namespaces** come into play: Just create the namespace as

```
(symbols "flow" 'pico)
```

i.e. use the transient symbol `"flow"` instead of the internal symbol `flow`. While it still creates a full copy of `pico`, it will become garbage as soon as `"flow"` goes out of scope, and no space is wasted in the long term!

Normally, you would create a library for such flow functions in a separate source file, containing more than a single function definition. To be sure to avoid symbol conflicts also between those function, you may use more than one call to local[5] (analog to '====[6]' calls for separating transient symbols):

```
(symbols "flow" 'pico)

(local Args A)

(de pico~do* Args
   (bind (mapcar car (car Args))
      (for A (car Args)
         (set (car A) (eval (cadr A))) )
      (until (eval (caadr Args))
         (run (cddr Args))
         (for A (car Args)
            (and (cddr A) (set (car A) (run @))) ) )
      (run (cdadr Args)) ) )

(local X Y Prg)

(do pico~mumble (X Y . Prg)
   ... )
...
```

[5] http://software-lab.de/doc/refL.html#local
[6] http://software-lab.de/doc/ref_.html#====

11

Native C Calls

Alexander Burger

abu@software-lab.de

Summary. This article describes how to call C functions in shared object files (libraries) from PicoLisp, using the built-in `native` function – possibly with the help of the `struct` and `lisp` functions. It applies only to the 64-bit version of PicoLisp.

11.1 Overview

`native` calls a C function in a shared library. It tries to

1. find a library by name
2. find a function by name in the library
3. convert the function's argument(s) from Lisp to C structures
4. call the function's C code
5. convert the function's return value(s) from C to Lisp structures

The direct return value of `native` is the Lisp representation of the C function's return value. Further values, returned by reference from the C function, are available in Lisp variables (symbol values).

`struct` is a helper function, which can be used to manipulate C data structures in memory. It may take a scalar (a numeric representation of a C value) to convert it to a Lisp item, or (more typically) a pointer to a memory area to build and extract data structures. `lisp` allows you to install callback functions, callable from C code, written in Lisp.

In combination, these three functions can interface PicoLisp to almost any C function.

The above steps are fully dynamic; `native` doesn't have (and doesn't require) a priory knowledge about the library, the function or the involved data. No

need to write any glue code, interfaces or include files. All functions can even be called interactively from the REPL.

11.2 Syntax

The arguments to `native` are

1. a library
2. a function
3. a return value specification
4. optional arguments

The simplest form is a call to a function without return value and without arguments. If we assume a library "lib.so", containing a function with the prototype

```
void fun(void);
```

then we can call it as

```
(native "lib.so" "fun")
```

11.2.1 Libraries

The first argument to `native` specifies the library. It is either the *name* of a library (a symbol), or the *handle* of a previously found library (a number).

As a special case, a transient symbol `"@"` can be passed for the library name. It then refers to the current main program (instead of an external library), and can be used for standard functions like `"malloc"` or `"printf"`.

`native` uses `dlopen(3)` internally to find and open the library, and to obtain the handle. If the name contains a slash ('/'), then it is interpreted as a (relative or absolute) pathname. Otherwise, the dynamic linker searches for the library according to the system's environment and directories. See the man page of `dlopen(3)` for further details.

If called with a symbolic argument, `native` automatically caches the handle of the found library in the value of that symbol. The most natural way is to pass the library name as a *transient* symbol (`"lib.so"` above): The initial value of a transient symbol is that symbol itself, so that `native` receives the

library name upon the first call. After successfully finding and opening the library, `native` stores the handle of that library in the value of the passed symbol (`"lib.so"`). As `native` evaluates its arguments in the normal way, subsequent calls within the same transient scope will receive the numeric value (the handle), and don't need to open and search the library again.

11.2.2 Functions

The same rules applies to the second argument, the function. When called with a symbol, `native` stores the function pointer in its value, so that subsequent calls evaluate to that pointer, and `native` can directly jump to the function.

`native` uses `dlsym(3)` internally to obtain the function pointer. See the man page of `dlsym(3)` for further details.

In most cases a program will call more than one function from a given library. If we keep the code within the same transient scope (i.e. in the same source file, and not separated by the `====` function), each library will be opened – and each function searched – only once.

```
(native "lib.so" "fun1")
(native "lib.so" "fun2")
(native "lib.so" "fun3")
```

After `"fun1"` was called, `"lib.so"` will be open, and won't be re-opened for `"fun2"` and `"fun3"`. Consider the definition of helper functions:

```
(de fun1 ()
   (native "lib.so" "fun1") )

(de fun2 ()
   (native "lib.so" "fun2") )

(de fun3 ()
   (native "lib.so" "fun3") )
```

After any one of `fun1`, `fun2` or `fun3` was called, the symbol `"lib.so"` will hold the library handle. And each function function `"fun1"`, `"fun2"` and `"fun3"` will be searched only when called the first time.

Warning: It should be avoided to put more than one library into a single transient scope if there is a chance that two different functions with the same name will be called in two different libraries. Because of the function pointer caching, the second call would otherwise (wrongly) go to the first function.

11.2.3 Return Value

The (optional) third argument to `native` specifies the return value. A C function can return many types of values, like integer or floating point numbers, string pointers, or pointers to structures which in turn consist of those types, and even other structures or pointers to structures. `native` tries to cover most of them.

As described in the *result specification*, the third argument should consist of a pattern which tells `native` how to extract the proper value.

Primitive Types

In the simplest case, the result specification is `NIL` like in the examples so far. This means that either the C function returns `void`, or that we are not interested in the value. The return value of `native` will be `NIL` in that case.

If the result specification is one of the symbols `B`, `I` or `N`, an integer number is returned, by interpreting the result as a `char` (8 bit unsigned byte), `int` (32 bit signed integer), or `long` number (64 bit signed integer), respectively. Other (signed or unsigned numbers, and of different sizes) can be produced from these types with logical and arithmetic operations if necessary.

If the result specification is the symbol `C`, the result is interpreted as a 16 bit number, and a single-char transient symbol (string) is returned.

A specification of `S` tells `native` to interpret the result as a pointer to a C string (null terminated), and to return a transient symbol (string).

If the result specification is a number, it will be used as a scale to convert a returned `double` (if the number is positive) or `float` (if the number is negative) to a scaled fixpoint number.

Examples for function calls, with their corresponding C prototypes:

```
(native "lib.so" "fun" 'I)            # int fun(void);
(native "lib.so" "fun" 'N)            # long fun(void);
(native "lib.so" "fun" 'N)            # void *fun(void);
(native "lib.so" "fun" 'S)            # char *fun(void);
(native "lib.so" "fun" 1.0)           # double fun(void);
```

Arrays and Structures

If the result specification is a list, it means that the C function returned a pointer to an array, or an arbitrary memory structure. The specification list

should then consist of either the above primitive specifications (symbols or numbers), or of cons pairs of a primitive specification and a repeat count, to denote arrays of the given type.

Examples for function calls, with their corresponding pseudo C prototypes:

```
(native "lib.so" "fun" '(I . 8))       # int *fun(void);  // 8 integers
(native "lib.so" "fun" '(B . 16))      # unsigned char *fun(void);  // 16 bytes

(native "lib.so" "fun" '(I I))         # struct {int i; int j;} *fun(void);
(native "lib.so" "fun" '(I . 4))       # struct {int i[4];} *fun(void);

(native "lib.so" "fun" '(I (B . 4)))   # struct {
                                       #    int i;
                                       #    unsigned char c[4];
                                       # } *fun(void);

(native "lib.so" "fun"                 # struct {
   '(((B . 4) I) (S . 12) (N . 8)) )   #    struct {unsigned char c[4]; int i;}
                                       #    char *names[12];
                                       #    long num[8];
                                       # } *fun(void);
```

If a returned structure has an element which is a *pointer* to some other structure (i.e. not an embedded structure like in the last example above), this pointer must be first obtained with a `N` pattern, which can then be passed to `struct` for further extraction.

11.2.4 Arguments

The (optional) fourth and following arguments to `native` specify the arguments to the C function.

Primitive Types

Integer arguments (up to 64 bits, signed or unsigned `char`, `short`, `int` or `long`) can be passed as they are: As numbers.

```
(native "lib.so" "fun" NIL 123)        # void fun(int);
(native "lib.so" "fun" NIL 1 2 3)      # void fun(int, long, short);
```

String arguments can be specified as symbols. `native` allocates memory for each string (with `strdup(3)`), passes the pointer to the C function, and releases the memory (with `free(3)`) when done.

```
(native "lib.so" "fun" NIL "abc")      # void fun(char*);
(native "lib.so" "fun" NIL 3 "def")    # void fun(int, char*);
```

Note that the allocated string memory is released *after* the return value is extracted. This allows a C function to return the argument string pointer, perhaps after modifying the data in-place, and receive the new string as the return value (with the `S` specification).

```
(native "lib.so" "fun" 'S "abc")       # char *fun(char*);
```

Also note that specifying `NIL` as an argument passes an empty string (“”, which also reads as `NIL` in PicoLisp) to the C function. Physically, this is a pointer to a NULL-byte, and is not a NULL-pointer. Be sure to pass `0` (the number zero) if a NULL-pointer is desired.

Floating point arguments are specified as cons pairs, where the value is in the CAR, and the CDR holds the fixpoint scale. If the scale is positive, the number is passed as a `double`, otherwise as a `float`.

```
(native "lib.so" "fun" NIL             # void fun(double, float);
   (12.3 . 1.0) (4.56 . -1.0) )
```

Arrays and Structures

Composite arguments are specified as nested list structures. `native` allocates memory for each array or structure (with `malloc(3)`), passes the pointer to the C function, and releases the memory (with `free(3)`) when done.

This implies that such an argument can be both an input and an output value to a C function (pass by reference).

The CAR of the argument specification can be `NIL` (then it is an input-only argument). Otherwise, it should be a variable which receives the returned structure data.

The CADR of the argument specification must be a cons pair with the total size of the structure in its CAR. The CDR is ignored for input-only arguments, and should contain a *result specification* for the output value to be stored in the variable.

For example, a minimal case is a function that takes an integer reference, and stores the number '123' in that location:

```
void fun(int *i) {
   *i = 123;
}
```

We call `native` with a variable `X` in the CAR of the argument specification, a size of 4 (i.e. `sizeof(int)`), and `I` for the result specification. The stored value is then available in the variable `X`:

```
: (native "lib.so" "fun" NIL '(X (4 . I)))
-> NIL
: X
-> 123
```

The rest (CDDR) of the argument specification may contain initialization data, if the C function expects input values in the structure. It should be a list of *initialization items*, optionally with a fill-byte value in the CDR of the last cell.

If there are *no* initialization items and just the final fill-bye, then the whole buffer is filled with that byte. For example, to pass a buffer of 20 bytes, initialized to zero:

```
: (native "lib.so" "fun" NIL '(NIL (20) . 0))
```

A buffer of 20 bytes, with the first 4 bytes initialized to 1, 2, 3, and 4, and the rest filled with zero:

```
: (native "lib.so" "fun" NIL '(NIL (20) 1 2 3 4 . 0))
```

and the same, where the buffer contents are returned as a list of bytes in the variable `X`:

```
: (native "lib.so" "fun" NIL '(X (20 B . 20) 1 2 3 4 . 0))
```

For a more extensive example, let's use the following definitions:

```
typedef struct value {
   int x, y;
   double a, b, c;
   int z;
   char nm[4];
} value;

void fun(value *val) {
   printf("%d %d\n", val->x, val->y);
   val->x = 3;
   val->y = 4;
   strcpy(val->nm, "OK");
}
```

We call this function with a structure of 40 bytes, requesting the returned data in `V`, with two integers `(I . 2)`, three doubles `(100 . 3)` with a scale of 2 (1.0 = 100), another integer `I` and four characters `(C . 2)`. If the structure gets initialized with two integers 7 and 6, three doubles 0.11, 0.22 and 0.33, and another integer 5 while the rest of the 40 bytes is cleared to zero

```
: (native "lib.so" "fun" NIL
   '(V (40 (I . 2) (100 . 3) I (C . 4)) -7 -6 (100 11 22 33) -5 . 0) )
```

then it will print the integers 7 and 6, and `V` will contain the returned list

```
((3 4) (11 22 33) 5 ("O" "K" NIL NIL))
```

i.e. the original integer values 7 and 6 replaced with 3 and 4.

Note that the allocated structure memory is released *after* the return value is extracted. This allows a C function to return the argument structure pointer, perhaps after modifying the data in-place, and receive the new structure as the return value – instead of (or even in addition to) to the direct return via the argument reference.

11.3 Memory Management

The preceding *Arguments* section mentions that `native` implicitly allocates and releases memory for strings, arrays and structures.

Technically, this mimics *automatic variables* in C.

For a simple example, let's assume that we want to call `read(2)` directly, to fetch a 4-byte integer from a given file descriptor. This could be done with the following C function:

```
int read4bytes(int fd) {
   char buf[4];

   read(fd, buf, 4);
   return *(int*)buf;
}
```

`buf` is an automatic variable, allocated on the stack, which disappears when the function returns. A corresponding `native` call would be:

```
(native "@" "read" 'N Fd '(Buf (4 . I)) 4)
```

The structure argument `(Buf (4 . I))` says that a space of 4 bytes should be allocated and passed to `read`, then an integer `I` returned in the variable `Buf` (the return value of `native` itself is the number returned by `read`). The memory space is released after that.

(Note that we use `"@"` for the library here, as `read` resides in the main program.)

Instead of a single integer, we might want a list of four bytes to be returned from `native`:

```
(native "@" "read" 'N Fd '(Buf (4 B . 4)) 4)
```

The difference is that we wrote `(B . 4)` (a list of 4 bytes) instead of `I` (a single integer) for the *result specification* (see the *Arrays and Structures* section).

Let's see what happens if we extend this example. We'll write the four bytes to another file descriptor, after reading them from the first one:

```
void copy4bytes(int fd1, int fd2) {
   char buf[4];

   read(fd1, buf, 4);
   write(fd2, buf, 4);
}
```

Again, `buf` is an automatic variable. It is passed to both `read` and `write`. A direct translation would be:

```
(native "@" "read" 'N Fd '(Buf (4 B . 4)) 4)
(native "@" "write" 'N Fd (cons NIL (4) Buf) 4)
```

This work as expected. `read` returns a list of four bytes in `Buf`. The call to `cons` builds the structure

```
(NIL (4) 1 2 3 4)
```

i.e. no return variable, a four-byte memory area, filled with the four bytes (assuming that `read` returned 1, 2, 3 and 4). Then this structure is passed to `write`.

But: This solution induces quite some overhead. The four-byte buffer is allocated before the call to `read` and released after that, then allocated and released again for `write`. Also, the bytes are converted to a list to be stored in `Buf`, then that list is extended for the structure argument to `write`, and converted again back to the raw byte array. The data in the list itself are never used.

If the above operation is to be used more than once, it is better to allocate the buffer manually, use it for both reading and writing, and then release it. This also avoids all intermediate list conversions.

```
(let Buf (native "@" "malloc" 'N 4) # Allocate memory
   (native "@" "read" 'N Fd Buf 4)  # (Possibly repeat this several times)
   (native "@" "write" 'N Fd Buf 4)
   (native "@" "free" NIL Buf) )    # Release memory
```

11.3.1 Fast Fourier Transform

For a more typical example, we might call the Fast Fourier Transform using the library from the FFTW package. With the example code for calculating Complex One-Dimensional DFTs:

```
#include <fftw3.h>
...
{
   fftw_complex *in, *out;
   fftw_plan p;
   ...
   in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
   out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
   p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
   ...
   fftw_execute(p); /* repeat as needed */
   ...
   fftw_destroy_plan(p);
   fftw_free(in); fftw_free(out);
}
```

we can build the following equivalent:

```
(load "@lib/math.l")

(de FFTW_FORWARD . -1)
(de FFTW_ESTIMATE . 64)

(de fft (Lst)
   (let
      (Len (length Lst)
         In (native "libfftw3.so" "fftw_malloc" 'N (* Len 16))
         Out (native "libfftw3.so" "fftw_malloc" 'N (* Len 16))
         P (native "libfftw3.so" "fftw_plan_dft_1d" 'N
            Len In Out FFTW_FORWARD FFTW_ESTIMATE ) )
      (struct In NIL (cons 1.0 (apply append Lst)))
      (native "libfftw3.so" "fftw_execute" NIL P)
      (prog1 (struct Out (make (do Len (link (1.0 . 2)))))
         (native "libfftw3.so" "fftw_destroy_plan" NIL P)
         (native "libfftw3.so" "fftw_free" NIL Out)
         (native "libfftw3.so" "fftw_free" NIL In) ) ) )
```

This assumes that the argument list `Lst` is passed as a list of complex numbers, each as a list of two numbers for the real and imaginary part, like

```
(fft '((1.0 0) (1.0 0) (1.0 0) (1.0 0) (0 0) (0 0) (0 0) (0 0)))
```

The above translation to Lisp is quite straightforward. After the two buffers are allocated, and a plan is created, `struct` is called to store the argument list in the `In` structure as a list of double numbers (according to the `1.0` *initialization item*). Then `fftw_execute` is called, and `struct` is called again to retrieve the result from `Out` and return it from `fft` via the `prog1`. Finally, all memory is released.

11.3.2 Constant Data

If such allocated data (strings, arrays or structures passed to `native`) are constant during the lifetime of a program, it makes sense to allocate them only once, before their first use. A typical candidate is the format string of a `printf` call. Consider a function which prints a floating point number in scientific notation:

```
(load "@lib/math.l")

: (de prf (Flt)
   (native "@" "printf" NIL "%e^J" (cons Flt 1.0)) )
-> prf

: (prf (exp 12.3))
2.196960e+05
```

As we know that the format string `"%e^J"` will be converted from a Lisp symbol to a C string with `strdup` – and then thrown away – on each call to `prf`, we might as well perform a little optimization and delegate this conversion to the program load time:

```
: (de prf (Flt)
   (native "@" "printf" NIL '(native "@" "strdup" 'N "%e^J") (cons Flt 1.0)) )
-> prf

: (prf (exp 12.3))
2.196960e+05
```

If we look at the `prf` function, we see that it now contains the pointer to the allocated string memory:

```
: (pp 'prf)
(de prf (Flt)
   (native "@" "printf" NIL 24662032 (cons Flt 1000000)) )
-> prf
```

This pointer will be used by `printf` directly, without any further conversion or memory management.

11.4 Callbacks

Sometimes it is necessary to do the reverse: Call Lisp code from C code. This can be done in two ways – with certain limitations.

11.4.1 Call by Name

The first way is actually not a callback in the strict sense. It just allows to call a Lisp function with a given name.

The limitation is that this function can accept only maximally five numeric arguments, and returns a number.

The prerequisite is, of course, that you have access to the C source code. To use it from C, insert the following prototype somewhere before the first call:

```
long lisp(char*,long,long,long,long,long);
```

Then you can call `lisp` from C:

```
long n = lisp("myLispFun", a, b, 0, 0, 0);
```

The first argument should be the name of a Lisp function (built-in, or defined in Lisp). It is searched for at runtime, so it doesn't need to exist at the time the C library is compiled or loaded.

Be sure to pass dummy arguments (e.g. zero) if your function expects less than five arguments, to keep the C compiler happy.

This mechanism can generally be used for any type of argument and return value (not only `long`). On the C side, appropriate casts or a adapted prototype should be used. It is then up to the called Lisp function to prepare and/or extract the proper data with `struct` and memory management operations.

11.4.2 Function Pointer

This is a true callback mechanism. It uses the Lisp-level function `lisp` (not to confuse with the C-level function with the same name in the previous section). No C source code access is required.

`lisp` returns a function pointer, which can be passed to C functions via `native`. When this function pointer is dereferenced and called from the C code, the corresponding Lisp function is invoked. Here, too, only five numeric arguments and a numeric return value can be used, and other data types must be handled by the Lisp function with `struct` and memory management operations.

Callbacks are often used in user interface libraries, to handle key-, mouse- and other events. Examples can be found in `"@lib/openGl.l"`. The following function `mouseFunc` takes a Lisp function, installs it under the tag `mouseFunc` (any other tag would be all right too) as a callback, and passes the resulting function pointer to the OpenGL `glutMouseFunc()` function, to set it as a callback for the current window:

```
(de mouseFunc (Fun)
   (native `*GlutLib "glutMouseFunc" NIL (lisp 'mouseFunc Fun)) )
```

(The global `*GlutLib` holds the library `"/usr/lib/libglut.so"`. The backquote (`) is important here, so that the transient symbol with the library name (and not the global `*GlutLib`) is evaluated by `native`, resulting in the proper library handle at runtime).

A program using OpenGL may then use `mouseFunc` to install a function

```
(mouseFunc
   '((Btn State X Y)
      (do-something-with Btn State X Y) ) )
```

so that future clicks into the window will pass the button, state and coordinates to that function.

12

The 'select' Predicate

Alexander Burger

abu@software-lab.de

Summary. The *Pilog select/3* predicate is rather complex, and quite different from other predicates. This article tries to explain it in detail, and shows some typical use cases.

12.1 Syntax

`select` takes at least three arguments:

- A list of unification variables,
- a list of generator clauses
- and an arbitrary number of filter clauses

We will describe these arguments in the following, but demonstrate them first on a concrete example.

12.2 First Example

The examples in this document will use the demo application in "app/*.l" (see also "*A Minimal Complete Application*"). To get an interactive prompt, start it as

```
$ pil app/main.l -main +
:
```

As ever, you can terminate the interpreter by hitting `Ctrl-D`.

For a first, typical example, let's write a complete call to *solve* that returns a list of articles with numbers between 1 and 4, which contain "Part" in their description, and have a price less than 100:

```
(let (Nr (1 . 4)  Nm Part  Pr '(NIL . 100.00))
   (solve
      (quote
         @Nr Nr
         @Nm Nm
         @Pr Pr
         (select (@Item)
            ((nr +Item @Nr) (nm +Item @Nm) (pr +Item @Pr))
               (range @Nr @Item nr)
               (part @Nm @Item nm)
               (range @Pr @Item pr) ) )
      @Item ) )
```

This expression will return, with the default database setup of "app/init.l", a list of exactly one item (`{3-2}`), the item with the number 2.

The `let` statement assigns values to the search parameters for number `Nr`, description `Nm` and price `Pr`. The Pilog query (the first argument to `solve`) passes these values to the Pilog variables `@Nr`, `@Nm` and `@Pr`. Ranges of values are always specified by cons pairs, so `(1 . 4)` includes the numbers 1 through 4, while `(NIL . 100.00)` includes prices from minus infinite up to one hundred.

The list of unification variables is

```
(@Item)
```

The list of generator clauses is

```
((nr +Item @Nr) (nm +Item @Nm) (pr +Item @Pr))
```

The filter clauses are

```
(range @Nr @Item nr)
(part @Nm @Item nm)
(range @Pr @Item pr)
```

12.3 Unification Variables

As stated above, the first argument to `select` should be a list of variables. These variables communicate values (via `unify`) from the `select` environment to the enclosing Pilog environment.

The first variable in this list (`@Item` in the above example) is mandatory, it takes the direct return value of `select`. Additional optional variables may be unified by clauses in the body of `select`, and return further values.

12.4 Generator Clauses

The second argument to `select` is a list of "generator clauses". Each of these clauses specifies some kind of database B-Tree `+index`, to be traversed by `select`, step by step, where each step returns a suitable single database object. In the simplest case, they consist like here just of a relation name (e.g. `nr`), a class (e.g. `+Item`), an optional hook specifier (not in this example), and a pattern (values or ranges, e.g. (1 . 4) or "Part").

The generator clauses are the core of 'select'. In some way, they behave analog to `or/2`, as each of them generates a sequence of values. However, the generator clauses behave different, as they will not generate an exhaustive set of values upon backtracking, one after the other, where the next gets its turn when the previous one is exhausted. Instead, all clauses will generate their values quasi-parallel, with a built-in optimization so that successful clauses will be called with a higher probability. "Successful" means that the returned values successfully pass `select`'s filter clauses.

12.4.1 B-Tree Stepping

In its basic form, a generator clause is equivalent to the `db/3` predicate, stepping through a single B-Tree. The clause

```
(nr +Item @Nr)
```

generates the same values as would be produced by a stand-alone Pilog clause

```
(db nr +Item @Nr @Item)
```

as can be seen in the following two calls:

```
: (? (db nr +Item (1 . 4) @Item))
 @Item={3-1}
 @Item={3-2}
 @Item={3-3}
 @Item={3-4}
-> NIL
: (? (select (@Item) ((nr +Item (1 . 4)))))
 @Item={3-1}
 @Item={3-2}
 @Item={3-3}
 @Item={3-4}
-> NIL
```

12.4.2 Interaction of Generator Clauses

`select` is mostly useful if more than one generator clause is involved. The tree search parameters of all clauses are meant to form a logical `AND`. Only those objects should be returned, for which all search parameters (and the associated filter clauses) are valid. As soon as one of the clauses finishes stepping through its database (sub)tree, the whole call to `select` will terminate, because further values returned from other generator clauses cannot be part of the result set.

Therefore, `select` would find all results most quickly if it could simply call only the generator clause with the smallest (sub)tree. Unfortunately, this is usually not known in advance. It depends on the distribution of the data in the database, and on the search parameters to each generator clause.

Instead, `select` single-steps each generator clause in turn, in a round-robin scheme, applies the filter clauses to each generated object, and re-arranges the order of generator clauses so that the more successful clauses will be preferred. This process usually converges quickly and efficiently.

12.4.3 Combined Indexes

A generator clause can also combine several (similar) indexes into a single one. Then the clause is written actually as a list of clauses.

For example, a generator clause to search for a customer by phone number is

```
(tel +CuSu @Tel)
```

If we want to search for a customer without knowing whether a given number is a normal or a mobile phone number, then a combined generator clause searching both index trees could look like

```
((tel +CuSu @Tel  mob +CuSu @Tel))
```

The generator will first traverse all matching entries in the `+Ref` tree of the `tel` relation, and then, when these are exhausted, all matching entries in the `mob` index tree.

12.4.4 Indirect Object Associations

But generator clauses are not limited to the direct B-Tree interaction of `db/3`. They can also traverse trees of associated objects, and then follow `+Link` / `+Joint` relations, or tree relations like `+Ref` to arrive at database objects with a type suitable for return values from `select`.

To locate appropriate objects from associated objects, the generator clause can contain - in addition to the standard relation/class/pattern specification (see *Generator Clauses* above) - an arbitrary number of association specifiers. Each association specifier can be

1. A symbol. Then a `+Link` or `+Joint` will be followed, or a `+List` of those will be traversed to locate appropriate objects.
2. A list. Then this list should hold a relation and a class (and an optional hook) which specify some B-Tree `+index` to be traversed to locate appropriate objects.

In this way, a single generator clause can cause the traversal of a tree of object relations to generate the desired sequence of objects. An example can be found in "app/gui.l", in the 'choOrd' function which implements the search dialog for `+Ord` (order) objects. Orders can be searched for order number and date, customer name and city, item description and supplier name:

```
(select (@@)
   ((nr +Ord @Nr) (dat +Ord @Dat)
      (nm +CuSu @Cus (cus +Ord))
      (ort +CuSu @Ort (cus +Ord))
      (nm +Item @Item (itm +Pos) ord)
      (nm +CuSu @Sup (sup +Item) (itm +Pos) ord) )
```

While `(nr +Ord @Nr)` and `(dat +Ord @Dat)` are direct index traversals, `(nm +CuSu @Cus (cus +Ord))` iterates the `nm` (name) index of customers/suppliers `+CuSu`, and then follows the `+Ref +Link` of the `cus` relation to the orders. The same applies to the search for city names via `ort`.

The most complex example is `(nm +CuSu @Sup (sup +Item) (itm +Pos) ord)`, where the supplier name is searched in the `nm` tree of `+CuSu`, then the

`+Ref` tree `(sup +Item)` tree is followed to locate items of that supplier, then all positions for those items are found using `(itm +Pos)`, and finally the `ord +Joint` is followed to arrive at the order object(s).

12.4.5 Nested Pilog Queries

In the most general case, a generator clause can be an arbitrary Pilog query. Often this is a query to a database on a remote machine, using the `remote/2` predicate, or some other resource not accessible via database indexes, like iterating a `+List` of `+Link` or `+Joint` .

Syntactically, such a generator clause is recognized by the fact that its CAR is a Pilog variable to denote the return value.

The second argument is a list of Pilog variables to communicate values (via `unify`) from the surrounding `select` environment.

The third argument is the actual list of clauses for the nested query.

Finally, an arbitrary number of association specifiers may follow, as described in the *Indirect Object Associations* section.

We can illustrate this with a somewhat useless (but simple) example, which replaces the standard generators for item number and supplier name

```
(select (@Item)
   (
      (nr +Item @Nr)
      (nm +CuSu @Sup (sup +Item))
   )
   ...
```

with the equivalent form

```
(select (@Item)
   (
      (@A (@Nr) ((db nr +Item @Nr @A)))
      (@B (@Sup) ((db nm +CuSu @Sup @B)) (sup +Item))
   )
```

That is, a query with the `db/3` tree iteration predicate is used to generate appropriate values.

12.5 Filter Clauses

The generator clauses produce - independent from each other - lots of objects, which match the patterns of individual generator clauses, but not necessarily the desired result set of the total `select` call. Therefore, the filter clauses are needed to retain the good, and throw away the bad objects. In addition, they give feedback to the generator for optimizing its traversal priorities (as described in *Generator Clauses*).

`select` then collects all objects which passed through the filters into a unique list, to avoid duplicates which would otherwise appear, because most objects can be found by more than one generator clause.

Technically, the filters are normal Pilog clauses, which just happen to be evaluated in the context of `select`. Arbitrary Pilog predicates can be used, though there exist some predicates (e.g. `isa/2`, `same/3`, `bool/3`, `range/3`, `head/3`, `fold/3`, `part/3` or `tolr/3`) especially suited for that task.

12.5.1 A Little Report

Assume we want to know how many pieces of item #2 were sold in the year

1. Then we must find all `+Pos` (position) objects referring to that

item and at the same time belonging to orders of the year 2007 (see the class definition for `+Pos` in "app/er.l"). The number of sold pieces is then in the `cnt` property of the `+Pos` objects.

As shown in the complete `select` below, we will hold the item number in the variable `@Nr` and the date range for the year in `@Year`.

Now, all positions referred by item #2 can be found by the generator clause

```
(nr +Item @Nr (itm +Pos))
```

and all positions sold in 2007 can be found by

```
(dat +Ord @Year pos)
```

However, the combination of both generator clauses

```
(select (@Pos)
   ((nr +Item @Nr (itm +Pos)) (dat +Ord @Year pos)) )
```

will probably generate too many results, namely all positions with item the full search expression will be:

```
(?
   @Nr 2                                                  # Item number
   @Year (cons (date 2007 1 1) (date 2007 12 31))         # Date range 2007
   (select (@Pos)
      ((nr +Item @Nr (itm +Pos)) (dat +Ord @Year pos))   # Generator clauses
      (same @Nr @Pos itm nr)                              # Filter item number
      (range @Year @Pos ord dat) ) )                      # Filter order date
```

For completeness, let's calculate the total count of sold items:

```
(let Cnt 0     # Counter variable
   (pilog
      (quote
         @Nr 2
         @Year (cons (date 2007 1 1) (date 2007 12 31))
         (select (@Pos)
            ((nr +Item @Nr (itm +Pos)) (dat +Ord @Year pos))
            (same @Nr @Pos itm nr)
            (range @Year @Pos ord dat) ) )
      (inc 'Cnt (get @Pos 'cnt)) )  # Increment total count
   Cnt )  # Return count
```

12.5.2 Filter Predicates

As mentioned under *Filter Clauses*, some predicates exists mainly for `select` filtering.

Some of these predicates are of general use: `isa/2` can be used to check for a type, `same/3` checks for a definite vaue, `bool/3` looks if the value is non `NIL` These predicates are rather independent of the `+relation` type.

`range/3` checks whether a value is within a given range. This could be used with any `+relation` type, but typically it will be used for numeric (`+Number`) or time (`+Date` and `+Time`) relations.

Other predicates make only sense in the context of a certain `+relation` type:

- `head/3` is mainly intended for `(+Key +String)` or `(+Ref +String)` relations,
- `fold/3` is useful for `(+Fold +Ref +String)` relations,
- `part/3` for `(+Fold +Idx +String)` relations, and

- `tolr/3` for `(+Sn +Idx +String)` relations.

13

Using 'edit'

Alexander Burger

abu@software-lab.de

Summary. This articles is about browsing the database or arbitrary data structures and definitions with the powerful `edit` function.

13.1 Introduction

A quite powerful - but little known - function in PicoLisp is 'edit'. It allows you to edit any Lisp symbol (intern, transient or extern), by pretty-printing its value and properties to a temporary file, calling an external editor, and 'read'ing back the changes when done. For the external editor, currently 'vim' is supported. The clou: You can "click" on any other symbol somewhere embedded in the nested structures of the value or property list, to have it added to the editor screen, and thus browse through potentially the whole system. This works transparently not only for internal symbols, but also for transient (which are normally not directly accessible) and external (database) symbols. In the case of external symbols, it doesn't even matter whether these are objects in a local database, or whether they reside on remote machines in a distributed system (except that remote objects cannot be modified).

13.2 PicoLisp Symbols

A symbol in PicoLisp consists of three - possibly empty - components: A value, a property list and a name. Of those, usually only the value and the properties are modified during programming. The value is nothing more than a special property, used implicitly for prominent purposes like variable bindings or function definitions. There exist a large number of functions to access, set or modify a symbol's value or property list. Setting the value and a few properties of the symbol X:

```
: (setq X "Hello")
-> "Hello"

: (put 'X 'a 1)
-> 1

: (with 'X
   (=: b 2)
   (push (:: lst) "OK" '(a b c d) 17) )
-> 17
```

These data can be accessed individually

```
: X
-> "Hello"

: (val 'X)
-> "Hello"

: (get 'X 'lst)
-> (17 (a b c d) "OK")
```

or looked at as a whole, using the function 'show':

```
: (show 'X)
X "Hello"
   lst (17 (a b c d) "OK")
   b 2
   a 1
-> X
```

show displays the symbol's name (here "X"), then the value (here "Hello") on the same line, followed by all properties, each on its own line indented by three spaces.

13.3 Editing a Symbol

Instead of just showing the symbol to the console, you can use edit to get a similar display in a 'vim' session:

```
: (edit 'X)
```

The editor's window will appear as:

```
X "Hello"
   a 1
   b 2
   lst (17 (a b c d) "OK")

(********)
```

The difference to a plain show is that you can change the value or properties. The pattern (********) is used by edit internally as a delimiter, and should not be modified. For example, change the e in the value "Hello" to a, and the a in the lst property to x:

```
X "Hallo"
   a 1
   b 2
   lst (17 (x b c d) "OK")

(********)
```

then exit 'vim' in the normal way, e.g. with ":x". On your console you see

```
: (edit 'X)
# X redefined
# X lst redefined
-> NIL
```

You can use show or other commands to see that X was indeed changed

```
: X
-> "Hallo"

: (get 'X 'lst)
-> (17 (x b c d) "OK")
```

It is even possible to start with an empty - or "nonexistent" - symbol

```
: (edit 'Y)
```

It displays, as expected

```
Y NIL

(********)
```

Now add a value and some properties

```
Y (This is a value)
   bar (and this is a property)
   foo 17

(********)
```

After exiting the editor:

```
: (show 'Y)
Y (This is a value)
   foo 17
   bar (and this is a property)
-> Y
```

13.4 Browsing

Now let's try the "browsing" capability mentioned above. When edit starts up the 'vim' editor, it defines two key mappings for that edit session:

- Once you edit a symbol, you can move the cursor to the first character of some other symbol appearing in the value or properties, and press an upper-case 'K'. This will cause that symbol to be added to the edit session, separated by another (********)
- Pressing an upper-case 'Q' goes one step back to the previous view

We can try this while editing X. Moving the cursor to the x in the lst property, and hitting 'K' gives:

```
x NIL

(********)

X "Hallo"
   a 1
   b 2
   lst (17 (x b c d) "OK")

(********)
```

Now we see both x and X being edited. Unfortunately, x is not very interesting here, as it has only the default value of NIL and no properties. The same effect can be achieved by calling

```
(edit 'x 'X)
```

You can pass any number of symbols to edit. A little more happens if we move down to lst again, and hit 'K' on the symbol d:

```
d (NIL (let *Dbg NIL (dbg ^)))
   *Dbg ((216 . "/usr/lib/picolisp/lib/debug.l"))

(********)

x NIL

(********)

X "Hallo"
   a 1
   b 2
   lst (17 (x b c d) "OK")

(********)
```

Indeed, now we found something! This is not surprising, though, as 'd' has a definition in the debugger context. The value is the function

```
(() (let *Dbg NIL (dbg ^)))
```

and the '*Dbg' property contains the file and line number of its source.

13.5 Transient Symbols

We can use edit to inspect itself.

```
: (edit 'edit)
```

The result looks meager

```
edit (@
   (let *Dbg NIL
      (setq "*F" (tmp '"edit.l"))
      (catch NIL ("edit" (rest))) ) )
   *Dbg ((6 . "lib/edit.l"))

(********)
```

because - as can be seen in the fourth line - edit is a short function wich calls "edit" (defined in a transient symbol) to do the actual work. The transient symbol "edit" is not directly reachable. In the REPL

```
: (pp '"edit")
(de "edit" . "edit")
-> "edit"
```

we see just the string "edit". But if we edit edit, place the cursor on the first double quote character of "edit" in line four, and press 'K', we get

```
"edit" (("Lst")
   (let "N" 1
      (loop
         (out "*F"
            (setq
               "*Lst" (make
                  (for "S" "Lst"
                     ("loc" (printsp "S"))
                     ("loc" (val "S"))
...

(********)

edit (@
   (let *Dbg NIL
      (setq "*F" (tmp '"edit.l"))
      (catch NIL ("edit" (rest))) ) )
   *Dbg ((6 . "lib/edit.l"))

(********)
```

BTW, you can see another transient function in line 8: "loc". You may click on that one to see its definition. In contrast, if you look at the 'locale' function

```
(edit 'locale)
```

you'll find in there another, completely different, "loc" function. This is an example for the locality of transient symbols. The two "loc"'s have nothing to do with each other, and don't conflict in their definitions, yet you can see - and possibly change - them both (separately) in the editor.

13.6 Browsing the Database

For the following examples we use the demo application in the PicoLisp distribution. Start it as described in the Getting Started section:

```
$ ln -s /usr/share/picolisp/app
$ pil app/main.l -main -go +
```

Then connect with a browser to 'http://localhost:8080' to get a PicoLisp REPL prompt in your terminal window. Log in as "admin" / "admin" in the

browser GUI. Now you can navigate through the whole database. Start at an arbitrary object. For a first overview, the '*DB' root object is just fine.

```
(edit *DB)
```

You see the external symbol {1}, pointing to the base objects of the entity classes.

```
{1} NIL
   +Role {3}
   +User {7}
   +Sal {16}
   +CuSu {31}
   +Item {32}
   +Ord {33}
   +Pos {34}

(********)
```

The first one, {3}, is the base of the +Role entity. Move to the opening brace and press 'K'.

```
{3} NIL
   nm (3 . {D1})

(********)

{1} NIL
   +Role {3}
   +User {7}
   +Sal {16}
   +CuSu {31}
   +Item {32}
   +Ord {33}
   +Pos {34}

(********)
```

We see that {3} contains only a single index, the nm (name) property of roles. The number 3 tells us that this index tree has three nodes, and its root node is {D1}. If we inspect that index root node, by clicking on {D1}

```
{D1} (NIL ("Accounting" NIL . {4}) ("Administration" NIL . {2}) ("Assistance" NIL . {5}))
...
```

The first role in that list is "Accounting", the object {4}. {4} in turn leads us to

```
{4} (+Role)
   nm "Accounting"
   usr ({12} {11} {10})
   perm (Customer Item Order Report Delete)
...
```

We see an object of class +Role (as expected), with the name "Accounting", the users in the usr list, and a list of permissions. Again, we might click on the first user, {12}

```
{12} (+User)
   role {4}  # (+Role)
   nam "Sandra Bullock"
   nm "sandy"
   pw "sandy"
...
```

The role of that user points back to {4}, as we have a +Joint - a bi-directional relation. We might verify this, by exiting edit with ":q" and call (vi '+User) to inspect the sources

```
...
### Role ###
(class +Role +Entity)

(rel nm (+Need +Key +String))          # Role name
(rel perm (+List +Symbol))             # Permission list
(rel usr (+List +Joint) role (+User))  # Associated users


### User ###
(class +User +Entity)

(rel nm (+Need +Key +String))          # User name
(rel pw (+String))                     # Password
(rel role (+Joint) usr (+Role))        # User role
...
```

showing that role of +User points to a +Role object, and the usr property of +Role has a list of +User objects. A similar information can also be obtained directly from the runtime system. Go back to the user {4} again

```
(edit '{4})
```

then click on the first character of `+Role` (i.e. the '+' character) in the classes list of {4}.

```
+Role ((url> (Tab) (and (may RoleAdmin) (list "app/role.l" '*ID This)))
   +Entity )
   nm $53165764545663  # (+Need +Key +String)
   perm $53165764545716  # (+List +Symbol)
   usr $53165764545754  # (+List +Joint)
   Dbf (1 . 512)
   *Dbg ((39 . "lib/adm.l")
      (url> 26 . "app/er.l")
      (usr 43 . "lib/adm.l")
      (perm 42 . "lib/adm.l")
      (nm 41 . "lib/adm.l") )
...
```

Note that now we are no longer in a database object, but in the class definition. It shows that +Role defines a single method url¿, is a subclass of +Entity, and has relations nm, perm and usr. The property Dbf used for database maintenance, and *Dbg holds debug information. You may experiment more. You can click on the '$' of a relation maintenance daemon object, and even on a commented symbol like +List or +Joint.

13.7 Debugging

edit comes in handy also during debugging. You can easily do on-the-fly changes to a function, like inserting a call to print a 'msg', or setting some explicit breakpoint with '!', without actually touching the source code. To edit a certain object in a large database, it is often easier to find it by going to that object in the GUI. In the demo app, click on the "Orders" menu item to the left, then on the '@' link in the leftmost column of the first order. You should get a form with that order. Now, in the REPL, you can access the form that is currently shown in the browser via the *Top global variable. You may look at it with (show *Top), or edit it with (edit *Top). You get an awful lot of data, mostly for the GUI components in that form. As before, you can click

on any of them to see what they contain. Scrolling down a bit, there is an obj property. This is the database object held by that form.

```
...
evt 0
obj {B7}  # (+Ord)
gui ($53165764713535
   $53165764713635
   $53165764713747
...
```

Here, it is {B7}. Again, you can click on that,

```
{B7} (+Ord)
   nr 1
   dat 733027  # 2007-02-14
   cus {C3}  # (+CuSu)
   pos ({A1} {A2} {A3})
...
```

and again you are "in" the database. You can follow the links to the customer (the +CuSu object {C3}), or the three positions in that order pos. Let's pick the first position {A1}

```
{A1} (+Pos)
   cnt 1
   pr 29900
   itm {B1}  # (+Item)
   ord {B7}  # (+Ord)
...
```

and see a link to the item {B1}, and back to the order {B7}. The item {B1} leads us to

```
{B1} (+Item)
   nr 1
   inv 100
   pr 29900
   sup {C1}  # (+CuSu)
   nm "Main Part"
   ...
```

in turn pointing to sup, the item's supplier {C1}, and so on. The database objects can be modified here in any conceivable way, but you should be very sure about what you do, if you don't want an inconsistent database. Relations involving index trees or "joint"ed objects need corresponding changes in other objects, and are better avoided. In any case, a change to a DB object will only be manifest if you enter (commit) after exiting from the editor.

13.8 Distributed Database

Though the demo app doesn't really make use of remote objects, it contains a hook to experiment with them. If the demo application was started as above, it automatically also listens on port 4040 for remote requests. A distributed database requires some setup and administration. We don't go into the details here, but a simple setup can be made by starting (in addition to the app server above) a stand-alone PicoLisp interpreter in another terminal window

```
$ pil +
```

and initialize the *Ext variable as described in 'remote/2'

```
(setq *Ext
   (mapcar
      '((@Host @Ext)
         (cons @Ext
            (curry (@Host @Ext (Sock)) (Obj)
               (when (or Sock (setq Sock (connect @Host 4040)))
                  (ext @Ext
                     (out Sock (pr (cons 'qsym Obj)))
                     (prog1 (in Sock (rd))
                        (unless @
                           (close Sock)
                           (off Sock) ) ) ) ) ) ) )
      '("localhost")
      '(20) ) )
```

to let the system know where where to fetch remote objects from. If you started the remote server on another machine (you didn't forget to open port 4040 in the firewall, did you?), supply its name or IP address instead of "localhost". Then request the order with the number 1, and edit it:

```
: (let Sock (connect "localhost" 4040)
   (ext 20
      (out Sock (pr '(pr (db 'nr '+Ord 1))))
      (prog1 (in Sock (rd)) (close Sock)) ) )
-> {AF7}

: (edit @)
```

From here on, continue as with the local database. Just that the (now remote) order object {B7} appears locally as {AF7}.

```
{AF7} (+Ord)
   nr 1
   dat 733027  # 2007-02-14
   cus {AG3}  # (+CuSu)
   pos ({AE1} {AE2} {AE3})
```

The same holds for the customer and the positions. Clicking on the first position {AE1} gives

```
{AE1} (+Pos)
   cnt 1
   pr 29900
   itm {AF1}  # (+Item)
   ord {AF7}  # (+Ord)
...
```

Except for the fact that the names of all external symbols appear with an offset, everything else behaves like in the local case.

27oct11 abu

14

Bash Completion

Alexander Burger

abu@software-lab.de

Summary. This article describes how Bash completion works in PicoLisp.

14.1 Bash Completion

Since picoLisp-3.0.9 there is support for Bash completion.

While it is not precisely the absolute killer-feature, bash completion is quite handy when developing PicoLisp applications from the shell command line.

If you installed PicoLisp locally (i.e. not from a distribution package), you need to copy two files

```
$ cp lib/complete.l /usr/lib/picolisp/lib/
$ cp lib/bash_completion /etc/bash_completion.d/pil
```

As ever, source `. /etc/bash_completion` in your .bashrc

Per default, if you hit the TAB key during command line input, Bash completes things it knows about, like commands and path names.

PicoLisp – in addition to normal path/file name arguments – accepts two particular types of arguments:

1. If the argument's first character is '-', then the rest of that argument is taken as a Lisp function call (without the surrounding parentheses).
2. If the argument's first character is '@', then it is interpreted as a path into the interpreter's installation directory.

For (1), the expansion actually searches all built-in function names of the given invocation. For example, entering

```
$ pil -ver
```

and then hitting TAB will expand to "-version".

The expansion also honors single or double quotes, to allow for function arguments:

```
$ pil -'pri
```

or

```
$ pil -"pri
```

This expands to the printing functions.

For (2), the intended path name is properly expanded. This works regardless of whether it is a global or a local installation, as it always searches the invoked interpreter's environment.

```
$ pil @lib/xh
```

and

```
$ <somePath>/pil @lib/xh
```

both will expand to "@lib/xhtml.l".

As an extra goody, an empty argument expands to '+' (the trailing debug flag – perhaps the most often needed command line argument).

15

The Need for Speed

Alexander Burger

abu@software-lab.de

Summary. This article discusses why relative speed of a language implementation is overrated in comparison to features like expressiveness, flexibility and orthogonality.

15.1 Introduction

One of the greatest mysteries in the history of computer language comparisons is to me the question why most people are more interested in the relative speed of a language implementation, rather than in features like expressiveness, flexibility and orthogonality.

Several years ago I wrote an article[1] where - among other things - I compared PicoLisp with a "compiled" Lisp (CLisp). After that, postings appeared who claimed that picking CLisp was an unfortunate choice, because it compiles only to bytecode, and that SBCL would have been better.

Contrary to the intention of those postings, I see this quite as an assertion of my argument. After all, why go through the troubles and disadvantages of supporting a compiler, when the resulting speed is *lower* than without?

And while I still believe that raw execution speed is a relatively unimportant issue, I feel I should supply an update.

I did a local install of SBCL on a Linux x86-64 System with two Dual-Core 1-GHz Opterons, and compared that with the current 64-bit version of PicoLisp.

[1] http://software-lab.de/radical.pdf

15.2 Fibonacci

In the above article, the fibonacci function was called with an argument of 30. As today's machines are faster, I took 40 instead, and got:

```
(fibo 40)
   PicoLisp    34.8 sec
   sbcl         5.1 sec
   sbcl(i)    33:45 min
```

(sbcl(i) means "SBCL interpreted" – More than half an hour is beyond good and evil, of course)

The relation of SBCL (5.1 sec) to PicoLisp (34.8 sec) looks reasonable. Fibonacci on compiled SBCL runs about 6.8 times faster than on (interpreted) PicoLisp. And can probably be even improved with some declaration magic. PicoLisp, on the other hand, is not designed for arithmetic speed, it is always handicapped by its bignum-only number type.

But, as I also wrote in the above article, integer primitive operations can be easily optimized by a compiler. They are, however, not typical for a Lisp program, where direct list mappings are used instead of array index calculations.

BTW, out of interest I also tried an equivalent Python program. It took 1:45 minutes. In general I can say that on most occasions where I compared PicoLisp to Python I observed such a factor of 1 to 3.

15.3 List Operations

So I tried the second example from that article, the `tst` function doing extensive list operations.

```
(load "tst.l")
   PicoLisp     2.0 sec
   sbcl         1.8 sec
   sbcl(i)     72.8 sec
```

The difference is negligible. Not much to say here.

15.4 Binary Trees

Some people claimed the above examples are not "real" benchmarks. Let's move to the Alioth Benchmarks Game platform, where the Binary Trees[2] benchmark does things quite similar to the above `tst` (though also a certain amount of arithmetics). The SBCL version is

```
(defun build-btree (item depth)
  (declare (fixnum item depth))
  (if (zerop depth) (list item)
      (let ((item2 (+ item item))
            (depth-1 (1- depth)))
        (declare (fixnum item2 depth-1))
        (cons item
            (cons (build-btree (the fixnum (1- item2)) depth-1)
                  (build-btree item2 depth-1))))))

(defun check-node (node)
  (declare (values fixnum))
  (let ((data (car node))
        (kids (cdr node)))
    (declare (fixnum data))
    (if kids
        (- (+ data (check-node (car kids)))
           (check-node (cdr kids)))
        data)))

(defun loop-depths (max-depth &key (min-depth 4))
  (declare (type fixnum max-depth min-depth))
  (loop for d of-type fixnum from min-depth by 2 upto max-depth do
       (loop with iterations of-type fixnum = (ash 1 (+ max-depth min-depth (- d)))
          for i of-type fixnum from 1 upto iterations
          sum (+ (the fixnum (check-node (build-btree i d)))
                 (the fixnum (check-node (build-btree (- i) d))))
          into result of-type fixnum
          finally
            (format t "~D   trees of depth ~D    check: ~D~%"
                    (the fixnum (+ iterations iterations )) d result))))
```

[2] http://shootout.alioth.debian.org/u64q/performance.php?test=binarytrees

```
(defun main (&optional (n (parse-integer
                           (or (car (last #+sbcl sb-ext:*posix-argv*
                                          #+cmu  extensions:*command-line-strings*
                                          #+gcl  si::*command-args*))
                               "1"))))
  (declare (type (integer 0 255) n))
  (format t "stretch tree of depth ~D   check: ~D~%" (1+ n) (check-node (build-btree 0 (1+ n))))
  (let ((*print-pretty* nil) (long-lived-tree (build-btree 0 n)))
    (purify)
    (loop-depths n)
    (format t "long lived tree of depth ~D    check: ~D~%" n (check-node long-lived-tree))))
```

The corresponding PicoLisp program is

```
(de buildTree (Item Depth)
   (cons Item
      (and
         (n0 Depth)
         (cons
            (buildTree
               (dec (setq Item (>> -1 Item)))
               (dec 'Depth) )
            (buildTree Item Depth) ) ) ) )

(de checkNode (Node)
   (if2 (cadr Node) (cddr Node)
      (- (+ (car Node) (checkNode (cadr Node))) (checkNode @))
      (+ (car Node) (checkNode @))
      (- (car Node) (checkNode @))
      (car Node) ) )
```

```
(let (N (format (opt))  Min 4)
   (prinl
      "stretch tree of depth "
      (inc N)
      "^I check: "
      (checkNode (buildTree 0 (inc N))) )
   (let LongLivedTree (buildTree 0 N)
      (for (D Min (>= N D) (+ 2 D))
         (let (Sum 0  Iterations (>> (- D Min N) 1))
            (for I Iterations
               (inc 'Sum
                  (+
                     (checkNode (buildTree I D))
                     (checkNode (buildTree (- I) D)) ) ) )
            (prinl
               (* 2 Iterations)
               "^I trees of depth "
               D
               "^I check: "
               Sum ) ) )
      (prinl
         "long lived tree of depth "
         N
         "^I check: "
         (checkNode LongLivedTree) ) ) )
```

When called with an argument of 20, we get

```
PicoLisp    4:03 min
sbcl        1:02 min
```

If we optimize the PicoLisp version, by calling (gc 100) at the beginning, the time is reduced to three and a half minutes, but this seems be forbidden by the benchmark rule.

In any case, here a factor of 4 is also not really overwhelming.

15.5 Fannkuch

Finally, I looked at the Alioth Fannkuch[3] benchmark. The SBCL version is

[3] http://shootout.alioth.debian.org/u64q/performance.php?test=fannkuch

```
(defun write-permutation (perm)
  (loop for i across perm do
   (princ (1+ i)))
  (terpri))

(defun fannkuch (n)
  (declare (optimize (speed 3) (safety 0) (debug 0)) (fixnum n))
  (assert (< 1 n 128))
  (let ((perm (make-array n :element-type 'fixnum))
        (perm1 (make-array n :element-type 'fixnum))
        (count (make-array n :element-type 'fixnum))
        (flips 0) (flipsmax 0) (r n) (check 0) (k 0)
   (i 0) (perm0 0))

    (declare ((simple-array fixnum (*)) perm perm1 count)
             (fixnum flips flipsmax check k r i perm0))

    (dotimes (i n) (setf (aref perm1 i) i))

    (loop

     (when (< check 30)
       (write-permutation perm1)
       (incf check))

     (loop while (> r 1) do
           (setf (aref count (1- r)) r)
           (decf r))

     (unless (or (= (aref perm1 0) 0)
       (= (aref perm1 (1- n)) (1- n)))
       (setf flips 0)
       (dotimes (i n) (setf (aref perm i) (aref perm1 i)))
       (setf k (aref perm1 0))
       (loop while (/= k 0) do
             (loop for j fixnum downfrom (1- k)
                   for i fixnum from 1
                   while (< i j) do (rotatef (aref perm i) (aref perm j)))
             (incf flips)
             (rotatef k (aref perm k)))
       (setf flipsmax (max flipsmax flips)))

     (loop do
      (when (= r n)
        (return-from fannkuch flipsmax))
      (setf i 0 perm0 (aref perm1 0))
      (loop while (< i r) do
       (setf k (1+ i)
             (aref perm1 i) (aref perm1 k)
             i k))
      (setf (aref perm1 r) perm0)
      (when (> (decf (aref count r)) 0) (loop-finish))
      (incf r)))))
```

```
(defun main ()
  (let ((n (parse-integer (second *posix-argv*))))
    (format t "Pfannkuchen(~D) = ~D~%" n (fannkuch n))))
```

Wow, what a piece! Compare that to the equivalent PicoLisp program:

```
(let (N (format (opt))  Lst (range N 1)  L Lst  M)
   (recur (L)  # Permute
      (if (cdr L)
         (do (length L)
            (recurse (cdr L))
            (rot L) )
         (let I 0  # For each permutation
            (and (ge0 (dec (30))) (prinl (reverse Lst)))
            (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
               (inc 'I) )
            (setq M (max I M)) ) ) )
   (prinl "Pfannkuchen(" N ") = " M) )
```

But at last we can find some significance:

```
(fannkuch 10)
   PicoLisp     6.4 sec
   sbcl         1.0 sec
   sbcl(i)     > 30 min   (aborted)

(fannkuch 11)
   PicoLisp    71.1 sec
   sbcl         5.0 sec
```

We see a factor of 14.2.

But at what a price! I'm not only talking about the discussed disadvantages of the compiler per se, but of that mess of code. I would not want to write my production code in such a style, and always prefer simplicity and succinctness over a bureaucratic monster.

If we remove the (declare (optimize ..)) statement, the execution time of SBCL doubles - from 5.0 to 10.0 seconds - and the factor goes down to 7.1.

BTW, the speed advantage is melting down if we use this parallized PicoLisp version (using the later[4] function):

[4] http://software-lab.de/doc/refL.html#later

```
(let (N (format (opt))  Lst (range N 1)  L Lst)
   (let (Res (need N)  M)
      (for (R Res R (cdr R))
         (later R
            (let L (cdr Lst)
               (recur (L)  # Permute
                  (if (cdr L)
                     (do (length L)
                        (recurse (cdr L))
                        (rot L) )
                     (let I 0  # For each permutation
                        (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
                           (inc 'I) )
                        (setq M (max I M)) ) ) )
               M ) )
         (rot Lst) )
      (wait NIL (full Res))
      (prinl "Pfannkuchen(" N ") = " (apply max Res)) ) )
```

Then we get on the above 4-core machine

```
(fannkuch 10)
   PicoLisp     1.9 sec
   sbcl         1.0 sec

(fannkuch 11)
   PicoLisp    18.4 sec
   sbcl         5.0 sec
```

Up to `N` this scales almost linearly with the number of cores. With an 8-core machine it would well outperform SBCL.

Note: The printing of the first 30 results - as required by the Alioth benchmark - was omitted here, because their order is unpredictable for parallel execution and thus would not match the Alioth byte-by-byte comparison. A conformant solution (it shows no measurable timing difference) would be:

```
(let (N (format (opt))  Lst (range N 1)  L Lst)
   (catch NIL
      (recur (L)  # Print the first 30 permutations
         (cond
            ((cdr L)
               (do (length L)
                  (recurse (cdr L))
                  (rot L) ) )
            ((ge0 (dec (30)))
               (prinl (reverse Lst)) )
            (T (throw)) ) ) )
   (let (Res (need N)  M)
      (for (R Res R (cdr R))
         (later R
            (let L (cdr Lst)
               (recur (L)  # Permute
                  (if (cdr L)
                     (do (length L)
                        (recurse (cdr L))
                        (rot L) )
                     (let I 0  # For each permutation
                        (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
                           (inc 'I) )
                        (setq M (max I M)) ) ) )
               M ) )
         (rot Lst) )
      (wait NIL (full Res))
      (prinl "Pfannkuchen(" N ") = " (apply max Res)) ) )
```

16

GUI Scripting

Alexander Burger

abu@software-lab.de

Summary. This article explains how to programmatically control the standard PicoLisp GUI.

16.1 Introduction

The standard PicoLisp GUI (in "@lib/xhtml.l" and "@lib/form.l") can be completely driven under program control. A set of functions in "@lib/scrape.l" allows you to write scripts for automated

- unit-tests
- database queries and updates
- stress-tests
- debugging

This is possible because **all** GUI functionality is available via plain HTML components and HTTP transactions. Though there exist some extensions and enhancements in JavaScript, which run in parallel to speed up certain user interactions or provide additional conveniences like drag and drop, they are never mission-critical, nor an exclusive way to do the job.

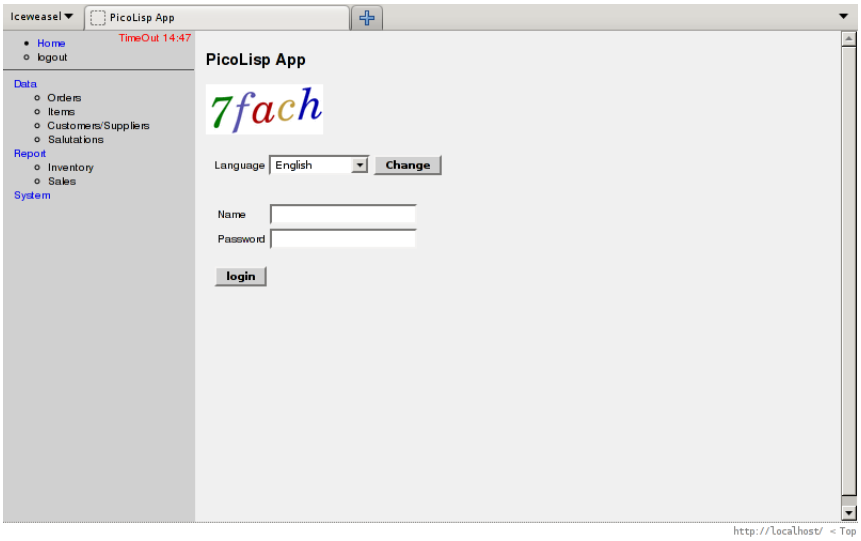
16.2 A Simple Example

Let's use the demo application[1] that comes with the PicoLisp distribution. You can start it locally, and access it as http://localhost:8080[2] (recommended), or – if this is not an option – try the online demo at http://app.7fach.de[3].

The demo application is a simplified ERP system, containing things like customers, articles and orders. For the following example, we want to find the price of an article with the name "Spare Part".

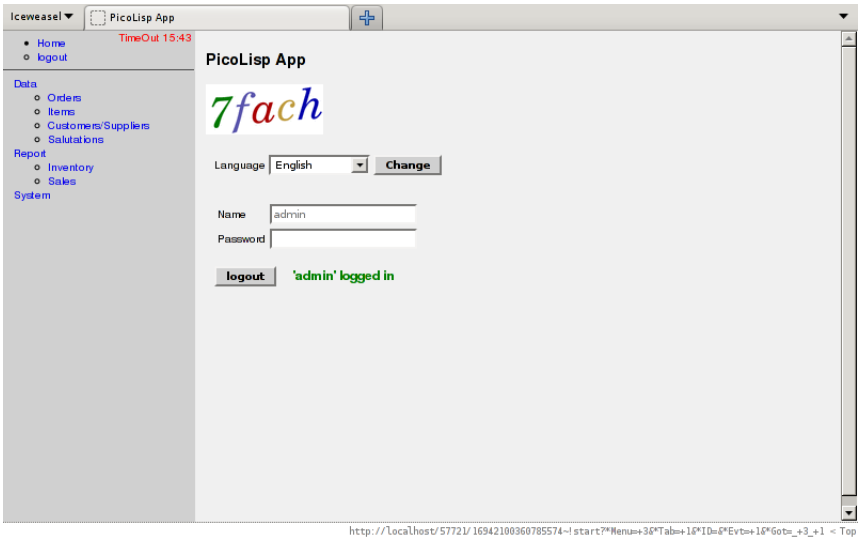
16.2.1 Using the Browser GUI

Normally, you would connect to the server with a browser,

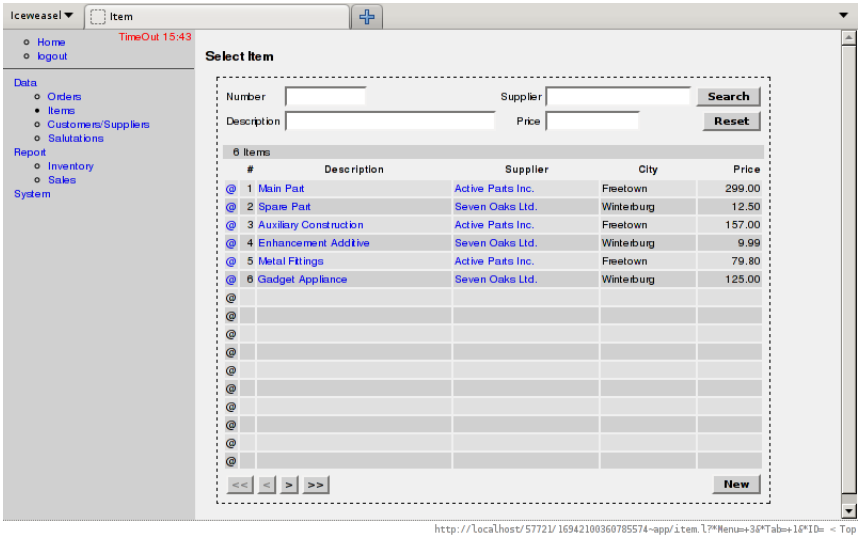


log in on the first page (by typing user name and password, and pressing the "login" button),

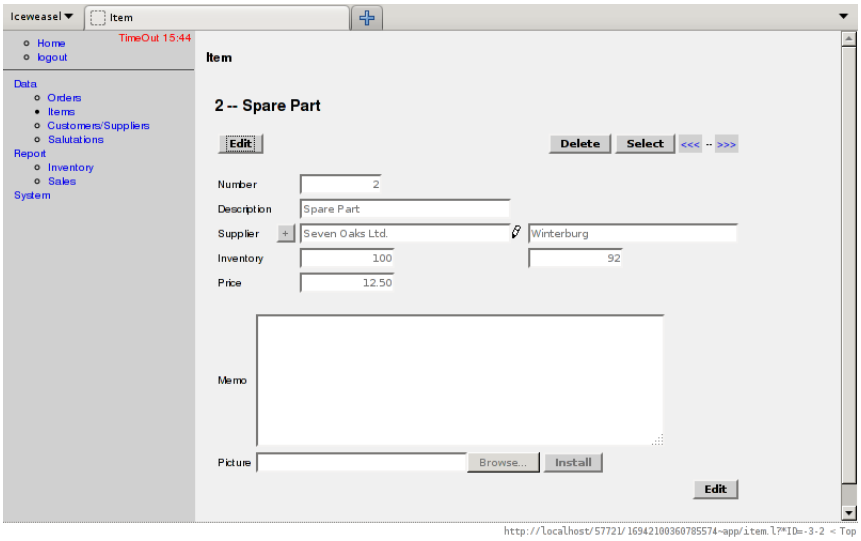
[1] http://software-lab.de/doc/app.html#minApp
[2] http://localhost:8080
[3] http://app.7fach.de



then search for that article in the "Items" search dialog (after clicking on the "Items" link in the "Data" submenu),



and read the price on the page of that item (here: 12.50).



16.2.2 Using GUI Scripting

The same result can be obtained without a browser, by interacting on the REPL with the remote application. We assume that the demo application is still running locally at http://localhost:8080[4].

Start PicoLisp, and load the necessary libraries:

```
$ pil +
: (load "@lib/http.l" "@lib/scrape.l")
```

Call `scrape` to connect to the server

```
: (scrape "localhost" 8080)
-> "PicoLisp App"
```

If you must use the online demo server, call `(scrape "7fach.de" 80 "8080")` instead, telling `scrape` to connect to port 80 of that server (where 'httpGate' is running), and then to dispatch to 8080 (the application's address).

Now log in:

[4] http://localhost:8080

```
: (expect "'admin' logged in"
   (enter 3 "admin")
   (enter 4 "admin")
   (press "login") )
-> NIL
```

(**Caveat**: Keep in mind a fundamental feature of the PicoLisp server: Whenever a second log in of the same user from the same IP address is detected, the first session of that user is automatically terminated. This means that you cannot run a GUI and a `scrape` session with user "admin" at the same time. If you already had a gui session open, it will be *dead* now.)

`expect` takes a string argument – a pattern which is to be expected on the resulting page after the body (the remaining arguments) was executed. This body enters two values (the user name and the password) into the appropriate fields, and presses the "login" button.

`enter` takes a field specification and a value. The first call with `3` refers to the "Name" field, as this is the third field on the form (the first field is the "TimeOut" indicator, and the second one is the language selector). The next field, field number `4`, is the password field.

Finally, `press` simulates a press of the corresponding button, and the message "'admin' logged in" appears on the page, satisfying `expect`.

Then, just as in the GUI, click on the "Items" link to open the dialog. We forgo the `expect` here, and assume that the dialog can't fail:

```
: (click "Items")
-> "Item"
```

We also assume that "Spare Part" readily appears in the result list of the search dialog (it was on the second line). A real script would interact with the dialog to find the desired item. So we just click on it:

```
: (click "Spare Part")
-> "Item"
```

Now the page of that article is open. Counting the fields, we find that the price is in the 8th one. We can read that field's value, and get the same result as in the GUI session above:

```
: (value 8)
-> "12.50"
```

Note that you can also get an overview of the current page with `display`:

```
: (display)
###############
click "Home" "logout" "Data" "Orders" "Items" "Customers/Suppliers"
"Salutations" "Report" "Inventory" "Sales" "System" "<<<" ">>>" "obj"

press "Edit" "Delete" "Select" "+" "Install" "Edit"

value "TimeOut 12:11" "2" "Spare Part" "Seven Oaks UnLtd." "Winterburg" "100"
"98" "12.50"

-> "Item"
```

This helps you to find the positions of links, buttons and fields.

Finally, we should log out:

```
: (click "logout")
-> "PicoLisp App"
```

16.3 The Scrape Library

To use the GUI scripting functionality, you need to load "@lib/http.l" and "@lib/scrape.l". It implements the following functions:

- `(scrape 'host 'port 'how) -> sym | lst` Sets up an environment for further operations on an application, which can be `connect`ed on a server at `host` and `port`, and an optional URL argument `how`. Returns the title of the page if no errors occurred, otherwise a list of error messages.
- `(expect 'sym . prg)` Executes a `prg` body (holding calls to the functions explained below), and checks for an expected pattern `sym` in the server output.
- `(click 'label ['cnt]) -> sym | lst` Simulates the click on a link. The first (or `cnt`'th) link found on the page which has `label` as a prefix is used. Returns the result of `scrape`.
- `(press 'label ['cnt]) -> sym | lst` Simulates the press of a button. The first (or `cnt`'th) button found on the page which has `label` as a prefix is used. Returns the result of `scrape`.
- `(value 'field ['cnt]) -> sym` Returns the current value of the GUI field. The `field` argument may be either a positive number (to specify

the `cnt`'th field on the page), a negative number (to specify the `-cnt`'th last field on the page), or the field's name. If the name is given, then `cnt` may specify the form if more than one form is on the page.

- `(enter 'field 'sym ['cnt]) -> sym` Enters a value `sym` into the GUI field. The `field` argument may be either a positive number (to specify the `cnt`'th field on the page), a negative number (to specify the `-cnt`'th last field on the page), or the field's name. If the name is given, then `cnt` may specify the form if more than one form is on the page.
- `(display) -> sym` Display the state of the current page. First a line `###############` is printed as a visual separator, then all available links and buttons, and the values of all GUI fields is printed.

All these functions can be used interactively (e.g. during development and debugging of a script), and in stand-alone programs. In the latter case, they can be used as building-blocks of higher-level functions, to interact flexibly with forms, dialogs and alerts.

Let your imagination fly!

17

Manual Page

Alexander Burger

abu@software-lab.de

Summary. This is the PicoLisp Manual Page.

17.1 NAME

pil, picolisp - a fast, lightweight Lisp interpreter

17.2 SYNOPSIS

pil [arguments ...] [-] [arguments ...] [+] **/installpath/bin/picolisp** [arguments ...] [-] [arguments ...] [+]

17.3 DESCRIPTION

PicoLisp is a Lisp interpreter with a small memory footprint, yet relatively high execution speed. It combines an elegant and powerful language with built-in database functionality.

pil is the startup front-end for the interpreter. It takes care of starting the binary base system and loading a useful runtime environment.

picolisp is just the bare interpreter binary. It is usually called in stand-alone scripts, using the she-bang notation in the first line, passing the minimal environment in *lib.l* and loading additional files as needed:

```
(load ``@ext.l'' ``myfiles/lib.l'' ``myfiles/foo.l'')

(do \ldots{} something \ldots{})

(bye)
```

17.4 INVOCATION

PicoLisp has no pre-defined command line flags; applications are free to define their own. Any built-in or user-level Lisp function can be invoked from the command line by prefixing it with a hyphen. Examples for built-in functions useful in this context are **version** (print the version number) or **bye** (exit the interpreter). Therefore, a minimal call to print the version number and then immediately exit the interpreter would be:

```
$ pil -version -bye
```

Any other argument (not starting with a hyphen) should be the name of a file to be loaded. If the first character of a path or file name is an at-mark, it will be substituted with the path to the installation directory.

All arguments are evaluated from left to right, then an interactive *read-eval-print* loop is entered (with a colon as prompt).

A single hyphen stops the evaluation of the rest of the command line, so that the remaining arguments may be processed under program control.

If the very last command line argument is a single plus character, debugging mode is switched on at interpreter startup, before evaluating any of the command line arguments. A minimal interactive session is started with:

```
$ pil +
```

Here you can access the reference manual

```
(doc)
```

and the online documentation for most functions,

```
(doc 'vi)
```

or directly inspect their sources:

```
(vi 'doc)
```

The interpreter can be terminated with

```
(bye)
```

or by typing Ctrl-D.

17.5 FILES

Runtime files are maintained in the ~/.pil directory:

```
~{}/.pil/tmp/<pid>/
```

Process-local temporary directories

```
~{}/.pil/history
```

The line editor's history file

17.6 BUGS

PicoLisp doesn't try to protect you from every possible programming error ("You asked for it, you got it").

17.7 AUTHOR

Alexander Burger abu@software-lab.de

17.8 RESOURCES

Home page: http://home.picolisp.com

Download: http://www.software-lab.de/down.html

18

README

Alexander Burger

abu@software-lab.de

Summary. This is the README file of the PicoLisp distribution.

18.1 The PicoLisp System

```
_PI_co Lisp is not _CO_mmon Lisp
```

PicoLisp can be viewed from two different aspects: As a general purpose programming language, and a dedicated application server framework.

18.1.1 Programming Language

As a programming language, PicoLisp provides a 1-to-1 mapping of a clean and powerful Lisp derivate, to a simple and efficient virtual machine. It supports persistent objects as a first class data type, resulting in a database system of Entity/Relation classes and a Prolog-like query language tightly integrated into the system.

The virtual machine was designed to be

Simple The internal data structure should be as simple as possible. Only one single data structure is used to build all higher level constructs.

Unlimited There are no limits imposed upon the language due to limitations of the virtual machine architecture. That is, there is no upper bound in symbol name length, number digit counts, or data structure and buffer sizes, except for the total memory size of the host machine.

Dynamic Behavior should be as dynamic as possible ("run"-time vs. "compile"-time). All decisions are delayed till runtime where possible. This involves matters like memory management, dynamic symbol binding, and late method binding.

Practical PicoLisp is not just a toy of theoretical value. PicoLisp is used since 1988 in actual application development, research and production.

The language inherits the major advantages of classical Lisp systems like

- Dynamic data types and structures
- Formal equivalence of code and data
- Functional programming style
- An interactive environment

PicoLisp is very different from any other Lisp dialect. This is partly due to the above design principles, and partly due to its long development history since 1984.

You can download the latest release version at http://software-lab.de/down.html

18.1.2 Application Server Framework

As an application server framework, PicoLisp provides for

NoSQL Database Management

- Index trees
- Object local indexes
- Entity/Relation classes
- Pilog (PicoLisp Prolog) queries
- Multi-user synchronization
- DB Garbage collection
- Journaling, Replication

User Interface

- Browser GUI
- X)HTML/CSS
- XMLHttpRequest/JavaScript

Application Server

- Process management
- Process family communication
- XML I/O
- Import/export
- User administration
- Internationalization
- Security
- Object linkage
- Postscript/Printing

PicoLisp is not an IDE. All program development in Software Lab. is done using the console, bash, vim and the Lisp interpreter.

The only type of GUI supported for applications is through a browser via HTML. This makes the client side completely platform independent. The GUI is created dynamically. Though it uses JavaScript and XMLHttpRequest for speed improvements, it is fully functional also without JavaScript or CSS.

The GUI is deeply integrated with - and generated dynamically from - the application's data model. Because the application logic runs on the server, multiple users can view and modify the same database object without conflicts, everyone seeing changes done by other users on her screen immediately due to the internal process and database synchronization.

PicoLisp is free software, and you are welcome to use and redistribute it under the conditions of the MIT/X11 License (see "COPYING").

It compiles and runs on current 32-bit GNU/Linux, FreeBSD, Mac OS X (Darwin), Cygwin/Win32 (and possibly other) systems. A native 64-bit version is available for x86-64/Linux, x86-64/SunOS and ppc64/Linux.

19

INSTALL

Alexander Burger

abu@software-lab.de

Summary. This is the INSTALL file from the PicoLisp distribution.

19.1 PicoLisp Installation

There is no 'configure' procedure, but the PicoLisp file structure is simple enough to get along without it (we hope). It should compile and run on GNU/Linux, FreeBSD, Mac OS X (Darwin), Cygwin/Win32, and possibly other systems without problems.

PicoLisp supports two installation strategies: Local and Global.

The default (if you just download, unpack and compile the release) is a local installation. It will not interfere in any way with the world outside its directory. There is no need to touch any system locations, and you don't have to be root to install it. Many different versions - or local modifications - of PicoLisp can co-exist on a single machine.

For a global installation, allowing system-wide access to the executable and library/documentation files, you can either install it from a ready-made distribution, or set some symbolic links to one of the local installation directories as described below.

Note that you are still free to have local installations along with a global installation, and invoke them explicitly as desired.

19.2 Local Installation

19.2.1 Unpack the distribution

```
$ tar xfz picoLisp-XXX.tgz
```

19.2.2 Change the directory

```
$ cd picoLisp-XXX
```

19.2.3 Compile the PicoLisp interpreter

```
$ (cd src; make)
```

Or - if you have an x86-64 system (under Linux or SunOS), or a ppc64 system (under Linux) - build the 64-bit version

```
$ (cd src64; make)
```

In both cases the executable bin/picolisp will be created.

To build the 64-bit version the first time (bootstrapping), you have the following three options:

1. If a Java runtime system (version 1.6 or higher) is installed, it will build right out of the box.
2. Otherwise, download one of the pre-generated "*.s" file packages
   - http://software-lab.de/x86-64.linux.tgz
   - http://software-lab.de/x86-64.sunOs.tgz
   - http://software-lab.de/ppc64.linux.tgz
3. Else, build a 32-bit version first, and use the resulting bin/picolisp to generate the "*.s" files:

```
$ (cd src; make)
$ (cd src64; make x86-64.linux)
```

After that, the 64-bit binary can be used to rebuild itself.

Note that on the BSD family of operating systems, 'gmake' must be used instead of 'make'.

19.3 Global Installation

The recommended way for a global installation is to use a picolisp package from the OS distribution.

If that is not available, you can (as root) create four symbolic links from /usr/lib, /usr/share and /usr/bin to a local installation directory

```
# ln -s /<installdir> /usr/lib/picolisp
# ln -s /<installdir> /usr/share/picolisp
# ln -s /usr/lib/picolisp/bin/picolisp /usr/bin/picolisp
# ln -s /usr/lib/picolisp/bin/pil /usr/bin/pil
```

19.4 Invocation

In a global installation, the 'pil' command should be used. You can either start in plain or in debug mode. The difference is that for debug mode the command is followed by single plus ('+') sign. The '+' must be the very last argument on the command line.

```
$ pil       # Plain mode
:

$ pil +     # Debug mode
:
```

In both cases, the colon ':' is PicoLisp's prompt. You may enter some Lisp expression,

```
: (+ 1 2 3)
  -> 6
```

To exit the interpreter, enter

```
: (bye)
```

or just type Ctrl-D.

For a local invocation, specify a path name, e.g.

```
$ ./pil     # Plain mode
:

$ ./pil +   # Debug mode
:
```

or

```
$ /home/app/pil  # Invoking a local installation from some other directory
```

A shortcut for debug mode is the 'dbg' script:

```
$ ./dbg
:
```

It is available only for local installaions, and is eqivalent to

```
$ ./pil +
```

Note that 'pil' can also serve as a template for your own stand-alone scripts.

If you just want to test the ready-to-run Ersatz PicoLisp (it needs a Java runtime system), use

```
$ ersatz/pil +
:
```

instead of './dbg' or './pil +'.

19.5 Documentation

For further information, please look at "doc/index.html". There you find the PicoLisp Reference Manual ("doc/ref.html"), the PicoLisp tutorials ("doc/tut.html", "doc/app.html", "doc/select.html" and "doc/native.html"), and the frequently asked questions ("doc/faq.html").

For details about the 64-bit version, refer to "doc64/README", "doc64/asm" and "doc64/structures".

As always, the most accurate and complete documentation is the source code ;-) Included in the distribution are many utilities and pet projects, including tests, demo databases and servers, games (chess, minesweeper), 3D animation (flight simulator), and more.

Any feedback is welcome! Hope you enjoy :-)

Part III

PicoLisp Tutorials

20

A PicoLisp Tutorial

Alexander Burger

abu@software-lab.de

Summary. This article demonstrates some aspects of the PicoLisp system in detail and example. For a general description of the PicoLisp kernel please look at the *PicoLisp Reference*.

This is not a Lisp tutorial, as it assumes some basic knowledge of programming, Lisp, and even PicoLisp. Please read these sections in the *PicoLisp Reference* before coming back here: *Introduction* and *The PicoLisp Machine*. This tutorial concentrates on the specificities of PicoLisp, and its differences with other Lisp dialects.

20.1 Now let's start

If not stated otherwise, all examples assume that PicoLisp was started from a global installation (see Installation) from the shell prompt as

```
$ pil +
:
```

It loads the PicoLisp base system and the debugging environment, and waits for you to enter input lines at the interpreter prompt (`:`). You can terminate the interpreter and return to the shell at any time, by either hitting the `Ctrl-D` key, or by executing the function `(bye)`.

Please note that special handling is done during character input. This one is incompatible with `rlwrap` for example but is more powerful.

- `vi`-like command-line editing (typos fixes and history with ESC, `h`, `j`, `k` and `l` but not arrows),
- auto-formating (underlined) of double-quoted strings (don't try and struggle to make `"` appear).

If you prefer to use Emacs, please use the picolisp-mode bundled in "@lib/el".

If you feel that you absolutely have to use an IDE, `rlwrap` or another input front-end, please remove the entry "@lib/led.l" from "lib.l" and "dbg.l". Note that in this case, however, you will not have the TAB symbol completion feature available during command line editing.

20.2 Command Line Editing

PicoLisp permanently reads input from the current input channel (i.e. the console in interactive mode), evaluates it, and prints the result to the current output channel. This is called a "read-eval-print-loop" (REPL).

20.2.1 VI-like editing

It is very helpful - though not absolutely necessary - when you know how to use the `vi` editor.

To alleviate the task of manual line input, a command line editor is provided which is similar to (though much simpler than) the `readline` feature of the `bash` shell. Only a subset of the `vi` mode is supported, which is restricted to single-key commands (the "real" `vi` supports multi-key commands and the modification of most commands with count prefixes). It is loaded at startup in debug mode, you find its source in "lib/led.l".

You can enter lines in the normal way, correcting mistypes with the BACKSPACE key, and terminating them with the ENTER key. This is the *Insert Mode*.

If you hit ESC, you get into *Command Mode*. Now you can navigate horizontally in the current input line, or vertically in the history of previously entered lines, with key commands borrowed from the `vi` editor (only `h`, `j`, `k` and `l` and not arrows). Note, however, that there is always only a single line visible.

Let's say you did some calculation

```
: (* (+ 2 3) (- 7 2))
-> 25
:
```

If you want to repeat a modified version of this command, using `8` instead of `7`, you don't have to re-type the whole command, but type

- ESC to get into *Command Mode*
- `k` to get one line "up"

- `f` and `7` to "find" the character `7`
- `r` and `8` to "replace" with `8`

Then you hit ENTER to execute the modified line. Instead of jumping to the `7` with the "find" command, you may also type `l` (move "right") repeatedly till you reach the correct position.

The key commands in the *Command Mode* are listed below. Some commands change the mode back to *Insert Mode* as indicated in parentheses. Deleting or changing a "word" take either the current atom (number or symbol), or a whole expression when the cursor is at a left parenthesis.

- `k` - Go up one line
- `j` - Go down one line
- `l` - Go right one character
- `h` - Go left one character
- `w` - Go right one word
- `b` - Go back (left) one word
- `0` - Go to the beginning of the line
- `$` - Go to the end of the line
- `i` - Enter *Insert Mode* at the cursor position
- `a` - Append (*Insert Mode*) after the cursor position
- `A` - Append (*Insert Mode*) at the end of the line
- `I` - Insert (*Insert Mode*) at the beginning of the line
- `x` - Delete the character at the cursor position
- `X` - Delete the character left of the cursor position
- `r` - Replace the character at the cursor position with the next key
- `s` - Substitute the character at the cursor position (*Insert Mode*)
- `S` - Substitute the whole line (*Insert Mode*)
- `d` - Delete the word at the cursor position (*Insert Mode*)
- `D` - Delete the rest of the line
- `c` - Change the word at the cursor position (*Insert Mode*)
- `C` - Change the rest of the line (*Insert Mode*)
- `f` - Find next key in the rest of the current line
- `p` - Paste data deleted with `x`, `X`, `d` or `D` after the cursor position

- `P` - Paste data deleted with `x`, `X`, `d` or `D` before the cursor position
- `/` - Accept an input pattern and search the history for it
- `n` - Search for next occurrence of pattern (as entered with `/`)
- `N` - Search for previous occurrence of pattern
- `%` - Go to matching parenthesis
- `~` - Convert character to opposite (lower or upper) case and move right
- `u` - Undo the last change (one level only)
- `U` - Undo all changes of the current line
- `g` - Display current contents of cut buffer (not in `vi`)

Notes:

- The `d` command corresponds to the `dw` command of the `vi` editor, and `c` corresponds to `cw`.
- Search patterns may contain ' `@` ' characters as wildcards.
- Lines shorter than 3 characters, lines beginning with a space character, or duplicate lines are not entered into the history.
- The history is stored in the file ".pil/history" in the user's home directory. The length of the history is limited to 1000 lines.

The following two key-combinations work both in Insert and Command Mode:

- `Ctrl-D` will immediately terminate the current process.
- `Ctrl-X` discards all input, abandons further processing, and returns to the interpreter's top level (equivalent to invoking `quit`). This is also useful when the program stopped at a breakpoint (see single-stepping *Debugging*), or after program execution was interrupted with `Ctrl-C`.

Besides these two keys, in *Insert Mode* only the following keys have a special meaning:

- BACKSPACE (`Ctrl-H`) and DEL erase the character to the left
- `Ctrl-V` inserts the next key literally
- TAB performs symbol and/or path completion: When a symbol (or path) name is entered partially and TAB is pressed subsequently, all internal symbols (and/or path names in the file system) matching the partial input are shown in sequence.
- ESC terminates *Input Mode* and enters *Command Mode*

20.2.2 Conclusion

Please take some time to experiment and to get used to command line editing. It will make life much easier in the future :-)

20.3 Browsing

PicoLisp provides some functionality for inspecting pieces of data and code within the running system.

20.3.1 Basic tools

The really basic tools are of course available and their name alone is enough to know: `print`, `size` ...

But you will appreciate some more powerful tools like:

- `match`, a predicate which compares S-expressions with bindable wildcards when matching,

20.3.2 Inspect a symbol with *show*

The most commonly used tool is probably the `show` function. It takes a symbolic argument, and shows the symbol's name (if any), followed by its value cell, and then the contents of the property list on the following lines (assignment of such things to a symbol can be done with `set`, `setq`, and `put`).

```
: (setq A '(This is the value))  # Set the value cell of 'A'
-> (This is the value)
: (put 'A 'key1 'val1)           # Store property 'key1'
-> val1
: (put 'A 'key2 'val2)           # and 'key2'
-> val2
: (show 'A)                      # Now 'show' the symbol 'A'
A (This is the value)
   key2 val2
   key1 val1
-> A
```

`show` accepts an arbitrary number of arguments which are processed according to the rules of `get`, resulting in a symbol which is showed then.

```
: (put 'B 'a 'A)        # Put 'A' under the 'a'-property of 'B'
-> A
: (setq Lst '(A B C))   # Create a list with 'B' as second argument
-> (A B C)
: (show Lst 2 'a)       # Show the property 'a of the 2nd element of 'Lst'
A (This is the value)   # (which is 'A' again)
   key2 val2
   key1 val1
-> A
```

20.3.3 Inspect and edit with *edit*

Similar to `show` is `edit`. It takes an arbitrary number of symbolic arguments, writes them to a temporary file in a format similar to `show`, and starts the `vim` editor with that file.

```
: (edit 'A 'B)
```

The `vim` window will look like

```
A (This is the value)
key1 val1
key2 val2

(********)

B NIL
a A  # (This is the value)

(********)
```

Now you can modify values or properties. You should not touch the parenthesized asterisks, as they serve as delimiters. If you position the cursor on the first character of a symbol name and type `K` ("Keyword lookup"), the editor will be restarted with that symbol added to the editor window. `Q` (for "Quit") will bring you back to the previous view.

`edit` is also very useful to browse in a database. You can follow the links between objects with `K` , and even - e.g. for low-level repairs

- modify the data (but only if you are really sure about what you are

doing, and don't forget to `commit` when you are done).

20.3.4 Built-in pretty print with *pp*

The *pretty-print* function `pp` takes a symbol that has a function defined (or two symbols that specify message and class for a method definition), and displays that definition in a formatted and indented way.

```
: (pp 'pretty)
(de pretty (X N . @)
   (setq N (abs (space (or N 0))))
   (while (args) (printsp (next)))
   (if (or (atom X) (>= 12 (size X)))
      (print X)
      (while (== 'quote (car X))
         (prin "'")
         (pop 'X) )
      (let Z X
         (prin "(")
         (cond
            ((memq (print (pop 'X)) *PP)
               (cond
                  ((memq (car Z) *PP1)
                     (if (and (pair (car X)) (pair (cdar X)))
                        (when (>= 12 (size (car X)))
                           (space)
                           (print (pop 'X)) )
                        (space)
                        (print (pop 'X))
                        (when
                           (or
                              (atom (car X))
                              (>= 12 (size (car X))) )
                           (space)
                           (print (pop 'X)) ) ) )
                  ((memq (car Z) *PP2)
                     (inc 'N 3)
                     (loop
                        (prinl)
                        (pretty (cadr X) N (car X))
                        (NIL (setq X (cddr X)) (space)) ) )
```

```
                  ((or (atom (car X)) (>= 12 (size (car X))))
                     (space)
                     (print (pop 'X)) ) ) )
               ((and (memq (car Z) *PP3) (>= 12 (size (head 2 X))))
                  (space)
                  (print (pop 'X) (pop 'X)) ) )
            (when X
               (loop
                  (T (== Z X) (prin " ."))
                  (T (atom X) (prin " . ") (print X))
                  (prinl)
                  (pretty (pop 'X) (+ 3 N))
                  (NIL X) )
               (space) )
            (prin ")") ) ) )
-> pretty
```

The style is the same as we use in source files:

- The indentation level is three spaces
- If a list is too long (to be precise: if its `size` is greater than 12), pretty-print the CAR on the current line, and each element of the CDR recursively on its own line.
- A closing parenthesis a preceded by a space if the corresponding open parenthesis is not on the same line

20.3.5 Inspect elements one by one with *more*

`more` is a simple tool that displays the elements of a list one by one. It stops after each element and waits for input. If you just hit ENTER, `more` continues with the next element, otherwise (usually I type a dot (`.`) followed by ENTER) it terminates.

```
: (more (1 2 3 4 5 6))
1                              # Hit ENTER
2.                             # Hit '.' and ENTER
-> T                           # stopped
```

Optionally `more` takes a function as a second argument and applies that function to each element (instead of the default `print`). Here, often `show` or `pp` (see below) is used.

```
: (more '(A B))            # Step through 'A' and 'B'
A
B
-> NIL
: (more '(A B) show)       # Step through 'A' and 'B' with 'show'
A (This is the value)      # showing 'A'
   key2 val2
   key1 val1
                           # Hit ENTER
B NIL                      # showing 'B'
   a A
-> NIL
```

20.3.6 Search through available symbols with *what*

The `what` function returns a list of all internal symbols in the system which match a given pattern (with `@` wildcard characters).

```
: (what "prin@")
-> (prin print prinl print> printsp println)
```

20.3.7 Search through values or properties of symbols with *who*

The function `can` returns a list which indicates which classes *can* accept a given message. Again, this list is suitable for iteration with `pp`:

```
: (can 'del>)                                  # Which classes accept 'del>' ?
-> ((del> . +List) (del> . +Entity) (del> . +relation))

: (more (can 'del>) pp)                        # Inspect the methods with 'pp'
(dm (del> . +List) (Obj Old Val)
   (and ((<> Old Val) (delete Val Old)) )

(dm (del> . +Entity) (Var Val)
   (when
      (and
         Val
         (has> (meta This Var) Val (get This Var)) )
```

```
      (let Old (get This Var)
         (rel>
            (meta This Var)
            This
            Old
            (put This Var (del> (meta This Var) This Old @)) )
         (when (asoq Var (meta This 'Aux))
            (relAux This Var Old (cdr @)) )
         (upd> This Var Old) ) ) )

(dm (del> . +relation) (Obj Old Val)
   (and ((<> Old Val) Val) )
```

20.3.8 Inspect dependencies with *dep*

`dep` shows the dependencies in a class hierarchy. That is, for a given class it displays the tree of its (super)class(es) above it, and the tree of its subclasses below it.

To view the complete hierarchy of input fields, we start with the root class `+relation`:

```
: (dep '+relation)
+relation
   +Bag
   +Any
   +Blob
   +Link
      +Joint
   +Bool
   +Symbol
      +String
   +Number
      +Time
      +Date
-> +relation
```

If we are interested in `+Link`:

```
: (dep '+Link)
   +relation
+Link
   +Joint
-> +Link
```

This says that `+Link` is a subclass of `+relation`, and has a single subclass (`+Joint`).

20.4 Defining Functions

Most of the time during programming is spent defining functions (or methods). In the following we will concentrate on functions, but most will be true for methods as well except for using `dm` instead of `de`.

20.4.1 Functions with no argument

The notorious "Hello world" function must be defined:

```
: (de hello ()
   (prinl "Hello world") )
-> hello
```

The `()` in the first line indicates a function without arguments. The body of the function is in the second line, consisting of a single statement. The last line is the return value of `de`, which here is the defined symbol. From now on we will omit the return values of examples when they are unimportant.

Now you can call this function this way:

```
: (hello)
Hello world
```

20.4.2 Functions with one argument

A function with an argument might be defined this way:

```
: (de hello (X)
   (prinl "Hello " X) )
# hello redefined
-> hello
```

PicoLisp informs you that you have just redefined the function. This might be a useful warning in case you forgot that a bound symbol with that name already existed.

```
: (hello "world")
Hello world


: (hello "Alex")
Hello Alex
```

20.4.3 Preventing arguments evaluation and variable number of arguments

Normally, PicoLisp evaluates the arguments before it passes them to a function:

```
: (hello (+ 1 2 3))
Hello 6
```

```
: (setq A 1  B 2)       # Set 'A' to 1 and 'B' to 2
-> 2
: (de foo (X Y)         # 'foo' returns the list of its arguments
   (list X Y) )
-> foo
: (foo A B)             # Now call 'foo' with 'A' and 'B'
-> (1 2)                # -> We get a list of 1 and 2, the values of 'A' and 'B'
```

In some cases you don't want that. For some functions (`setq` for example) it is better if the function gets all arguments unevaluated, and can decide for itself what to do with them.

For such cases you do not define the function with a *list* of parameters, but give it a *single atomic* parameter instead. PicoLisp will then bind all (unevaluated) arguments as a list to that parameter.

```
: (de foo X
   (list (car X) (cadr X)) )        # 'foo' lists the first two arguments

: (foo A B)                         # Now call it again
-> (A B)                            # -> We don't get '(1 2)', but '(A B)'

: (de foo X
   (list (car X) (eval (cadr X))) ) # Now evaluate only the second argument

: (foo A B)
-> (A 2)                            # -> We get '(A 2)'
```

20.4.4 Mixing evaluated arguments and variable number of unevaluated

arguments

As a logical consequence, you can combine these principles. To define a function with 2 evaluated and an arbitrary number of unevaluated arguments:

```
: (de foo (X Y . Z)     # Evaluate only the first two args
   (list X Y Z) )

: (foo A B C D E)
-> (1 2 (C D E))        # -> Get the value of 'A' and 'B' and the remaining list
```

20.4.5 Variable number of evaluated arguments

More common, in fact, is the case where you want to pass an arbitrary number of *evaluated* arguments to a function. For that, PicoLisp recognizes the symbol `@` as a single atomic parameter and remembers all evaluated arguments in an internal frame. This frame can then be accessed sequentially with the `args`, `next`, `arg` and `rest` functions.

```
: (de foo @
   (list (next) (next)) )     # Get the first two arguments

: (foo A B)
-> (1 2)
```

Again, this can be combined:

```
: (de foo (X Y . @)
   (list X Y (next) (next)) ) # 'X' and 'Y' are fixed arguments

: (foo A B (+ 3 4) (* 3 4))
-> (1 2 7 12)                 # All arguments are evaluated
```

These examples are not very useful, because the advantage of a variable number of arguments is not used. A function that prints all its evaluated numeric arguments, each on a line followed by its squared value:

```
: (de foo @
   (while (args)                              # Check if there are some args left
      (println (next) (* (arg) (arg))) ) )  # Call the last arg (next) returned

: (foo (+ 2 3) (- 7 1) 1234 (* 9 9))
5 25
6 36
1234 1522756
81 6561
-> 6561
```

This next example shows the behaviour of `args` and `rest`:

```
: (de foo @
   (while (args)
      (next)
      (println (arg) (args) (rest)) ) )
: (foo 1 2 3)
1 T (2 3)
2 T (3)
3 NIL NIL
```

Finally, it is possible to pass all these evaluated arguments to another function, using `pass`:

```
: (de foo @
   (pass println 9 8 7)       # First print all arguments preceded by 9, 8, 7
   (pass + 9 8 7) )           # Then add all these values

: (foo (+ 2 3) (- 7 1) 1234 (* 9 9))
9 8 7 5 6 1234 81             # Printing ...
-> 1350                       # Return the result
```

20.4.6 Anonymous functions without the *lambda* keyword

There's no distinction between code and data in PicoLisp, `quote` will do what you want (see also *this FAQ entry*).

```
: ((quote (X) (* X X)) 9)
-> 81
```

```
: (setq f '((X) (* X X)))  # This is equivalent to (de f (X) (* X X))
-> ((X) (* X X))
: f
-> ((X) (* X X))
: (f 3)
-> 9
```

20.5 Debugging

There are two major ways to debug functions (and methods) at runtime: *Tracing* and *single-stepping*.

In this section we will use the REPL to explore the debugging facilities, but in the *Scripting* section, you will learn how to launch PicoLisp scripts with some selected functions debugged:

```
$ pil app/file1.l -"trace 'foo" -main -"debug 'bar" app/file2.l +
```

20.5.1 Tracing

Tracing means letting functions of interest print their name and arguments when they are entered, and their name again and the return value when they are exited.

For demonstration, let's define the unavoidable factorial function (or just `load` the file "`@doc/fun.l`"):

```
(de fact (N)
   (if (=0 N)
      1
      (* N (fact (dec N))) ) )
```

With `trace` we can put it in trace mode:

```
: (trace 'fact)
-> fact
```

Calling `fact` now will display its execution trace.

```
: (fact 3)
 fact : 3
  fact : 2
   fact : 1
    fact : 0
    fact = 1
   fact = 1
  fact = 2
 fact = 6
-> 6
```

As can be seen here, each level of function call will indent by an additional space. Upon function entry, the name is separated from the arguments with a colon (`:`), and upon function exit with an equals sign `=` from the return value.

`trace` works by modifying the function body, so generally it works only for functions defined as lists (lambda expressions, see *Evaluation*). Tracing a C-function is possible, however, when it is a function that evaluates all its arguments.

So let's trace the functions `=0` and `*`:

```
: (trace '=0)
-> =0
: (trace '*)
-> *
```

If we call `fact` again, we see the additional output:

```
: (fact 3)
 fact : 3
  =0 : 3
  =0 = NIL
  fact : 2
   =0 : 2
   =0 = NIL
   fact : 1
    =0 : 1
    =0 = NIL
    fact : 0
     =0 : 0
     =0 = 0
    fact = 1
    * : 1 1
    * = 1
   fact = 1
   * : 2 1
   * = 2
  fact = 2
  * : 3 2
  * = 6
 fact = 6
-> 6
```

To reset a function to its untraced state, call `untrace`:

```
: (untrace 'fact)
-> fact
: (untrace '=0)
-> =0
: (untrace '*)
-> *
```

or simply use `mapc`:

```
: (mapc untrace '(fact =0 *))
-> *
```

20.5.2 Single-stepping

Single-stepping means to execute a function step by step, giving the programmer an opportunity to look more closely at what is happening. The function

`debug` inserts a breakpoint into each top-level expression of a function. When the function is called, it stops at each breakpoint, displays the expression it is about to execute next (this expression is also stored into the global variable `^`) and enters a read-eval-loop. The programmer can then

- inspect the current environment by typing variable names or calling functions
- execute `(d)` to recursively debug the next expression (looping through subexpressions of this expression)
- execute `(e)` to evaluate the next expression, to see what will happen without actually advancing on
- type ENTER (that is, enter an empty line) to leave the read-eval loop and continue with the next expression

Thus, in the simplest case, single-stepping consists of just hitting ENTER repeatedly to step through the function.

To try it out, let's look at the `stamp` system function. You may need to have a look at

- `=T` (T test),
- `date` and `time` (grab system date and time)
- `default` (conditional assignments)
- `pack` (kind of concatenation), and
- `dat$` and `tim$` (date and time formats)

to understand this definition.

```
: (pp 'stamp)
(de stamp (Dat Tim)
   (and (=T Dat) (setq Dat (date T)))
   (default Dat (date) Tim (time T))
   (pack (dat$ Dat "-") " " (tim$ Tim T)) )
-> stamp
```

```
: (debug 'stamp)                        # Debug it
-> T
: (stamp)                               # Call it again
(and (=T Dat) (setq Dat (date T)))      # stopped at first expression
!                                       # ENTER
(default Dat (date) Tim (time T))       # second expression
!                                       # ENTER
(pack (dat$ Dat "-") " " (tim$ ...      # third expression
! Tim                                  # inspect 'Tim' variable
-> 41908
! (time Tim)                           # convert it
-> (11 38 28)
!                                       # ENTER
-> "2004-10-29 11:38:28"                # done, as there are only 3 expressions
```

Now we execute it again, but this time we want to look at what's happening inside the second expression.

```
: (stamp)                               # Call it again
(and (=T Dat) (setq Dat (date T)))
!                                       # ENTER
(default Dat (date) Tim (time T))
!                                       # ENTER
(pack (dat$ Dat "-") " " (tim$ ...      # here we want to look closer
! (d)                                   # debug this expression
-> T
!                                       # ENTER
(dat$ Dat "-")                          # stopped at first subexpression
! (e)                                   # evaluate it
-> "2004-10-29"
!                                       # ENTER
(tim$ Tim T)                            # stopped at second subexpression
! (e)                                   # evaluate it
-> "11:40:44"
!                                       # ENTER
-> "2004-10-29 11:40:44"                # done
```

The breakpoints still remain in the function body. We can see them when we pretty-print it:

```
: (pp 'stamp)
(de stamp (Dat Tim)
   (! and (=T Dat) (setq Dat (date T)))
   (! default Dat (date) Tim (time T))
   (! pack
      (! dat$ Dat "-")
      " "
      (! tim$ Tim T) ) )
-> stamp
```

To reset the function to its normal state, call `unbug`:

```
: (unbug 'stamp)
```

Often, you will not want to single-step a whole function. Just use `edit` (see above) to insert a single breakpoint (the exclamation mark followed by a space) as CAR of an expression, and run your program. Execution will then stop there as described above; you can inspect the environment and continue execution with ENTER when you are done.

20.6 Functional I/O

Input and output in PicoLisp is functional, in the sense that there are not variables assigned to file descriptors, which need then to be passed to I/O functions for reading, writing and closing. Instead, these functions operate on implicit input and output channels, which are created and maintained as dynamic environments.

Standard input and standard output are the default channels. Try reading a single expression:

```
: (read)
(a b c)        # Console input
-> (a b c)
```

To read from a file, we redirect the input with `in`. Note that comments and whitespace are automatically skipped by `read`:

```
: (in "@doc/fun.l" (read))
-> (de fact (N) (if (=0 N) 1 (* N (fact (dec N)))))
```

The `skip` function can also be used directly. To get the first non-white character in the file with `char`:

```
: (in "@doc/fun.l" (skip "#") (char))
-> "("
```

`from` searches through the input stream for given patterns. Typically, this is not done with Lisp source files (there are better ways), but for a simple example let's extract all items immediately following `fact` in the file,

```
: (in "@doc/fun.l" (while (from "fact ") (println (read))))
(N)
(dec N)
```

or the word following "(de" with `till`:

```
: (in "@doc/fun.l" (from "(de ") (till " " T)))
-> "fact"
```

With `line`, a line of characters is read, either into a single *transient* symbol (the type used by PicoLisp for strings),

```
: (in "@doc/tut.html" (line T))
-> "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://..."
```

or into a list of symbols (characters):

```
: (in "@doc/tut.html" (line))
-> ("<" "!" "D" "O" "C" "T" "Y" "P" "E" " " "H" "T" "M" "L" ...
```

`line` is typically used to read tabular data from a file. Additional arguments can split the line into fixed-width fields, as described in the `reference manual`. If, however, the data are of variable width, delimited by some special character, the `split` function can be used to extract the fields. A typical way to import the contents of such a file is:

```
(load "@lib/import.l")

(in '("bin/utf2" "importFile.txt")              # Pipe: Convert to UTF-8
   (until (eof)                                 # Process whole file
      (let L (split (line) "^I")                # TAB-delimited data
         ... use 'getStr', 'getNum' etc ...     # process them
```

Some more examples with `echo`:

```
(in "a"                                    # Copy the first 40 Bytes
   (out "b"                                # from file "a" to file "b"
      (echo 40) ) )

(in "@doc/tut.html"                        # Show the HTTP-header
   (line)
   (echo "<body>") )

(out "file.mac"                            # Convert to Macintosh
   (in "file.txt"                          # from Unix or DOS format:
      (while (char)
         (prin
            (case @
               ("^M" NIL)                  # ignore CR
               ("^J" "^M")                 # convert CR to LF
               (T @) ) ) ) ) )             # otherwise no change

(out "c"                                   # Merge the contents of "a"
   (in "b"                                 # and "b" into "c"
      (in "a"
         (while (read)                     # Read an item from "a",
            (println @ (in -1 (read))) ) ) ) )  # print it with an item from "b"
```

20.7 Scripting

There are two possibilities to get the PicoLisp interpreter into doing useful work: via command line arguments, or as a stand-alone script.

20.7.1 Command line arguments for the PicoLisp interpreter

The command line can specify either files for execution, or arbitrary Lisp expressions for direct evaluation (see *Invocation*): if an argument starts with a hyphen, it is evaluated, otherwise it is `load` d as a file. A typical invocation might look like:

```
$ pil app/file1.l -main app/file2.l +
```

It loads the debugging environment, an application source file, calls the main function, and then loads another application source. In a typical development

and debugging session, this line is often modified using the shell's history mechanisms, e.g. by inserting debugging statements:

```
$ pil app/file1.l -"trace 'foo" -main -"debug 'bar" app/file2.l +
```

Another convenience during debugging and testing is to put things into the command line (shell history) which would otherwise have to be done each time in the application's user interface:

```
$ pil app/file1.l -main app/file2.l -go -'login "name" "password"' +
```

The final production release of an application usually includes a shell script, which initializes the environment, does some bookkeeping and cleanup, and calls the application with a proper command line. It is no problem if the command line is long and complicated.

For small utility programs, however, this is overkill. Enter full PicoLisp scripts.

20.7.2 PicoLisp scripts

It is better to write a single executable file using the mechanisms of "interpreter files". If the first two characters in an executable file are ' `#!` ', the operating system kernel will pass this file to an interpreter program whose pathname is given in the first line (optionally followed by a single argument). This is fast and efficient, because the overhead of a subshell is avoided.

Let's assume you installed PicoLisp in the directory "/home/foo/picolisp/", and put links to the executable and the installation directory as:

```
$ ln -s /home/foo/picolisp /usr/lib/picolisp
$ ln -s /usr/lib/picolisp/bin/picolisp /usr/bin/picolisp
```

Then a simple hello-world script might look like:

```
#!/usr/bin/picolisp /usr/lib/picolisp/lib.l
(prinl "Hello world!")
(bye)
```

If you write this into a text file, and use `chmod` to set it to "executable", it can be executed like any other command. Note that (because `#` is the comment character in PicoLisp) the first line will not be interpreted, and you can still

use that file as a normal command line argument to PicoLisp (useful during debugging).

20.7.3 Grab command line arguments from PicoLisp scripts

The fact that a hyphen causes evaluation of command line arguments can be used to simulate something like command line options. The following script defines two functions `a` and `f`, and then calls `(load T)` to process the rest of the command line (which otherwise would be ignored because of the `(bye)` statement):

```
#!/usr/bin/picolisp /usr/lib/picolisp/lib.l

(de a ()
   (println '-a '-> (opt)) )

(de f ()
   (println '-f '-> (opt)) )

(load T)
(bye)
```

(`opt` retrieves the next command line option)

Calling this script (let's say we named it "testOpts") gives:

```
$ ./testOpts -f abc
-f -> "abc"
$ ./testOpts -a xxx  -f yyy
-a -> "xxx"
-f -> "yyy"
```

We have to be aware of the fact, however, that the aggregation of arguments like

```
$ ./testOpts -axxx  -fyyy
```

or

```
$ ./testOpts -af yyy
```

cannot be achieved with this simple and general mechanism of command line processing.

20.7.4 Run scripts from arbitrary places on the host file system

Utilities are typically used outside the context of the PicoLisp environment. All examples above assumed that the current working directory is the PicoLisp installation directory, which is usually all right for applications developed in that environment. Command line file arguments like "app/file1.l" will be properly found.

To allow utilities to run in arbitrary places on the host file system, the concept of *home directory substitution* was introduced. The interpreter remembers internally at start-up the pathname of its first argument (usually "lib.l"), and substitutes any leading ' `@` ' character in subsequent file names with that pathname. Thus, to run the above example in some other place, simply write:

```
$ /home/foo/picolisp/dbg @app/file1.l -main @app/file2.l
```

that is, supply a full path name to the initial command (here 'p'), or put it into your `PATH` variable, and prefix each file which has to be loaded from the PicoLisp home directory with a `@` character. "Normal" files (not prefixed by `@`) will be opened or created relative to the current working directory as usual.

Stand-alone scripts will often want to load additional modules from the PicoLisp environment, beyond the "lib.l" we provided in the first line of the hello-world script. Typically, at least a call to

```
(load "@lib/misc.l")
```

(note the home directory substitution) will be included near the beginning of the script.

As a more complete example, here is a script which extracts the date, name and size of the latest official PicoLisp release version from the download web site, and prints it to standard output:

```
#!/usr/bin/picolisp /usr/lib/picolisp/lib.l

(load "@lib/misc.l" "@lib/http.l")

(use (@Date @Name @Size)
   (when
      (match
         '(@Date " " "-" " " @Name " " "(" @Size ")")
         (client "software-lab.de" 80 "down.html"
            (from "Release Archive")
            (from ".tgz">")
            (till "") ) )
      (prinl @Name)
      (prinl @Date " -- " @Size) ) )

(bye)
```

20.7.5 Editing scripts

We recommend that you have a terminal window open, and try the examples by yourself. You may either type them in, directly to the PicoLisp interpreter, or edit a separate source file (e.g. `''@doc/fun.l''`) in a second terminal window and load it into PicoLisp with

```
: (load "@doc/fun.l")
```

each time you have modified and saved it.

20.7.6 Editing scripts with vi

Once a function is loaded from a source file, you can call 'vim' directly on that function with

```
: (vi 'fact)
```

The function 'vi' opens the appropriate source file, and jumps to the right line where 'fact' is defined. When you modify it, you can simply call 'ld' to (re)load that source file

```
: (ld)
```

20.8 Objects and Classes

The PicoLisp object model is very simple, yet flexible and powerful. Objects as well as classes are both implemented as symbols. In fact, there is no formal difference between objects and classes; classes are more a conceptual design consideration in the head of the programmer than a physical reality.

Having said this, we declare that normally:

1. A Class
   - Has a name (interned symbol)
   - Has method definitions and superclass(es) in the value cell
   - May have class variables (attributes) in the property list
2. An Object
   - Has no name (anonymous symbol) or is an external symbol
   - Has class(es) (and optionally method definitions) in the value cell
   - Has instance variables (attributes) in the property list

So the main difference between classes and objects is that the former ones usually are internal symbols. By convention, their names start with a `+` . Sometimes it makes sense, however, to create named objects (as global singletons, for example), or even anonymous classes.

Both classes and objects have a list in their value cell, consisting of method definitions (often empty for objects) and (super)class(es). And both classes and objects have local data in their property lists (often empty for classes). This implies, that any given object (as an instance of a class) may have private (object-local) methods defined.

It is rather difficult to contrive a simple OOP example. We constructed a hierarchy of geometric shapes, with a base class `+Shape` and two subclasses `+Rectangle` and `+Circle`.

The source code is included as "`@doc/shape.l`" in the PicoLisp distribution, so you don't have to type it in. Just `load` the file, or start it from the shell as:

```
$ pil @doc/shape.l +
```

Let's look at it piece by piece. Here's the base class:

```
(class +Shape)
# x y

(dm T (X Y)
   (=: x X)
   (=: y Y) )

(dm move> (DX DY)
   (inc (:: x) DX)
   (inc (:: y) DY) )
```

The first line '`(class +Shape)`' defines the symbol `+Shape` as a class without superclasses. The following method definitions will go to that class.

The comment `# x y` in the second line is just a convention, to indicate what instance variables (properties) that class uses. As PicoLisp is a dynamic language, a class can be extended at runtime with any number of properties, and there is nothing like a fixed object size or structure. This comment is a hint of what the programmer thinks to be essential and typical for that class. In the case of `+Shape`, `x` and `y` are the coordinates of the shape's origin.

Then we have two method definitions, using the keyword `dm` for "define method". The first method is special, in that its name is `T`. Each time a new object is created, and a method with that name is found in its class hierarchy, that method will be executed. Though this looks like a "constructor" in other programming languages, it should probably better be called "initializer". The `T` method of `+Shape` takes two arguments `X` and `Y`, and stores them in the object's property list.

The second method `move>` changes the object's origin by adding the offset values `DX` and `DY` to the object's origin.

Now to the first derived class:

```
(class +Rectangle +Shape)
# dx dy

(dm T (X Y DX DY)
   (super X Y)
   (=: dx DX)
   (=: dy DY) )

(dm area> ()
   (* (: dx) (: dy)) )

(dm perimeter> ()
   (* 2 (+ (: dx) (: dy))) )

(dm draw> ()
   (drawRect (: x) (: y) (: dx) (: dy)) )
```

`+Rectangle` is defined as a subclass of `+Shape`. The comment `# dx dy` indicates that `+Rectangle` has a width and a height in addition to the origin coordinates inherited from `+Shape`.

The `T` method passes the origin coordinates `X` and `Y` to the `T` method of the superclass (`+Shape`), then stores the width and height parameters into `dx` and `dy`.

Next we define the methods `area>` and `perimeter>` which do some obvious calculations, and a method `draw>` which is supposed to draw the shape on the screen by calling some hypothetical function `drawRect`.

Finally, we define a `+Circle` class in an analog way, postulating the hypothetical function `drawCircle`:

```
(class +Circle +Shape)
# r

(dm T (X Y R)
   (super X Y)
   (=: r R) )

(dm area> ()
   (*/ (: r) (: r) 31415927 10000000) )

(dm perimeter> ()
   (*/ 2 (: r) 31415927 10000000) )

(dm draw> ()
   (drawCircle (: x) (: y) (: r)) )
```

Now we can experiment with geometrical shapes. We create a rectangle at point (0,0) with a width of 30 and a height of 20, and keep it in the variable `R`:

```
: (setq R (new '(+Rectangle) 0 0 30 20))  # New rectangle
-> $134432824                             # returned anonymous symbol
: (show R)
$134432824 (+Rectangle)                   # Show the rectangle
   dy 20
   dx 30
   y 0
   x 0
```

We see that the symbol `$134432824` has a list of classes '`(+Rectangle)`' in its value cell, and the coordinates, width and height in is property list.

Sending messages to that object

```
: (area> R)                               # Calculate area
-> 600
: (perimeter> R)                          # and perimeter
-> 100
```

will return the values for area and perimeter, respectively.

Then we move the object's origin:

```
: (move> R 10 5)                        # Move 10 right and 5 down
-> 5
: (show R)
$134432824 (+Rectangle)
   y 5                                  # Origin changed (0,0) -> (10,5)
   x 10
   dy 20
   dx 30
```

Though a method `move>` wasn't defined for the `+Rectangle` class, it is inherited from the `+Shape` superclass.

Similarly, we create and use a circle object:

```
: (setq C (new '(+Circle) 10 10 30))    # New circle
-> $134432607                           # returned anonymous symbol
: (show C)
$134432607 (+Circle)                    # Show the circle
   r 30
   y 10
   x 10
-> $134432607
: (area> C)                             # Calculate area
-> 2827
: (perimeter> C)                        # and perimeter
-> 188
: (move> C 10 5)                        # Move 10 right and 5 down
-> 15
: (show C)
$134432607 (+Circle)                    # Origin changed (10,10) -> (20,15)
   y 15
   x 20
   r 30
```

It is also easy to send messages to objects in a list:

```
: (mapcar 'area> (list R C))              # Get list of areas
-> (600 2827)
: (mapc
   '((Shape) (move> Shape 10 10))         # Move all 10 right and down
   (list R C) )
-> 25
: (show R)
$134431493 (+Rectangle)
   y 15
   x 20
   dy 20
   dx 30
-> $134431493
: (show C)
$134431523 (+Circle)
   y 25
   x 30
   r 30
```

Assume that we want to extend our shape system. From time to time, we need shapes that behave exactly like the ones above, but are tied to a fixed position. That is, they do not change their position even if they receive a `move>` message.

One solution would be to modify the `move>` method in the `+Shape` class to a no-operation. But this would require to duplicate the whole shape hierarchy (e.g. by defining `+FixedShape`, `+FixedRectangle` and `+FixedCircle` classes).

The PicoLisp Way is the use of Prefix Classes through multiple inheritance. It uses the fact that searching for method definitions is a depth-first, left-to-right search of the class tree. We define a prefix class:

```
: (class +Fixed)

(dm move> (DX DY))  # A do-nothing method
```

We can now create a fixed rectangle, and try to move it:

```
: (setq R (new '(+Fixed +Rectangle) 0 0 30 20))    # '+Fixed' prefix class
-> $134432881
: (move> R 10 5)                                   # Send 'move>' message
-> NIL
: (show R)
$134432881 (+Fixed +Rectangle)
   dy 20
   dx 30
   y 0                                             # Did not move!
   x 0
```

We see, prefix classes can surgically change the inheritance tree for selected objects or classes.

Alternatively, if fixed rectangles are needed often, it might make sense to define a new class `+FixRect`:

```
: (class +FixRect +Fixed +Rectangle)
-> +FixRect
```

and then use it directly:

```
: (setq R (new '(+FixRect) 0 0 30 20))
-> $13455710
```

20.9 Persistence (External Symbols)

PicoLisp has persistent objects built-in as a first class data type. With "first class" we mean not just the ability of being passed around, or returned from functions (that's a matter of course), but that they are a primary data type with their own interpreter tag bits. They are, in fact, a special type of symbolic atoms (called "*External Symbols*"), that happen to be read from pool file(s) when accessed, and written back automatically when modified.

In all other aspects they are normal symbols. They have a value cell, a property list and a name.

The name cannot be directly controlled by the programmer, as it is assigned when the symbol is created. It is an encoded index of the symbol's location in its database file. In its visual representation (output by the `print` functions and input by the `read` functions) it is surrounded by braces.

To make use of external symbols, you need to open a database first:

```
: (pool "test.db")
```

If a file with that name did not exist, it got created now. Also created at the same moment was {1}, the very first symbol in the file. This symbol is of great importance, and is handled especially by PicoLisp. Therefore a global constant *DB exists, which points to that symbol {1}, which should be used exclusively to access the symbol {1}, and which should never be modified by the programmer.

```
: *DB                   # The value of '*DB'
-> {1}                  # is '{1}'
: (show *DB)
{1} NIL                 # Value of '{1}' is NIL, property list empty
```

Now let's put something into the value cell and property list of {1}.

```
: (set *DB "Hello world")  # Set value of '{1}' to a transient symbol (string)
-> "Hello world"
: (put *DB 'a 1)           # Property 'a' to 1
-> 1
: (put *DB 'b 2)           # Property 'b' to 2
-> 2
: (show *DB)               # Now show the symbol '{1}'
{1} "Hello world"
   b 2
   a 1
```

Note that instead of '`(set *DB "Hello world")`', we might also have written '`(setq 1 "Hello world")`', and instead of '`(put *DB 'a 1)`' we might have written '`(put '1 'a 1)`'. This would have the same effect, but as a rule external symbols should never be be accessed literally in application programs, because the garbage collector might not be able to free these symbols and all symbols connected to them (and that might well be the whole database). It is all right, however, to access external symbols literally during interactive debugging.

Now we can create our first own external symbol. This can be done with `new` when a `T` argument is supplied:

```
: (new T)
-> {2}                  # Got a new symbol
```

We store it in the database root {1}:

```
: (put *DB 'newSym '{2})   # Literal '{2}' (ok during debugging)
-> {2}
: (show *DB)
{1} "Hello world"
   newSym {2}              # '{2}' is now stored in '{1}'
   b 2
   a 1
```

Put some property value into '{2}'

```
: (put *DB 'newSym 'x 777) # Put 777 as 'x'-property of '{2}'
-> 777
: (show *DB 'newSym)       # Show '{2}' (indirectly)
{2} NIL
   x 777
-> {2}
: (show '{2})              # Show '{2}' (directly)
{2} NIL
   x 777
```

All modifications to - and creations of - external symbols done so far are not written to the database yet. We could call `rollback` (or simply exit PicoLisp) to undo all the changes. But as we want to keep them:

```
: (commit)           # Commit all changes
-> T
: (bye)              # Exit picolisp
$                    # back to the shell
```

So, the next time when ..

```
$ pil +                 # .. we start PicoLisp
: (pool "test.db")      # and open the database file,
-> T
: (show *DB)            # our two symbols are there again
{1} "Hello world"
   newSym {2}
   b 2
   a 1
-> {1}
: (show *DB 'newSym)
{2} NIL
   x 777
-> {2}
```

20.10 Database Programming

To a database, there is more than just persistence. PicoLisp includes an entity/relation class framework (see also *Database*) which allows a close mapping of the application data structure to the database.

We provided a simple yet complete database and GUI demo application in `@doc/family.tgz` and `@doc/family64.tgz`. Please unpack the first one if you use a 32-bit system, and the second one on a 64-bit system. Both contain the sources in `@doc/family.l`, and an initial database in the "family/" subdirectory.

To use it, please unpack it first in your current working directory, then start it up in the following way:

```
$ pil family.l -main +
:
```

This loads the source file, initializes the database by calling the `main` function, and prompts for user input.

The data model is small and simple. We define a class `+Person` and two subclasses `+Man` and `+Woman`.

```
(class +Person +Entity)
```

`+Person` is a subclass of the `+Entity` system class. Usually all objects in a database are of a direct or indirect subclass of `+Entity`. We can then define the relations to other data with the `rel` function.

```
(rel nm (+Need +Sn +Idx +String))      # Name
```

This defines the name property (`nm`) of a person. The first argument to `rel` is always a list of relation classes (subclasses of `+relation`), optionally followed by further arguments, causing relation daemon objects be created and stored in the class definition. These daemon objects control the entity's behavior later at runtime.

Relation daemons are a kind of *metadata*, controlling the interactions between entities, and maintaining database integrity. Like other classes, relation classes can be extended and refined, and in combination with proper prefix classes a fine-grained description of the application's structure can be produced.

Besides primitive relation classes, like `+Number`, `+String` or `+Date`, there are

- relations between entities, like `+Link` (unidirectional link), `+Joint` (bidirectional link) or `+Hook` (object-local index trees)
- relations that bundle other relations into a single unit (`+Bag`)
- a `+List` prefix class
- a `+Blob` class for "binary large objects"
- prefix classes that maintain index trees, like `+Key` (unique index), `+Ref` (non-unique index) or `+Idx` (full text index)
- prefix classes which in turn modify index class behavior, like `+Sn` (modified soundex algorithm [1] for tolerant searches)
- a `+Need` prefix class, for existence checks
- a `+Dep` prefix class controlling dependencies between other relations

In the case of the person's name (`nm`) above, the relation object is of type `(+Need +Sn +Idx +String)`. Thus, the name of each person in this demo database is a mandatory attribute (`+Need`), searchable with the soundex algorithm (`+Sn`) and a full index (`+Idx`) of type `+String`.

```
(rel pa (+Joint) kids (+Man))          # Father
(rel ma (+Joint) kids (+Woman))        # Mother
(rel mate (+Joint) mate (+Person))     # Partner
```

The attributes for *father* (`pa`), *Mother* (`ma`) and *partner* (`mate`) are all defined as `+Joint`s. A `+Joint` is probably the most powerful relation mechanism in PicoLisp; it establishes a bidirectional link between two objects.

The above declarations say that the *father* (`pa`) attribute points to an object of type `+Man`, and is joined with that object's `kids` attribute (which is a list of joints back to all his children).

The consistency of `+Joint` is maintained automatically by the relation daemons. These become active whenever a value is stored to a person's `pa`, `ma`, `mate` or `kids` property.

For example, interesting things happen when a person's `mate` is changed to a new value. Then the `mate` property of the old mate's object is cleared (she has no mate after that). Now when the person pointed to by the new value already has a mate, then that mate's `mate` property gets cleared, and the happy new two mates now get their joints both set correctly.

The programmer doesn't have to care about all that. He just declares these relations as `+Joint` .

The last four attributes of person objects are just static data:

```
(rel job (+Ref +String))              # Occupation
(rel dat (+Ref +Date))                # Date of birth
(rel fin (+Ref +Date))                # Date of death
(rel txt (+String))                   # Info
```

They are all searchable via a non-unique index (`+Ref`). Date values in PicoLisp are just numbers, representing the day number (starting first of March of the year zero).

A method `url>` is defined:

```
(dm url> ()
   (list "!person" '*ID This) )
```

It is needed later in the GUI, to cause a click on a link to switch to that object.

The classes `+Man` and `+Woman` are subclasses of `+Person`:

```
(class +Man +Person)
(rel kids (+List +Joint) pa (+Person)) # Children

(class +Woman +Person)
(rel kids (+List +Joint) ma (+Person)) # Children
```

They inherit everything from `+Person`, except for the `kids` attribute. This attribute joins with the `pa` or `ma` attribute of the child, depending on the parent's gender.

That's the whole data model for our demo database application.

It is followed by a call to `dbs` ("database sizes"). This call is optional. If it is not present, the whole database will reside in a single file, with a block size of 256 bytes. If it is given, it should specify a list of items, each having a number in its CAR, and a list in its CDR. The CARs taken together will be passed later to *pool*, causing an individual database file with that size to be created. The CDRs tell what entity classes (if an item is a symbol) or index trees (if an item is a list with a class in its CAR and a list of relations in its CDR) should be placed into that file.

A handful of access functions is provided, that know about database relationships and thus allows higher-level access modes to the external symbols in a database.

For one thing, the B-Trees created and maintained by the index daemons can be used directly. Though this is rarely done in a typical application, they form the base mechanisms of other access modes and should be understood first.

The function `tree` returns the tree structure for a given relation. To iterate over the whole tree, the functions `iter` and `scan` can be used:

```
(iter (tree '+Person) '((P) (println (datStr (get P 'dat)) (get P 'nm))))
"1770-08-03" "Friedrich Wilhelm III"
"1776-03-10" "Luise Augusta of Mecklenburg-Strelitz"
"1797-03-22" "Wilhelm I"
...
```

They take a function as the first argument. It will be applied to all objects found in the tree (to show only a part of the tree, an optional begin- and end-value can be supplied), producing a simple kind of report.

More useful is `collect`; it returns a list of all objects that fall into a range of index values:

```
: (collect 'dat '+Person (date 1982 1 1) (date 1988 12 31))
-> ({2-M} {2-L} {2-E})
```

This returns all persons born between 1982 and 1988. Let's look at them with `show`:

```
: (more (collect 'dat '+Person (date 1982 1 1) (date 1988 12 31)) show)
{2-M} (+Man)
   nm "William"
   dat 724023
   ma {2-K}
   pa {2-J}
   job "Heir to the throne"

{2-L} (+Man)
   nm "Henry"
   dat 724840
   ma {2-K}
   pa {2-J}
   job "Prince"

{2-E} (+Woman)
   nm "Beatrice"
   dat 726263
   ma {2-D}
   job "Princess"
   pa {2-B}
```

If you are only interested in a certain attribute, e.g. the name, you can return it directly:

```
: (collect 'dat '+Person (date 1982 1 1) (date 1988 12 31) 'nm)
-> ("William" "Henry" "Beatrice")
```

To find a single object in the database, the function `db` is used:

```
: (db 'nm '+Person "Edward")
-> {2-;}
```

If the key is not unique, additional arguments may be supplied:

```
: (db 'nm '+Person "Edward"  'job "Prince"  'dat (date 1964 3 10))
-> {2-;}
```

The programmer must know which combination of keys will suffice to specify the object uniquely. The tree search is performed using the first value ("Edward"), while all other attributes are used for filtering. Later, in the *Pilog* section, we will show how more general (and possibly more efficient) searches can be performed.

20.11 User Interface (GUI) Programming

The only types of GUI supported by the PicoLisp application server framework is either dynamically generated (but static by nature) HTML, or an interactive XHTML/CSS framework with the optional use of JavaScript.

Before we explain the GUI of our demo database application, we present a minimal example for a plain HTML-GUI in `@doc/hello.l`. Start the application server as:

```
$ pil @lib/http.l -'server 8080 "@doc/hello.l"' -wait
```

Now point your browser to the address '`http://localhost:8080`'. You should see a very simple HTML page. You can come back here with the browser's BACK button.

You can call the page repeatedly, or concurrently with many clients if you like. To terminate the server, you have to send it a TERM signal (e.g. '`killall pil`'), or type the `Ctrl-C` key in the console window.

In our demo database application, a single function `person` is responsible for the whole GUI. Again, please look at `@doc/family.l`.

To start the database *and* the application server, call:

```
$ pil family.l -main -go +
```

As before, the database is opened with `main`. The function `go` is also defined in `@doc/family.l`:

```
(de go ()
   (server 8080 "!person") )
```

It starts the HTTP server listening on TCP port 8080 (we did a similar thing in our minimal GUI example above directly on the command line). Each connect to that port will cause the function `person` to be invoked.

Again, point your browser to the address '`http://localhost:8080`'.

You should see a new browser window with an input form created by the function `person`. We provided an initial database in "family/[1–4]". You can navigate through it by clicking on the pencil icons besides the input fields.

The chart with the children data can be scrolled using the down (`v`) and up (`^`) buttons.

A click on the button "Select" below opens a search dialog. You can scroll through the chart as before. Again, a click on a pencil will jump to that person. You can abort the dialog with a click on the "Cancel"-button.

The search fields in the upper part of the dialog allow a conjunctive search. If you enter "Edward" in the "Name" field and click "Search", you'll see all persons having the string "Edward" in their name. If you also enter "Duke" in the "Occupation" field, the result list will reduce to only two entries.

To create a new person, press the "New Man" or "New Woman" button. A new empty form will be displayed. Please type a name into the first field, and perhaps also an occupation and birth date. Any change of contents should be followed by a press on the "Done" button, though any other button (also Scroll or Select-buttons) will also do.

To assign a *father* attribute, you can either type a name directly into the field (if that person already exists in the database and you know the exact spelling), or use the "Set"-button (`->`) to the left of that field to open the search dialog. If you type in the name directly, your input must exactly match upper and lower case.

Alternatively, you may create a new person and assign a child in the "Children" chart.

On the console where you started PicoLisp, there should a prompt have appeared just when the browser connected. You can debug the application interactively while it is running. For example, the global variable `*Top` always contains the top level GUI object:

```
: (show *Top)
```

To take a look at the first field on the form:

```
: (show *Top 'gui 1)
```

A production application would be started in a slightly different way:

```
$ pil family.l -main -go -wait
```

In that case, no debug prompt will appear. In both cases, however, two `pil` processes will be running now. One is the initial server process which will continue to run until it is killed. The other is a child process holding the state of the GUI in the browser. It will terminate some time after the browser is closed, or when `(bye)` or a `Ctrl-D` is entered at the PicoLisp prompt.

Now back to the explanation of the GUI function `person`:

```
(de person ()
   (app)
   (action
      (html 0 (get (default *ID (val *DB)) 'nm) "@lib.css" NIL
         (form NIL
            (<h2> (<id> (: nm)))
```

For an in-depth explanation of that startup code, please refer to the guide to *PicoLisp Application Development*.

All components like fields and buttons are controlled by `form`. The function `gui` creates a single GUI component and takes the type (a list of classes) and a variable number of arguments depending on the needs of these classes.

```
(gui '(+E/R +TextField) '(nm : home obj) 40 "Name")
```

This creates a `+TextField` with the label "Name" and a length of 40 characters. The `+E/R` (: Entity/Relation) prefix class connects that field to a database object, the `nm` attribute of a person in this case, so that the person's name is displayed in that text field, and any changes entered into that field are propagated to the database automatically.

```
(gui '(+ClassField) '(: home obj) '(("Male" +Man) ("Female" +Woman)))
```

A `+ClassField` displays and changes the class of an object, in this case the person's sex from `+Man` to `+Woman` and vice versa.

As you see, there is no place where explicit accesses to the database have to be programmed, no `select` or `update`. This is all encapsulated in the GUI components, mainly in the `+E/R` prefix class. The above function `person` is fully functional as we present it and allows creation, modification and deletion of person objects in the database.

The two buttons on the bottom right generate simple reports:

The first one shows all contemporaries of the person that is currently displayed, i.e. all persons who did not die before, or were not born after that person. This is a typical PicoLisp report, in that in addition to the report's HTML page, a temporary file may be generated, suitable for download (and import into a spread sheet), and from which a PDF can be produced for print-out.

In PicoLisp, there is not a real difference between a plain HTML-GUI and a report. Again, the function `html` is used to generate the page.

The second report is much simpler. It produces a recursive structure of the family.

In both reports, links to the person objects are created which allow easy navigation through the database.

20.12 Pilog — PicoLisp Prolog

This sections explains some cases of using Pilog in typical application programming, in combination with persistent objects and databases. Please refer to the *Pilog* section of the PicoLisp Reference for the basic usage of Pilog.

Again, we use our demo application `@doc/family.l` that was introduced in the *Database Programming* section.

Normally, Pilog is used either interactively to query the database during debugging, or in applications to generate export data and reports. In the follow-

ing examples we use the interactive query front-end functions `?` and `select`. An application will use `goal` and `prove` directly, or use convenience functions like `pilog` or `solve`.

All Pilog access to external symbols is done via the two predicates `db/3` and `select/3`.

- `db/3` corresponds to the Lisp-level functions `db` and `collect`, as it derives its data from a single relation. It can be used for simple database queries.
- `select/3` provides for self-optimizing parallel access to an arbitrary number of relations. There is also a Lisp front-end function `select`, for convenient calls to the Pilog `select` predicate.

A predicate `show/1` is pre-defined for debugging purposes (a simple glue to the Lisp-level function `show`, see *Browsing*). Searching with `db/3` for all persons having the string "Edward" in their name:

```
: (? (db nm +Person "Edward" @P) (show @P))
{2-;} (+Man)
   nm "Edward"
   ma {2-:}
   pa {2-A}
   dat 717346
   job "Prince"
 @P={2-;}
{2-1B} (+Man)
   nm "Albert Edward"
   kids ({2-1C} {2-1D} {2-1E} {2-1F} {2-1G} {2-1H} {2-1I} {2-g} {2-a})
   job "Prince"
   mate {2-f}
   fin 680370
   dat 664554
 @P={2-1B}
...                # more results
```

To search for all persons with "Edward" in their name who are married to somebody with occupation "Queen":

```
   : (? (db nm +Person "Edward" @P) (val "Queen" @P mate job) (show @P))
   {2-1B} (+Man)
      mate {2-f}
      nm "Albert Edward"
      kids ({2-1C} {2-1D} {2-1E} {2-1F} {2-1G} {2-1H} {2-1I} {2-g} {2-a})
      job "Prince"
      fin 680370
      dat 664554
    @P={2-1B}
   -> NIL            # only one result
```

If you are interested in the names of "Albert Edward"'s children:

```
 : (? (db nm +Person "Albert Edward" @P) (lst @K @P kids) (val @Kid @K nm))
  @P={2-1B} @K={2-1C} @Kid="Beatrice Mary Victoria"
  @P={2-1B} @K={2-1D} @Kid="Leopold George Duncan"
  @P={2-1B} @K={2-1E} @Kid="Arthur William Patrick"
  @P={2-1B} @K={2-1F} @Kid="Louise Caroline Alberta"
  @P={2-1B} @K={2-1G} @Kid="Helena Augusta Victoria"
  @P={2-1B} @K={2-1H} @Kid="Alfred Ernest Albert"
  @P={2-1B} @K={2-1I} @Kid="Alice Maud Mary"
  @P={2-1B} @K={2-g} @Kid="Victoria Adelaide Mary"
  @P={2-1B} @K={2-a} @Kid="Edward VII"
 -> NIL
```

`db/3` can do a direct index access only for a single attribute (`nm` of `+Person` above). To search for several criteria at the same time, `select/3` has to be used:

```
: (?
   (select (@P)
      ((nm +Person "Edward") (nm +Person "Augusta" pa))  # Generator clauses
      (tolr "Edward" @P nm)                              # Filter clauses
      (tolr "Augusta" @P kids nm) )
   (show @P) )
{2-1B} (+Man)
   kids ({2-1C} {2-1D} {2-1E} {2-1F} {2-1G} {2-1H} {2-1I} {2-g} {2-a})
   mate {2-f}
   nm "Albert Edward"
   job "Prince"
   fin 680370
   dat 664554
 @P={2-1B}
-> NIL
```

`select/3` takes a list of generator clauses which are used to retrieve objects from the database, and a number of normal Pilog filter clauses. In the example above the generators are

- `(nm +Person "Edward")` to generate persons with "Edward" in their names, and
- `(nm +Person "Augusta" pa)` to find persons with "Augusta" in their names and generate persons using the `pa` ("father") attribute.

All persons generated are possible candidates for our selection. The `nm` index tree of `+Person` is traversed twice in parallel, optimizing the search in such a way that successful hits get higher priority in the search, depending on the filter clauses. The process will stop as soon as any one of the generators is exhausted. Note that this is different from the standard Prolog search algorithm.

The filter clauses in this example both use the pre-defined predicate `tolr/3` for *tolerant* string matches (according either to the soundex algorithm (see the section *Database Programming*) or to substring matches), and filter objects that

- match "Edward" in their name: `(tolr "Edward" @P nm)`, and
- match "Augusta" in one of their kids' names: `(tolr "Augusta" @P kids nm)`

A more typical and extensive example for the usage of `select` can be found in the `qPerson` function in `@doc/family.l`. It is used in the search dialog of the demo application, and searches for a person with the name, the parents' and partner's names, the occupation and a time range for the birth date. The relevant index trees in the database are searched (actually only those trees where the user entered a search key in the corresponding dialog field), and a logical AND of the search attributes is applied to the result.

For example, press the "Select" button, enter "Elizabeth" into the "Mother" search field and "Phil" in the "Partner" search field, meaning to look for all persons whose mother's name is like "Elizabeth" and whose partner's name is like "Phil". As a result, two persons ("Elizabeth II" and "Anne") will show up.

In principle, `db/3` can be seen as a special case of `select/3`. The following two queries are equivalent:

```
: (? (db nm +Person "Edward" @P))
 @P={2-;}
 @P={2-1B}
 @P={2-R}
 @P={2-1K}
 @P={2-a}
 @P={2-T}
-> NIL
: (? (select (@P) ((nm +Person "Edward"))))
 @P={2-;}
 @P={2-1B}
 @P={2-R}
 @P={2-1K}
 @P={2-a}
 @P={2-T}
-> NIL
```

20.13 Poor Man's SQL

20.13.1 select

For convenience, a `select` Lisp glue function is provided as a front-end to the `select` predicate. Note that this function does not evaluate its arguments (it is intended for interactive use), and that it supports only a subset of the predicate's functionality. The syntax resembles SELECT in the SQL language, for example:

```
# SELECT * FROM Person
: (select +Person)  # Step through the whole database
{2-o} (+Man)
   nm "Adalbert Ferdinand Berengar Viktor of Prussia"
   dat 688253
   ma {2-j}
   pa {2-h}
   fin 711698

{2-1B} (+Man)
   nm "Albert Edward"
   dat 664554
   job "Prince"
   mate {2-f}
   kids ({2-1C} {2-1D} {2-1E} {2-1F} {2-1G} {2-1H} {2-1I} {2-g} {2-a})
   fin 680370
...
```

```
# SELECT * FROM Person WHERE nm LIKE "%Edward%"
: (select +Person nm "Edward")  # Show all Edwards
{2-;} (+Man)
   nm "Edward"
   dat 717346
   job "Prince"
   ma {2-:}
   pa {2-A}

{2-1B} (+Man)
   nm "Albert Edward"
   dat 664554
   job "Prince"
   kids ({2-1C} {2-1D} {2-1E} {2-1F} {2-1G} {2-1H} {2-1I} {2-g} {2-a})
   mate {2-f}
   fin 680370
...

# SELECT nm, dat FROM Person WHERE nm LIKE "%Edward%"
: (select nm dat +Person nm "Edward")
"Edward" "1964-03-10" {2-;}
"Albert Edward" "1819-08-26" {2-1B}
"George Edward" NIL {2-R}
"Edward Augustus Hanover" NIL {2-1K}
...

# SELECT dat, fin, p1.nm, p2.nm
#    FROM Person p1, Person p2
#    WHERE p1.nm LIKE "%Edward%"
#    AND p1.job LIKE "King%"
#    AND p1.mate = p2.mate  -- Actually, in a SQL model we'd need
#                           -- another table here for the join
: (select dat fin nm (mate nm) +Person nm "Edward" job "King")
"1894-06-23" "1972-05-28" "Edward VIII" "Wallace Simpson" {2-T}
"1841-11-09" NIL "Edward VII" "Alexandra of Denmark" {2-a}
-> NIL
```

20.13.2 update

In addition (just to stay with the SQL terminology ;-), there is also an `update` function. It is a front-end to the `set!>` and `put!>` transaction methods, and should be used when single objects in the database have to be modified by hand.

In principle, it would also be possible to use the `edit` function to modify a database object. This is not recommended, however, because `edit` does not know about relations to other objects (like Links, Joints and index trees) and may easily cause database corruption.

In the most general case, the value of a property in a database object is changed with the `put!>` method. Let's look at "Edward" from the previous examples:

```
: (show '{2-;})
{2R} (+Man)
   job "Prince"
   nm "Edward"
   dat 717346
   ma {2-:}
   pa {20A}
-> {2-;}
```

We might change the name to "Johnny" with `put!>`:

```
: (put!> '{2-;} 'nm "Johnny")
-> "Johnny"
```

However, an easier and less error-prone prone way - especially when more than one property has to be changed - is using `update`. It presents the value cell (the list of classes) and then each property on its own line, allowing the user to change it with the *command line editor*.

Just hitting ENTER will leave that property unchanged. To modify it, you'll typically hit ESC to get into command mode, and move the cursor to the point of change.

For properties with nested list structures (`+List +Bag`), `update` will recurse into the data structure.

```
: (update '{2-;})
{2-;} (+Man)      # ENTER
nm "Johnny"       # Modified the name to "Johnny"
ma {2-:}          # ENTER
pa {2-A}          # ENTER
dat 1960-03-10    # Modified the year from "1964" to "1960"
job "Prince"      # ENTER
-> {2-;}
```

All changes are committed immediately, observing the rules of database synchronization so that any another user looking at the same object will have his GUI updated correctly.

To abort `update`, hit `Ctrl-X`.

If only a single property has to be changed, `update` can be called directly for that property:

```
: (update '{2-;} 'nm)
{2-;} nm "Edward"
...
```

References

1. Donald E. Knuth: "The Art of Computer Programming", Vol.3, Addison-Wesley, 1973, p. 392

21

PicoLisp Application Development

Alexander Burger

abu@software-lab.de

Summary. This article presents an introduction to writing browser-based applications in PicoLisp.

It concentrates on the XHTML/CSS GUI-Framework (as opposed to the previous Java-AWT, Java-Swing and Plain-HTML frameworks), which is easier to use, more flexible in layout design, and does not depend on plug-ins, JavaScript or cookies.

21.1 Introduction

A plain HTTP/HTML GUI has various advantages: It runs on any browser, and can be fully driven by scripts ("@lib/scrape.l").

To be precise: CSS *can* be used to enhance the layout. And browsers *with* JavaScript will respond faster and smoother. But this framework works just fine in browsers which do not know anything about CSS or JavaScript. All examples were also tested using the w3m text browser.

For basic informations about the PicoLisp system please look at the *PicoLisp Reference* and the *PicoLisp Tutorial*. Knowledge of HTML, and a bit of CSS and HTTP is assumed.

The examples assume that PicoLisp was started from a global installation (see *Installation*).

21.2 Static Pages

You can use PicoLisp to generate static HTML pages. This does not make much sense in itself, because you could directly write HTML code as well, but

it forms the base for interactive applications, and allows us to introduce the application server and other fundamental concepts.

21.2.1 Hello World

To begin with a minimal application, please enter the following two lines into a generic source file named "project.l" in the PicoLisp installation directory.

```
########################################################################
(html 0 "Hello" "@lib.css" NIL
   "Hello World!" )
########################################################################
```

(We will modify and use this file in all following examples and experiments. Whenever you find such a program snippet between hash ('#') lines, just copy and paste it into your "project.l" file, and press the "reload" button of your browser to view the effects)

Start the application server

Open a second terminal window, and start a PicoLisp application server

```
$ pil @lib/http.l @lib/xhtml.l @lib/form.l -'server 8080 "project.l"' +
```

No prompt appears. The server just sits, and waits for connections. You can stop it later by hitting `Ctrl-C` in that terminal, or by executing `killall pil` in some other window.

(In the following, we assume that this HTTP server is up and running)

Now open the URL '`http://localhost:8080`' with your browser. You should see an empty page with a single line of text.

How does it work?

The above line loads the debugger (via the '+' switch), the HTTP server code ("@lib/http.l"), the XHTML functions ("@lib/xhtml.l") and the input form framework ("@lib/form.l", it will be needed later for *interactive forms*).

Then the `server` function is called with a port number and a default URL. It will listen on that port for incoming HTTP requests in an endless loop.

Whenever a GET request arrives on port 8080, the file "project.l" will be `(load)`ed, causing the evaluation (= execution) of all its Lisp expressions.

During that execution, all data written to the current output channel is sent directly to to the browser. The code in "project.l" is responsible to produce HTML (or anything else the browser can understand).

21.2.2 URL Syntax

The PicoLisp application server uses a slightly specialized syntax when communicating URLs to and from a client. The "path" part of an URL - which remains when

- the preceding protocol, host and port specifications,
- and the trailing question mark plus arguments

are stripped off - is interpreted according so some rules. The most prominent ones are:

- If a path starts with an exclamation-mark ('!'), the rest (without the '!') is taken as the name of a Lisp function to be called. All arguments following the question mark are passed to that function.
- If a path ends with ".l" (a dot and a lower case 'L'), it is taken as a Lisp source file name to be `(load)`ed. This is the most common case, and we use it in our example "project.l".
- If the extension of a file name matches an entry in the global mime type table `*Mimes`, the file is sent to the client with mime-type and max-age values taken from that table.
- Otherwise, the file is sent to the client with a mime-type of "application/octet-stream" and a max-age of 1 second.

An application is free to extend or modify the `*Mimes` table with the `mime` function. For example

```
(mime "doc" "application/msword" 60)
```

defines a new mime type with a max-age of one minute.

Argument values in URLs, following the path and the question mark, are encoded in such a way that Lisp data types are preserved:

- An internal symbol starts with a dollar sign ('$')
- A number starts with a plus sign ('+')

- An external (database) symbol starts with dash ('-')
- A list (one level only) is encoded with underscores ('_')
- Otherwise, it is a transient symbol (a plain string)

In that way, high-level data types can be directly passed to functions encoded in the URL, or assigned to global variables before a file is loaded.

21.2.3 Security

It is, of course, a huge security hole that - directly from the URL - any Lisp source file can be loaded, and any Lisp function can be called. For that reason, applications must take care to declare exactly which files and functions are to be allowed in URLs. The server checks a global variable `*Allow`, and - when its value is non `NIL` - denies access to anything that does not match its contents.

Normally, `*Allow` is not manipulated directly, but set with the `allowed` and `allow` functions

```
(allowed ("app/")
   "!start" "@lib.css" "customer.l" "article.l" )
```

This is usually called in the beginning of an application, and allows access to the directory "@img/", to the function 'start', and to the files "@lib.css", "customer.l" and "article.l".

Later in the program, `*Allow` may be dynamically extended with `allow`

```
(allow "!foo")
(allow "newdir/" T)
```

This adds the function 'foo', and the directory "newdir/", to the set of allowed items.

The ".pw" File

For a variety of security checks (most notably for using the `psh` function, as in some later examples) it is necessary to create a file named ".pw" in the PicoLisp installation directory. This file should contain a single line of arbitrary data, to be used as a password for identifying local resources.

The recommeded way to create this file is to call the `pw` function, defined in "@lib/http.l"

```
$ pil @lib/http.l -'pw 12' -bye
```

Please execute this command.

21.2.4 The `html` Function

Now back to our "Hello World" example. In principle, you could write "project.l" as a sequence of print statements

```
#########################################################################
(prinl "HTTP/1.0 200 OK^M")
(prinl "Content-Type: text/html; charset=utf-8")
(prinl "^M")
(prinl "<html>")
(prinl "Hello World!")
(prinl "</html>")
#########################################################################
```

but using the `html` function is much more convenient.

Moreover, `html` **is** nothing more than a printing function. You can see this easily if you connect a PicoLisp Shell (`psh`) to the server process (you must have generated a *".pw" file* for this), and enter the `html` statement

```
$ /usr/lib/picolisp/bin/psh 8080
: (html 0 "Hello" "@lib.css" NIL "Hello World!")
HTTP/1.0 200 OK
Server: PicoLisp
Date: Fri, 29 Dec 2006 07:28:58 GMT
Cache-Control: max-age=0
Cache-Control: no-cache
Content-Type: text/html; charset=utf-8
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lan="en" lang="en">
<head>
<title>Hello</title>
<base href="http://localhost:8080/"/>
<link rel="stylesheet" type="text/css" href="http://localhost:8080/@lib.css"/>
</head>
<body>Hello World!</body>
</html>
-> </html>
:  # (type Ctrl-D here to terminate PicoLisp)
```

These are the arguments to `html`:

1. `0`: A max-age value for cache-control (in seconds, zero means "no-cache"). You might pass a higher value for pages that change seldom, or `NIL` for no cache-control at all.
2. `''Hello''`: The page title.
3. `"@lib.css"`: A CSS-File name. Pass `NIL` if you do not want to use any CSS-File, or a list of file names if you want to give more than one CSS-File.
4. `NIL`: A CSS style attribute specification (see the description of *CSS Attributes* below). It will be passed to the `body` tag.

After these four arguments, an arbitrary number of expressions may follow. They form the body of the resulting page, and are evaluated according to a special rule. This rule is slightly different from the evaluation of normal Lisp expressions:

- If an argument is an atom (a number or a symbol (string)), its value is printed immediately.
- Otherwise (a list), it is evaluated as a Lisp function (typically some form of print statement).

Therefore, our source file might as well be written as:

```
########################################################################
(html 0 "Hello" "@lib.css" NIL
   (prinl "Hello World!") )
########################################################################
```

The most typical print statements will be some HTML-tags:

```
########################################################################
(html 0 "Hello" "@lib.css" NIL
   (<h1> NIL "Hello World!")
   (<br> "This is some text.")
   (ht:Prin "And this is a number: " (+ 1 2 3)) )
########################################################################
```

`<h1>` and `<br>` are tag functions. `<h1>` takes a CSS attribute as its first argument.

Note the use of `ht:Prin` instead of `prin`. `ht:Prin` should be used for all direct printing in HTML pages, because it takes care to escape special characters.

21.2.5 CSS Attributes

The `html` function above, and many of the HTML *tag functions*, accept a CSS attribute specification. This may be either an atom, a cons pair, or a list of cons pairs. We demonstrate the effects with the `<h1>` tag function.

An atom (usually a symbol or a string) is taken as a CSS class name

```
: (<h1> 'foo "Title")
<h1 class="foo">Title</h1>
```

For a cons pair, the CAR is taken as an attribute name, and the CDR as the attribute's value

```
: (<h1> '(id . bar) "Title")
<h1 id="bar">Title</h1>
```

Consequently, a list of cons pairs gives a set of attribute-value pairs

```
: (<h1> '((id . "abc") (lang . "de")) "Title")
<h1 id="abc" lang="de">Title</h1>
```

21.2.6 Tag Functions

All pre-defined XHTML tag functions can be found in "@lib/xhtml.l". We recommend to look at their sources, and to experiment a bit, by executing them at a PicoLisp prompt, or by pressing the browser's "Reload" button after editing the "project.l" file.

For a suitable PicoLisp prompt, either execute (in a separate terminal window) the PicoLisp Shell (`psh`) command (works only if the application server is running, and you did generate a *".pw" file*)

```
$ /usr/lib/picolisp/bin/psh 8080
:
```

or start the interpreter stand-alone, with "@lib/xhtml.l" loaded

```
$ pil @lib/http.l @lib/xhtml.l +
:
```

Note that for all these tag functions the above *tag body evaluation rule* applies.

Simple Tags

Most tag functions are simple and straightforward. Some of them just print their arguments

```
: (<br> "Hello world")
Hello world<br/>

: (<em> "Hello world")
<em>Hello world</em>
```

while most of them take a *CSS attribute specification* as their first argument (like the `<h1>` tag above)

```
: (<div> 'main "Hello world")
<div class="main">Hello world</div>

: (<p> NIL "Hello world")
<p>Hello world</p>

: (<p> 'info "Hello world")
<p class="info">Hello world</p>
```

All of these functions take an arbitrary number of arguments, and may nest to an arbitrary depth (as long as the resulting HTML is legal)

```
: (<div> 'main
   (<h1> NIL "Head")
   (<p> NIL
      (<br> "Line 1")
      "Line"
      (<nbsp>)
      (+ 1 1) ) )
<div class="main"><h1>Head</h1>
<p>Line 1<br/>
Line2</p>
</div>
```

(Un)ordered Lists

HTML-lists, implemented by the `<ol>` and `<ul>` tags, let you define hierarchical structures. You might want to paste the following code into your copy of "project.l":

```
########################################################################
(html 0 "Unordered List" "@lib.css" NIL
   (<ul> NIL
      (<li> NIL "Item 1")
      (<li> NIL
         "Sublist 1"
         (<ul> NIL
            (<li> NIL "Item 1-1")
            (<li> NIL "Item 1-2") ) )
      (<li> NIL "Item 2")
      (<li> NIL
         "Sublist 2"
         (<ul> NIL
            (<li> NIL "Item 2-1")
            (<li> NIL "Item 2-2") ) )
      (<li> NIL "Item 3") ) )
########################################################################
```

Here, too, you can put arbitrary code into each node of that tree, including other tag functions.

Tables

Like the hierarchical structures with the list functions, you can generate two-dimensional tables with the `<table>` and `<row>` functions.

The following example prints a table of numbers and their squares:

```
########################################################################
(html 0 "Table" "@lib.css" NIL
   (<table> NIL NIL NIL
      (for N 10                                    # A table with 10 rows
         (<row> NIL N (prin (* N N))) ) ) )     # and 2 columns
########################################################################
```

The first argument to `<table>` is the usual CSS attribute, the second an optional title ("caption"), and the third an optional list specifying the column headers. In that list, you may supply a list for a each column, with a CSS attribute in its CAR, and a tag body in its CDR for the contents of the column header.

The body of `<table>` contains calls to the `<row>` function. This function is special in that each expression in its body will go to a separate column of the table. If both for the column header and the row function an CSS attribute is given, they will be combined by a space and passed to the HTML `<td>` tag. This permits distinct CSS specifications for each column and row.

As an extension of the above table example, let's pass some attributes for the table itself (not recommended - better define such styles in a CSS file and then just pass the class name to `<table>`), right-align both columns, and print each row in an alternating red and blue color

```
########################################################################
(html 0 "Table" "@lib.css" NIL
   (<table>
      '((width . "200px") (style . "border: dotted 1px;"))    # table style
      "Square Numbers"                                         # caption
      '((align "Number") (align "Square"))                     # 2 headers
      (for N 10                                                   # 10 rows
         (<row> (xchg '(red) '(blue))                          # red or blue
            N                                                  # 2 columns
            (prin (* N N) ) ) ) ) )
########################################################################
```

If you wish to concatenate two or more cells in a table, so that a single cell spans several columns, you can pass the symbol `-` for the additional cell data to `<row>`. This will cause the data given to the left of the `-` symbols to expand to the right.

You can also directly specify table structures with the simple `<th>`, `<tr>` and `<td>` tag functions.

If you just need a two-dimensional arrangement of components, the even simpler `<grid>` function might be convenient:

```
########################################################################
(html 0 "Grid" "@lib.css" NIL
   (<grid> 3
      "A" "B" "C"
      123 456 789 ) )
########################################################################
```

It just takes a specification for the number of columns (here: 3) as its first argument, and then a single expression for each cell. Instead of a number, you can also pass a list of CSS attributes. Then the length of that list will determine the number of columns. You can change the second line in the above example to

```
(<grid> '(NIL NIL right)
```

Then the third column will be right aligned.

Menus and Tabs

The two most powerful tag functions are `<menu>` and `<tab>`. Used separately or in combination, they form a navigation framework with

- menu items which open and close submenus
- submenu items which switch to different pages
- tabs which switch to different subpages

The following example is not very useful, because the URLs of all items link to the same "project.l" page, but it should suffice to demonstrate the functionality:

```
########################################################################
(html 0 "Menu+Tab" "@lib.css" NIL
   (<div> '(id . menu)
      (<menu>
         ("Item" "project.l")                        # Top level item
         (NIL (<hr>))                                # Plain HTML
         (T "Submenu 1"                              # Submenu
            ("Subitem 1.1" "project.l")
            (T "Submenu 1.2"
               ("Subitem 1.2.1" "project.l")
               ("Subitem 1.2.2" "project.l")
               ("Subitem 1.2.3" "project.l") )
            ("Subitem 1.3" "project.l") )
         (T "Submenu 2"
            ("Subitem 2.1" "project.l")
            ("Subitem 2.2" "project.l") ) ) )
   (<div> '(id . main)
      (<h1> NIL "Menu+Tab")
      (<tab>
         ("Tab1"
            (<h3> NIL "This is Tab 1") )
         ("Tab2"
            (<h3> NIL "This is Tab 2") )
         ("Tab3"
            (<h3> NIL "This is Tab 3") ) ) ) )
########################################################################
```

`<menu>` takes a sequence of menu items. Each menu item is a list, with its CAR either

- `NIL`: The entry is not an active menu item, and the rest of the list may consist of arbitrary code (usually HTML tags).
- `T`: The second element is taken as a submenu name, and a click on that name will open or close the corresponding submenu. The rest of the list recursively specifies the submenu items (may nest to arbitrary depth).
- Otherwise: The menu item specifies a direct action (instead of opening a submenu), where the first list element gives the item's name, and the second element the corresponding URL.

`<tab>` takes a list of subpages. Each page is simply a tab name, followed by arbitrary code (typically HTML tags).

Note that only a single menu and a single tab may be active at the same time.

21.3 Interactive Forms

In HTML, the only possibility for user input is via `<form>` and `<input>` elements, using the HTTP POST method to communicate with the server.

"@lib/xhtml.l" defines a function called `<post>`, and a collection of input tag functions, which allow direct programming of HTML forms. We will supply only one simple example:

```
########################################################################
(html 0 "Simple Form" "@lib.css" NIL
   (<post> NIL "project.l"
      (<field> 10 '*Text)
      (<submit> "Save") ) )
########################################################################
```

This associates a text input field with a global variable `*Text`. The field displays the current value of `*Text`, and pressing the submit button causes a reload of "project.l" with `*Text` set to any string entered by the user.

An application program could then use that variable to do something useful, for example store its value in a database.

The problem with such a straightforward use of forms is that

1. they require the application programmer to take care of maintaining lots of global variables. Each input field on the page needs an associated variable for the round trip between server and client.
2. they do not preserve an application's internal state. Each POST request spawns an individual process on the server, which sets the global variables to their new values, generates the HTML page, and terminates thereafter. The application state has to be passed along explicitly, e.g. using `<hidden>` tags.
3. they are not very interactive. There is typically only a single submit button. The user fills out a possibly large number of input fields, but changes will take effect only when the submit button is pressed.

Though we wrote a few applications in that style, we recommend the GUI framework provided by "@lib/form.l". It does not need any variables for the client/server communication, but implements a class hierarchy of GUI components for the abstraction of application logic, button actions and data linkage.

21.3.1 Sessions

First of all, we need to establish a persistent environment on the server, to handle each individual session (for each connected client).

Technically, this is just a child process of the server we started *above*, which does not terminate immediately after it sent its page to the browser. It is achieved by calling the `app` function somewhere in the application's startup code.

```
#########################################################################
(app)  # Start a session

(html 0 "Simple Session" "@lib.css" NIL
   (<post> NIL "project.l"
      (<field> 10 '*Text)
      (<submit> "Save") ) )
#########################################################################
```

Nothing else changed from the previous example. However, when you connect your browser and then look at the terminal window where you started the application server, you'll notice a colon, the PicoLisp prompt

```
$ pil @lib/http.l @lib/xhtml.l @lib/form.l -'server 8080 "project.l"' +
:
```

Tools like the Unix `ps` utility will tell you that now two `picolisp` processes are running, the first being the parent of the second.

If you enter some text, say "abcdef", into the text field in the browser window, press the submit button, and inspect the Lisp `*Text` variable,

```
: *Text
-> "abcdef"
```

you see that we now have a dedicated PicoLisp process, "connected" to the client.

You can terminate this process (like any interactive PicoLisp) by hitting `Ctrl-D` on an empty line. Otherwise, it will terminate by itself if no other browser requests arrive within a default timeout period of 5 minutes.

To start a (non-debug) production version, the server is commonly started without the '+' flag, and with `-wait`

```
$ pil @lib/http.l @lib/xhtml.l @lib/form.l -'server 8080 "project.l"' -wait
```

In that way, no command line prompt appears when a client connects.

21.3.2 Action Forms

Now that we have a persistent session for each client, we can set up an active GUI framework.

This is done by wrapping the call to the `html` function with `action`. Inside the body of `html` can be - in addition to all other kinds of tag functions - one or more calls to `form`

```
########################################################################
(app)                                                  # Start session

(action                                                # Action handler
   (html 0 "Form" "@lib.css" NIL                       # HTTP/HTML protocol
      (form NIL                                        # Form
         (gui 'a '(+TextField) 10)                     # Text Field
         (gui '(+Button) "Print"                       # Button
            '(msg (val> (: home a))) ) ) ) )
########################################################################
```

Note that there is no longer a global variable like `*Text` to hold the contents of the input field. Instead, we gave a local, symbolic name `a` to a `+TextField` component

```
(gui 'a '(+TextField) 10)                              # Text Field
```

Other components can refer to it

```
'(msg (val> (: home a)))
```

`(: home)` is always the form which contains this GUI component. So `(: home a)` evaluates to the component `a` in the current form. As `msg` prints its argument to standard error, and the `val>` method retrieves the current contents of a component, we will see on the console the text typed into the text field when we press the button.

An `action` without embedded `form`s - or a `form` without a surrounding `action` - does not make much sense by itself. Inside `html` and `form`, however, calls to

HTML functions (and any other Lisp functions, for that matter) can be freely mixed.

In general, a typical page may have the form

```
(action                                          # Action handler
   (html ..                                      # HTTP/HTML protocol
      (<h1> ..)                                  # HTML tags
      (form NIL                                  # Form
         (<h3> ..)
         (gui ..)                                # GUI component(s)
         (gui ..)
         .. )
      (<h2> ..)
      (form NIL                                  # Another form
         (<h3> ..)
         (gui ..)                                # GUI component(s)
         .. )
      (<br> ..)
      .. ) )
```

The `gui` Function

The most prominent function in a `form` body is `gui`. It is the workhorse of GUI construction.

Outside of a `form` body, `gui` is undefined. Otherwise, it takes an optional alias name, a list of classes, and additional arguments as needed by the constructors of these classes. We saw this example before

```
(gui 'a '(+TextField) 10)                # Text Field
```

Here, `a` is an alias name for a component of type `(+TextField)`. The numeric argument `10` is passed to the text field, specifying its width. See the chapter on *GUI Classes* for more examples.

During a GET request, `gui` is basically a front-end to `new`. It builds a component, stores it in the internal structures of the current form, and initializes it by sending the `init>` message to the component. Finally, it sends it the `show>` message, to produce HTML code and transmit it to the browser.

During a POST request, `gui` does not build any new components. Instead, the existing components are re-used. So `gui` does not have much more to do than sending the `show>` message to a component.

Control Flow

HTTP has only two methods to change a browser window: GET and POST. We employ these two methods in a certain defined, specialized way:

- GET means, a **new page** is being constructed. It is used when a page is visited for the first time, usually by entering an URL into the browser's address field, or by clicking on a link (which is often a *submenu item or tab*).
- POST is always directed to the **same page**. It is triggered by a button press, updates the corresponding form's data structures, and executes that button's action code.

A button's action code can do almost anything: Read and modify the contents of input fields, communicate with the database, display alerts and dialogs, or even fake the POST request to a GET, with the effect of showing a completely different document (See *Switching URLs*).

GET builds up all GUI components on the server. These components are objects which encapsulate state and behavior of the HTML page in the browser. Whenever a button is pressed, the page is reloaded via a POST request. Then - before any output is sent to the browser - the `action` function takes control. It performs error checks on all components, processes possible user input on the HTML page, and stores the values in correct format (text, number, date, object etc.) in each component.

The state of a form is preserved over time. When the user returns to a previous page with the browser's BACK button, that state is reactivated, and may be POSTed again.

The following silly example displays two text fields. If you enter some text into the "Source" field, you can copy it in upper or lower case to the "Destination" field by pressing one of the buttons

```
########################################################################
(app)

(action
   (html 0 "Case Conversion" "@lib.css" NIL
      (form NIL
         (<grid> 2
            "Source" (gui 'src '(+TextField) 30)
            "Destination" (gui 'dst '(+Lock +TextField) 30) )
         (gui '(+JS +Button) "Upper Case"
            '(set> (: home dst)
               (uppc (val> (: home src))) ) )
         (gui '(+JS +Button) "Lower Case"
            '(set> (: home dst)
               (lowc (val> (: home src))) ) ) ) ) )
########################################################################
```

The `+Lock` prefix class in the "Destination" field makes that field read-only. The only way to get some text into that field is by using one of the buttons.

Switching URLs

Because an action code runs before `html` has a chance to output an HTTP header, it can abort the current page and present something different to the user. This might, of course, be another HTML page, but would not be very interesting as a normal link would suffice. Instead, it can cause the download of dynamically generated data.

The next example shows a text area and two buttons. Any text entered into the text area is exported either as a text file via the first button, or a PDF document via the second button

```
#############################################################################
(load "@lib/ps.l")

(app)

(action
   (html 0 "Export" "@lib.css" NIL
      (form NIL
         (gui '(+TextField) 30 8)
         (gui '(+Button) "Text"
            '(let Txt (tmp "export.txt")
               (out Txt (prinl (val> (: home gui 1))))
               (url Txt) ) )
         (gui '(+Button) "PDF"
            '(psOut NIL "foo"
               (a4)
               (indent 40 40)
               (down 60)
               (hline 3)
               (font (14 . "Times-Roman")
                  (ps (val> (: home gui 1))) )
               (hline 3)
               (page) ) ) ) ) )
#############################################################################
```

(a text area is built when you supply two numeric arguments (columns and rows) to a `+TextField` class)

The action code of the first button creates a temporary file (i.e. a file named "export.txt" in the current process's temporary space), prints the value of the text area (this time we did not bother to give it a name, we simply refer to it as the form's first gui list element) into that file, and then calls the `url` function with the file name.

The second button uses the PostScript library "@lib/ps.l" to create a temporary file "foo.pdf". Here, the temporary file creation and the call to the `url` function is hidden in the internal mechanisms of `psOut`. The effect is that the browser receives a PDF document and displays it.

Alerts and Dialogs

Alerts and dialogs are not really what they used to be ;-)

They do not "pop up". In this framework, they are just a kind of simple-to-use, pre-fabricated form. They can be invoked by a button's action code, and

appear always on the current page, immediately preceding the form which created them.

Let's look at an example which uses two alerts and a dialog. In the beginning, it displays a simple form, with a locked text field, and two buttons

```
########################################################################
(app)

(action
   (html 0 "Alerts and Dialogs" "@lib.css" NIL
      (form NIL
         (gui '(+Init +Lock +TextField) "Initial Text" 20 "My Text")
         (gui '(+Button) "Alert"
            '(alert NIL "This is an alert " (okButton)) )
         (gui '(+Button) "Dialog"
            '(dialog NIL
               (<br> "This is a dialog.")
               (<br>
                  "You can change the text here "
                  (gui '(+Init +TextField) (val> (: top 1 gui 1)) 20) )
               (<br> "and then re-submit it to the form.")
               (gui '(+Button) "Re-Submit"
                  '(alert NIL "Are you sure? "
                     (yesButton
                        '(set> (: home top 2 gui 1)
                           (val> (: home top 1 gui 1)) ) )
                     (noButton) ) )
               (cancelButton) ) ) ) )
########################################################################
```

The `+Init` prefix class initializes the "My Text" field with the string "Initial Text". As the field is locked, you cannot modify this value directly.

The first button brings up an alert saying "This is an alert.". You can dispose it by pressing "OK".

The second button brings up a dialog with an editable text field, containing a copy of the value from the form's locked text field. You can modify this value, and send it back to the form, if you press "Re-Submit" and answer "Yes" to the "Are you sure?" alert.

A Calculator Example

Now let's forget our "project.l" test file for a moment, and move on to a more substantial and practical, stand-alone, example. Using what we have learned so far, we want to build a simple bignum calculator. ("bignum" because PicoLisp can do *only* bignums)

It uses a single form, a single numeric input field, and lots of buttons. It can be found in the PicoLisp distribution (e.g. under "/usr/share/picolisp/") in "misc/calc.l", together with a directly executable wrapper script "misc/calc".

To use it, change to the PicoLisp installation directory, and start it as

```
$ misc/calc
```

or call it with an absolute path, e.g.

```
$ /usr/share/picolisp/misc/calc
```

If you like to get a PicoLisp prompt for inspection, start it instead as

```
$ pil misc/calc.l -main -go +
```

Then - as before - point your browser to '`http://localhost:8080`'.

The code for the calculator logic and the GUI is rather straightforward. The entry point is the single function `calculator`. It is called directly (as described in *URL Syntax*) as the server's default URL, and implicitly in all POST requests. No further file access is needed once the calculator is running.

Note that for a production application, we inserted an allow-statement (as recommended by the *Security* chapter)

```
(allowed NIL "!calculator" "@lib.css")
```

at the beginning of "misc/calc.l". This will restrict external access to that single function.

The calculator uses three global variables, `*Init`, `*Accu` and `*Stack`. `*Init` is a boolean flag set by the operator buttons to indicate that the next digit should initialize the accumulator to zero. `*Accu` is the accumulator. It is always displayed in the numeric input field, accepts user input, and it holds the results of calculations. `*Stack` is a push-down stack, holding postponed calculations

(operators, priorities and intermediate results) with lower-priority operators, while calculations with higher-priority operators are performed.

The function `digit` is called by the digit buttons, and adds another digit to the accumulator.

The function `calc` does an actual calculation step. It pops the stack, checks for division by zero, and displays an error alert if necessary.

`operand` processes an operand button, accepting a function and a priority as arguments. It compares the priority with that in the top-of-stack element, and delays the calculation if it is less.

`finish` is used to calculate the final result.

The `calculator` function has one numeric input field, with a width of 60 characters

```
(gui '(+Var +NumField) '*Accu 60)
```

The `+Var` prefix class associates this field with the global variable `*Accu`. All changes to the field will show up in that variable, and modification of that variable's value will appear in the field.

The square root operator button has an `+Able` prefix class

```
(gui '(+Able +JS +Button) '(ge0 *Accu) (char 8730)
   '(setq *Accu (sqrt *Accu)) )
```

with an argument expression which checks that the current value in the accumulator is positive, and disables the button if otherwise.

The rest of the form is just an array (grid) of buttons, encapsulating all functionality of the calculator. The user can enter numbers into the input field, either by using the digit buttons, or by directly typing them in, and perform calculations with the operator buttons. Supported operations are addition, subtraction, multiplication, division, sign inversion, square root and power (all in bignum integer arithmetic). The `C` button just clears the accumulator, while the `A` button also clears all pending calculations.

All that in 53 lines of code!

21.3.3 Charts

Charts are virtual components, maintaining the internal representation of two-dimensional data.

Typically, these data are nested lists, database selections, or some kind of dynamically generated tabular information. Charts make it possible to view them in rows and columns (usually in HTML *tables*), scroll up and down, and associate them with their corresponding visible GUI components.

In fact, the logic to handle charts makes up a substantial part of the whole framework, with large impact on all internal mechanisms. Each GUI component must know whether it is part of a chart or not, to be able to handle its contents properly during updates and user interactions.

Let's assume we want to collect textual and numerical data. We might create a table

```
########################################################################
(app)

(action
   (html 0 "Table" "@lib.css" NIL
      (form NIL
         (<table> NIL NIL '((NIL "Text") (NIL "Number"))
            (do 4
               (<row> NIL
                  (gui '(+TextField) 20)
                  (gui '(+NumField) 10) ) ) )
         (<submit> "Save") ) ) )
########################################################################
```

with two columns "Text" and "Number", and four rows, each containing a `+TextField` and a `+NumField`.

You can enter text into the first column, and numbers into the second. Pressing the "Save" button stores these values in the components on the server (or produces an error message if a string in the second column is not a legal number).

There are two problems with this solution:

1. Though you can get at the user input for the individual fields, e.g.

   ```
   : (val> (get *Top 'gui 2))  # Value in the first row, second column
   -> 123
   ```

   there is no direct way to get the whole data structure as a single list. Instead, you have to traverse all GUI components and collect the data.

2. The user cannot input more than four rows of data, because there is no easy way to scroll down and make space for more.

A chart can handle these things:

```
######################################################################
(app)

(action
   (html 0 "Chart" "@lib.css" NIL
      (form NIL
         (gui '(+Chart) 2)                          # Inserted a +Chart
         (<table> NIL NIL '((NIL "Text") (NIL "Number"))
            (do 4
               (<row> NIL
                  (gui 1 '(+TextField) 20)          # Inserted '1'
                  (gui 2 '(+NumField) 10) ) ) )     # Inserted '2'
         (<submit> "Save") ) ) )
######################################################################
```

Note that we inserted a `+Chart` component before the GUI components which should be managed by the chart. The argument '2' tells the chart that it has to expect two columns.

Each component got an index number (here '1' and '2') as the first argument to `gui`, indicating the column into which this component should go within the chart.

Now - if you entered "a", "b" and "c" into the first, and 1, 2, and 3 into the second column - we can retrieve the chart's complete contents by sending it the `val>` message

```
: (val> (get *Top 'chart 1))  # Retrieve the value of the first chart
-> (("a" 1) ("b" 2) ("c" 3))
```

BTW, a more convenient function is `chart`

```
: (val> (chart))  # Retrieve the value of the current chart
-> (("a" 1) ("b" 2) ("c" 3))
```

`chart` can be used instead of the above construct when we want to access the "current" chart, i.e. the chart most recently processed in the current form.

Scrolling

To enable scrolling, let's also insert two buttons. We use the pre-defined classes `+UpButton` and `+DnButton`

```
########################################################################
(app)

(action
   (html 0 "Scrollable Chart" "@lib.css" NIL
      (form NIL
         (gui '(+Chart) 2)
         (<table> NIL NIL '((NIL "Text") (NIL "Number"))
            (do 4
               (<row> NIL
                  (gui 1 '(+TextField) 20)
                  (gui 2 '(+NumField) 10) ) ) )
         (gui '(+UpButton) 1)                    # Inserted two buttons
         (gui '(+DnButton) 1)
         (----)
         (<submit> "Save") ) ) )
########################################################################
```

to scroll down and up a single (argument '1') line at a time.

Now it is possible to enter a few rows of data, scroll down, and continue. It is not necessary (except in the beginning, when the scroll buttons are still disabled) to press the "Save" button, because **any** button in the form will send changes to the server's internal structures before any action is performed.

Put and Get Functions

As we said, a chart is a virtual component to edit two-dimensional data. Therefore, a chart's native data format is a list of lists: Each sublist represents a single row of data, and each element of a row corresponds to a single GUI component.

In the example above, we saw a row like

```
("a" 1)
```

being mapped to

```
(gui 1 '(+TextField) 20)
(gui 2 '(+NumField) 10)
```

Quite often, however, such a one-to-one relationship is not desired. The internal data structures may have to be presented in a different form to the user, and user input may need conversion to an internal representation.

For that, a chart accepts - in addition to the "number of columns" argument - two optional function arguments. The first function is invoked to 'put' the internal representation into the GUI components, and the second to 'get' data from the GUI into the internal representation.

A typical example is a chart displaying customers in a database. While the internal representation is a (one-dimensional) list of customer objects, 'put' expands each object to a list with, say, the customer's first and second name, telephone number, address and so on. When the user enters a customer's name, 'get' locates the matching object in the database and stores it in the internal representation. In the following, 'put' will in turn expand it to the GUI.

For now, let's stick with a simpler example: A chart that holds just a list of numbers, but expands in the GUI to show also a textual form of each number (in German).

```
#########################################################################
(app)

(load "@lib/zahlwort.l")

(action
   (html 0 "Numerals" "@lib.css" NIL
      (form NIL
         (gui '(+Init +Chart) (1 5 7) 2
            '((N) (list N (zahlwort N)))
            car )
         (<table> NIL NIL '((NIL "Numeral") (NIL "German"))
            (do 4
               (<row> NIL
                  (gui 1 '(+NumField) 9)
                  (gui 2 '(+Lock +TextField) 90) ) ) )
         (gui '(+UpButton) 1)
         (gui '(+DnButton) 1)
         (----)
         (<submit> "Save") ) ) )
#########################################################################
```

"@lib/zahlwort.l" defines the utility function `zahlwort`, which is required later by the 'put' function. `zahlwort` accepts a number and returns its wording in German.

Now look at the code

```
(gui '(+Init +Chart) (1 5 7) 2
   '((N) (list N (zahlwort N)))
   car )
```

We prefix the `+Chart` class with `+Init`, and pass it a list of numbers `(1 5 7)` for the initial value of the chart. Then, following the '2' (the chart has two columns), we pass a 'put' function

```
'((N) (list N (zahlwort N)))
```

which takes a number and returns a list of that number and its wording, and a 'get' function

```
car )
```

which in turn accepts such a list and returns a number, which happens to be the list's first element.

You can see from this example that 'get' is the inverse function of 'put'. 'get' can be omitted, however, if the chart is read-only (contains no (or only locked) input fields).

The field in the second column

```
(gui 2 '(+Lock +TextField) 90) ) ) )
```

is locked, because it displays the text generated by 'put', and is not supposed to accept any user input.

When you start up this form in your browser, you'll see three pre-filled lines with "1/eins", "5/fünf" and "7/sieben", according to the `+Init` argument `(1 5 7)`. Typing a number somewhere into the first column, and pressing ENTER or one of the buttons, will show a suitable text in the second column.

21.4 GUI Classes

In previous chapters we saw examples of GUI classes like `+TextField`, `+NumField` or `+Button`, often in combination with prefix classes like `+Lock`, `+Init` or `+Able`. Now we take a broader look at the whole hierarchy, and try more examples.

The abstract class `+gui` is the base of all GUI classes. A live view of the class hierarchy can be obtained with the `dep` ("dependencies") function:

```
: (dep '+gui)
+gui
   +JsField
   +Button
      +UpButton
      +PickButton
         +DstButton
      +ClrButton
      +ChoButton
         +Choice
      +GoButton
      +BubbleButton
      +DelRowButton
      +ShowButton
      +DnButton
   +Img
   +field
      +Checkbox
      +TextField
         +FileField
         +ClassField
         +numField
            +NumField
            +FixField
         +BlobField
         +DateField
         +SymField
         +UpField
         +MailField
         +SexField
         +AtomField
         +PwField
         +ListTextField
         +LinesField
         +TelField
         +TimeField
         +HttpField
      +Radio
-> +gui
```

We see, for example, that `+DnButton` is a subclass of `+Button`, which in turn is a subclass of `+gui`. Inspecting `+DnButton` directly

```
: (dep '+DnButton)
   +Tiny
   +Rid
   +JS
   +Able
      +gui
   +Button
+DnButton
-> +DnButton
```

shows that `+DnButton` inherits from `+Tiny`, `+Rid`, `+Able` and `+Button`. The actual definition of `+DnButton` can be found in "@lib/form.l"

```
(class +DnButton +Tiny +Rid +JS +Able +Button)
...
```

In general, "@lib/form.l" is the ultimate reference to the framework, and should be freely consulted.

21.4.1 Input Fields

Input fields implement the visual display of application data, and allow

- when enabled - input and modification of these data.

On the HTML level, they can take the form of

- Normal text input fields
- Textareas
- Checkboxes
- Drop-down selections
- Password fields
- HTML links
- Plain HTML text

Except for checkboxes, which are implemented by the *Checkbox* class, all these HTML representations are generated by `+TextField` and its content-specific subclasses like `+NumField`, `+DateField` etc. Their actual appearance (as one of the above forms) depends on their arguments:

We saw already "normal" text fields. They are created with a single numeric argument. This example creates an editable field with a width of 10 characters:

```
(gui '(+TextField) 10)
```

If you supply a second numeric for the line count ('4' in this case), you'll get a text area:

```
(gui '(+TextField) 10 4)
```

Supplying a list of values instead of a count yields a drop-down selection (combo box):

```
(gui '(+TextField) '("Value 1" "Value 2" "Value 3"))
```

In addition to these arguments, you can pass a string. Then the field is created with a label:

```
(gui '(+TextField) 10 "Plain")
(gui '(+TextField) 10 4 "Text Area")
(gui '(+TextField) '("Value 1" "Value 2" "Value 3") "Selection")
```

Finally, without any arguments, the field will appear as a plain HTML text:

```
(gui '(+TextField))
```

This makes mainly sense in combination with prefix classes like `+Var` and `+Obj`, to manage the contents of these fields, and achieve special behavior as HTML links or scrollable chart values.

Numeric Input Fields

A `+NumField` returns a number from its `val>` method, and accepts a number for its `set>` method. It issues an error message when user input cannot be converted to a number.

Large numbers are shown with a thousands-separator, as determined by the current locale.

```
########################################################################
(app)

(action
   (html 0 "+NumField" "@lib.css" NIL
      (form NIL
         (gui '(+NumField) 10)
         (gui '(+JS +Button) "Print value"
            '(msg (val> (: home gui 1))) )
         (gui '(+JS +Button) "Set to 123"
            '(set> (: home gui 1) 123) ) ) ) )
########################################################################
```

A `+FixField` needs an additional scale factor argument, and accepts/returns scaled fixpoint numbers.

The decimal separator is determined by the current locale.

```
########################################################################
(app)

(action
   (html 0 "+FixField" "@lib.css" NIL
      (form NIL
         (gui '(+FixField) 3 10)
         (gui '(+JS +Button) "Print value"
            '(msg (format (val> (: home gui 1)) 3)) )
         (gui '(+JS +Button) "Set to 123.456"
            '(set> (: home gui 1) 123456) ) ) ) )
########################################################################
```

Time & Date

A `+DateField` accepts and returns a `date` value.

```
########################################################################
(app)

(action
   (html 0 "+DateField" "@lib.css" NIL
      (form NIL
         (gui '(+DateField) 10)
         (gui '(+JS +Button) "Print value"
            '(msg (datStr (val> (: home gui 1)))) )
         (gui '(+JS +Button) "Set to \"today\""
            '(set> (: home gui 1) (date)) ) ) ) )
########################################################################
```

The format displayed to - and entered by - the user depends on the current locale (see `datStr` and `expDat`). You can change it, for example to

```
: (locale "DE" "de")
-> NIL
```

If no locale is set, the format is YYYY-MM-DD. Some pre-defined locales use patterns like DD.MM.YYYY (DE), YYYY/MM/DD (JP), DD/MM/YYYY (UK), or MM/DD/YYYY (US).

An error is issued when user input does not match the current locale's date format.

Independent from the locale setting, a `+DateField` tries to expand abbreviated input from the user. A small number is taken as that day of the current month, larger numbers expand to day and month, or to day, month and year:

- "7" gives the 7th of the current month
- "031" or "0301" give the 3rd of January of the current year
- "311" or "3101" give the 31st of January of the current year
- "0311" gives the 3rd of November of the current year
- "01023" or "010203" give the first of February in the year 2003
- and so on

Similar is the `+TimeField`. It accepts and returns a `time` value.

```
########################################################################
(app)

(action
   (html 0 "+TimeField" "@lib.css" NIL
      (form NIL
         (gui '(+TimeField) 8)
         (gui '(+JS +Button) "Print value"
            '(msg (tim$ (val> (: home gui 1)))) )
         (gui '(+JS +Button) "Set to \"now\""
            '(set> (: home gui 1) (time)) ) ) ) )
########################################################################
```

When the field width is '8', like in this example, time is displayed in the format `HH:MM:SS`. Another possible value would be '5', causing `+TimeField` to display its value as `HH:MM`.

An error is issued when user input cannot be converted to a time value.

The user may omit the colons. If he inputs just a small number, it should be between '0' and '23', and will be taken as a full hour. '125' expands to "12:05", '124517' to "12:45:17", and so on.

Telephone Numbers

Telephone numbers are represented internally by the country code (without a leading plus sign or zero) followed by the local phone number (ideally separated by spaces) and the phone extension (ideally separated by a hyphen). The exact format of the phone number string is not enforced by the GUI, but further processing (e.g. database searches) normally uses `fold` for better reproducibility.

To display a phone number, `+TelField` replaces the country code with a single zero if it is the country code of the current locale, or prepends it with a plus sign if it is a foreign country (see `telStr`).

For user input, a plus sign or a double zero is simply dropped, while a single leading zero is replaced with the current locale's country code (see `expTel`).

```
########################################################################
(app)
(locale "DE" "de")

(action
   (html 0 "+TelField" "@lib.css" NIL
      (form NIL
         (gui '(+TelField) 20)
         (gui '(+JS +Button) "Print value"
            '(msg (val> (: home gui 1))) )
         (gui '(+JS +Button) "Set to \"49 1234 5678-0\""
            '(set> (: home gui 1) "49 1234 5678-0") ) ) ) )
########################################################################
```

Checkboxes

A `+Checkbox` is straightforward. User interaction is restricted to clicking it on and off. It accepts boolean (`NIL` or non `NIL` values, and returns `T` or `NIL`.

```
########################################################################
(app)

(action
   (html 0 "+Checkbox" "@lib.css" NIL
      (form NIL
         (gui '(+Checkbox))
         (gui '(+JS +Button) "Print value"
            '(msg (val> (: home gui 1))) )
         (gui '(+JS +Button) "On"
            '(set> (: home gui 1) T) )
         (gui '(+JS +Button) "Off"
            '(set> (: home gui 1) NIL) ) ) ) )
########################################################################
```

21.4.2 Field Prefix Classes

A big part of this framework's power is owed to the combinatorial flexibility of prefix classes for GUI- and DB-objects. They allow to surgically override individual methods in the inheritance tree, and can be combined in various ways to achieve any desired behavior.

Technically, there is nothing special about prefix classes. They are just normal classes. They are called "prefix" because they are intended to be written *before* other classes in a class's or object's list of superclasses.

Usually they take their own arguments for their `T` method from the list of arguments to the `gui` function.

Initialization

`+Init` overrides the `init>` method for that component. The `init>` message is sent to a `+gui` component when the page is loaded for the first time (during a GET request). `+Init` takes an expression for the initial value of that field.

```
(gui '(+Init +TextField) "This is the initial text" 30)
```

Other classes which automatically give a value to a field are `+Var` (linking the field to a variable) and `+E/R` (linking the field to a database entity/relation).

`+Cue` can be used, for example in "mandatory" fields, to give a hint to the user about what he is supposed to enter. It will display the argument value, in angular brackets, if and only if the field's value is `NIL`, and the `val>` method will return `NIL` despite the fact that this value is displayed.

Cause an empty field to display "<Please enter some text here>":

```
(gui '(+Cue +TextField) "Please enter some text here" 30)
```

Disabling and Enabling

An important feature of an interactive GUI is the context-sensitive disabling and enabling of individual components, or of a whole form.

The `+Able` prefix class takes an argument expression, and disables the component if this expression returns `NIL`. We saw an example for its usage already in the *square root button* of the calculator example. Or, for illustration purposes, imagine a button which is supposed to be enabled only after Christmas

```
(gui '(+Able +Button)
   '(>= (cdr (date (date))) (12 24))
   "Close this year"
   '(endOfYearProcessing) )
```

or a password field that is disabled as long as somebody is logged in

```
(gui '(+Able +PwField) '(not *Login) 10 "Password")
```

A special case is the `+Lock` prefix, which permanently and unconditionally disables a component. It takes no arguments

```
(gui '(+Lock +NumField) 10 "Count")
```

('10' and "Count" are for the `+NumField`), and creates a read-only field.

The whole form can be disabled by calling `disable` with a non `NIL` argument. This affects all components in this form. Staying with the above example, we can make the form read-only until Christmas

```
(form NIL
   (disable (> (12 24) (cdr (date (date)))))  # Disable whole form
   (gui ..)
   .. )
```

Even in a completely disabled form, however, it is often necessary to re-enable certain components, as they are needed for navigation, scrolling, or other activities which don't affect the contents of the form. This is done by prefixing these fields with `+Rid` (i.e. getting "rid" of the lock).

```
(form NIL
   (disable (> (12 24) (cdr (date (date)))))
   (gui ..)
   ..
   (gui '(+Rid +Button) ..)  # Button is enabled despite the disabled form
   .. )
```

Formatting

GUI prefix classes allow a fine-grained control of how values are stored in - and retrieved from - components. As in predefined classes like `+NumField` or `+DateField`, they override the `set>` and/or `val>` methods.

`+Set` takes an argument function which is called whenever that field is set to some value. To convert all user input to upper case

```
(gui '(+Set +TextField) uppc 30)
```

`+Val` is the complement to `+Set`. It takes a function which is called whenever the field's value is retrieved. To return the square of a field's value

```
(gui '(+Val +NumField) '((N) (* N N)) 10)
```

`+Fmt` is just a combination of `+Set` and `+Val`, and takes two functional arguments. This example will display upper case characters, while returning lower case characters internally

```
(gui '(+Fmt +TextField) uppc lowc 30)
```

`+Map` does (like `+Fmt`) a two-way translation. It uses a list of cons pairs for a linear lookup, where the CARs represent the displayed values which are internally mapped to the values in the CDRs. If a value is not found in this list during `set>` or `val>`, it is passed through unchanged.

Normally, `+Map` is used in combination with the combo box incarnation of text fields (see *Input Fields*). This example displays "One", "Two" and "Three" to the user, but returns a number 1, 2 or 3 internally

```
########################################################################
(app)

(action
   (html 0 "+Map" "@lib.css" NIL
      (form NIL
         (gui '(+Map +TextField)
            '(("One" . 1) ("Two" . 2) ("Three" . 3))
            '("One" "Two" "Three") )
         (gui '(+Button) "Print"
            '(msg (val> (field -1))) ) ) ) )
########################################################################
```

Side Effects

Whenever a button is pressed in the GUI, any changes caused by `action` in the current environment (e.g. the database or application state) need to be reflected in the corresponding GUI fields. For that, the `upd>` message is sent to all components. Each component then takes appropriate measures (e.g. refresh from database objects, load values from variables, or calculate a new value) to update its value.

While the `upd>` method is mainly used internally, it can be overridden in existing classes via the `+Upd` prefix class. Let's print updated values to standard error

```
########################################################################
(app)
(default *Number 0)

(action
   (html 0 "+Upd" "@lib.css" NIL
      (form NIL
         (gui '(+Upd +Var +NumField)
            '(prog (extra) (msg *Number))
            '*Number 8 )
         (gui '(+JS +Button) "Increment"
            '(inc '*Number) ) ) ) )
########################################################################
```

Validation

To allow automatic validation of user input, the `chk>` message is sent to all components at appropriate times. The corresponding method should return `NIL` if the value is all right, or a string describing the error otherwise.

Many of the built-in classes have a `chk>` method. The `+NumField` class checks for legal numeric input, or the `+DateField` for a valid calendar date.

An on-the-fly check can be implemented with the `+Chk` prefix class. The following code only accepts numbers not bigger than 9: The `or` expression first delegates the check to the main `+NumField` class, and

- if it does not give an error - returns an error string when the

current value is greater than 9.

```
########################################################################
(app)

(action
   (html 0 "+Chk" "@lib.css" NIL
      (form NIL
         (gui '(+Chk +NumField)
            '(or
               (extra)
               (and (> (val> This) 9) "Number too big") )
            12 )
         (gui '(+JS +Button) "Print"
            '(msg (val> (field -1))) ) ) ) )
########################################################################
```

A more direct kind of validation is built-in via the `+Limit` class. It controls the `maxlength` attribute of the generated HTML input field component. Thus, it is impossible to type to more characters than allowed into the field.

```
######################################################################
(app)

(action
   (html 0 "+Limit" "@lib.css" NIL
      (form NIL
         (gui '(+Limit +TextField) 4 8)
         (gui '(+JS +Button) "Print"
            '(msg (val> (field -1))) ) ) ) )
######################################################################
```

Data Linkage

Although `set>` and `val>` are the official methods to get a value in and out of a GUI component, they are not very often used explicitly. Instead, components are directly linked to internal Lisp data structures, which are usually either variables or database objects.

The `+Var` prefix class takes a variable (described as the `var` data type - either a symbol or a cell - in the *Function Reference*). In the following example, we initialize a global variable with the value "abc", and let a `+TextField` operate on it. The "Print" button can be used to display its current value.

```
######################################################################
(app)

(setq *TextVariable "abc")

(action
   (html 0 "+Var" "@lib.css" NIL
      (form NIL
         (gui '(+Var +TextField) '*TextVariable 8)
         (gui '(+JS +Button) "Print"
            '(msg *TextVariable) ) ) ) )
######################################################################
```

`+E/R` takes an entity/relation specification. This is a cell, with a relation in its CAR (e.g. `nm`, for an object's name), and an expression in its CDR (typically `(: home obj)`, the object stored in the `obj` property of the current form).

For an isolated, simple example, we create a temporary database, and access the `nr` and `nm` properties of an object stored in a global variable `*Obj`.

```
######################################################################
(when (app)                # On start of session
   (class +Tst +Entity)    # Define data model
   (rel nr (+Number))      # with a number
   (rel nm (+String))      # and a string
   (pool (tmp "db"))       # Create temporary DB
   (setq *Obj              # and a single object
      (new! '(+Tst) 'nr 1 'nm "New Object") ) )

(action
   (html 0 "+E/R" "@lib.css" NIL
      (form NIL
         (gui '(+E/R +NumField) '(nr . *Obj) 8)    # Linkage to 'nr'
         (gui '(+E/R +TextField) '(nm . *Obj) 20)  # Linkage to 'nm'
         (gui '(+JS +Button) "Show"                # Show the object
            '(out 2 (show *Obj)) ) ) ) )           # on standard error
######################################################################
```

21.4.3 Buttons

Buttons are, as explained in *Control Flow*, the only way (via POST requests) for an application to communicate with the server.

Basically, a `+Button` takes

- a label, which may be either a string or the name of an image file
- an optional alternative label, shown when the button is disabled
- and an executable expression.

Here is a minimal button, with just a label and an expression:

```
(gui '(+Button) "Label" '(doSomething))
```

And this is a button displaying different labels, depending on the state:

```
(gui '(+Button) "Enabled" "Disabled" '(doSomething))
```

To show an image instead of plain text, the label(s) must be preceeded by the `T` symbol:

```
(gui '(+Button) T "img/enabled.png" "img/disabled.png" '(doSomething))
```

The expression will be executed during `action` handling (see *Action Forms*), when this button was pressed.

Like other components, buttons can be extended and combined with prefix classes, and a variety of predefined classes and class combinations are available.

Dialog Buttons

Buttons are essential for the handling of *alerts and dialogs*. Besides buttons for normal functions, like *scrolling* in charts or other *side effects*, special buttons exist which can *close* an alert or dialog in addition to doing their principal job.

Such buttons are usually subclasses of `+Close`, and most of them can be called easily with ready-made functions like `closeButton`, `cancelButton`, `yesButton` or `noButton`. We saw a few examples in *Alerts and Dialogs*.

Active JavaScript

When a button inherits from the `+JS` class (and JavaScript is enabled in the browser), that button will possibly show a much faster response in its action.

The reason is that the activation of a `+JS` button will - instead of doing a normal POST - first try to send only the contents of all GUI components via an XMLHttpRequest to the server, and receive the updated values in response. This avoids the flicker caused by reloading and rendering of the whole page, is much faster, and also does not jump to the beginning of the page if it is larger than the browser window. The effect is especially noticeable while scrolling in charts.

Only if this fails, for example because an error message was issued, or a dialog popped up, it will fall back, and the form will be POSTed in the normal way.

Thus it makes no sense to use the `+JS` prefix for buttons that cause a change of the HTML code, open a dialog, or jump to another page. In such cases, overall performance will even be worse, because the XMLHttpRequest is tried first (but in vain).

When JavaScript is disabled int the browser, the XMLHttpRequest will not be tried at all. The form will be fully usable, though, with identical functionality and behavior, just a bit slower and not so smooth.

21.5 A Minimal Complete Application

The PicoLisp release includes in the "app/" directory a minimal, yet complete reference application. This application is typical, in the sense that it implements many of the techniques described in this document, and it can be easily modified and extended. In fact, we use it as templates for our own production application development.

It is a kind of simplified ERP system, containing customers/suppliers, products (items), orders, and other data. The order input form performs live updates of customer and product selections, price, inventory and totals calculations, and generates on-the-fly PDF documents. Fine-grained access permissions are controlled via users, roles and permissions. It comes localized in six languages (English, Spanish, German, Norwegian, Russian and Japanese), with some initial data and two sample reports.

21.5.1 Getting Started

For a global installation (see *Installation*), please create a symbolic link to the place where the program files are installed. This is necessary because the application needs read/write access to the current working directory (for the database and other runtime data).

```
$ ln -s /usr/share/picolisp/app
```

As ever, you may start up the application in debugging mode

```
$ pil app/main.l -main -go +
```

or in (non-debug) production mode

```
$ pil app/main.l -main -go -wait
```

and go to '`http://localhost:8080`' with your browser. You can login as user "admin", with password "admin". The demo data contain several other users, but those are more restricted in their role permissions.

Another possibility is to try the online version of this application at app.7fach.de.

Localization

Before or after you logged in, you can select another language, and click on the "Change" button. This will effect all GUI components (though not text from the database), and also the numeric, date and telephone number formats.

Navigation

The navigation menu on the left side shows two items "Home" and "logout", and three submenus "Data", "Report" and "System".

Both "Home" and "logout" bring you back to the initial login form. Use "logout" if you want to switch to another user (say, for another set of permissions), and - more important - before you close your browser, to release possible locks and process resources on the server.

The "Data" submenu gives access to application specific data entry and maintenance: Orders, product items, customers and suppliers. The "Report" submenu contains two simple inventory and sales reports. And the "System" submenu leads to role and user administration.

You can open and close each submenu individually. Keeping more than one submenu open at a time lets you switch rapidly between different parts of the application.

The currently active menu item is indicated by a highlighted list style (no matter whether you arrived at this page directly via the menu or by clicking on a link somewhere else).

Choosing Objects

Each item in the "Data" or "System" submenu opens a search dialog for that class of entities. You can specify a search pattern, press the top right "Search" button (or just ENTER), and scroll through the list of results.

While the "Role" and "User" entities present simple dialogs (searching just by name), other entities can be searched by a variety of criteria. In those cases, a "Reset" button clears the contents of the whole dialog. A new object can be created with bottom right "New" button.

In any case, the first column will contain either a "@"-link (to jump to that object) or a "@"-button (to insert a reference to that object into the current form).

By default, the search will list all database objects with an attribute value greater than or equal to the search criterion. The comparison is done arithmetically for numbers, and alphabetically (case sensitive!) for text. This means, if you type "Free" in the "City" field of the "Customer/Supplier" dialog, the value of "Freetown" will be matched. On the other hand, an entry of "free" or "town" will yield no hits.

Some search fields, however, show a different behavior depending on the application:

- The names of persons, companies or products allow a tolerant search, matching either a slightly misspelled name ("Mhler" instead of "Miller") or a substring ("Oaks" will match "Seven Oaks Ltd.").
- The search field may specify an upper instead of a lower limit, resulting in a search for database objects with an attribute value less than or equal to the search criterion. This is useful, for example in the "Order" dialog, to list orders according to their number or date, by starting with the newest then and going backwards.

Using the bottom left scroll buttons, you can scroll through the result list without limit. Clicking on a link will bring up the corresponding object. Be careful here to select the right column: Some dialogs (those for "Item" and "Order") also provide links for related entities (e.g. "Supplier").

Editing

A database object is usually displayed in its own individual form, which is determined by its entity class.

The basic layout should be consistent for all classes: Below the heading (which is usually the same as the invoking menu item) is the object's identifier (name, number, etc.), and then a row with an "Edit" button on the left, and "Delete" button, a "Select" button and two navigation links on the right side.

The form is brought up initially in read-only mode. This is necessary to prevent more than one user from modifying an object at the same time (and contrary to the previous PicoLisp Java frameworks, where this was not a problem because all changes were immediately reflected in the GUIs of other users).

So if you want to modify an object, you have to gain exclusive access by clicking on the "Edit" button. The form will be enabled, and the "Edit" button changes to "Done". Should any other user already have reserved this object, you will see a message telling his name and process ID.

An exception to this are objects that were just created with "New". They will automatically be reserved for you, and the "Edit" button will show up as "Done".

The "Delete" button pops up an alert, asking for confirmation. If the object is indeed deleted, this button changes to "Restore" and allows to undelete the object. Note that objects are never completely deleted from the database as long as there are any references from other objects. When a "deleted" object is shown, its identifier appears in square brackets.

The "Select" button (re-)displays the search dialog for this class of entities. The search criteria are preserved between invocations of each dialog, so that you can conveniently browse objects in this context.

The navigation links, pointing left and right, serve a similar purpose. They let you step sequentially through all objects of this class, in the order of the identifier's index.

Other buttons, depending on the entity, are usually arranged at the bottom of the form. The bottom rightmost one should always be another "Edit" / "Done" button.

As we said in the chapter on *Scrolling*, any button in the form will save changes to the underlying data model. As a special case, however, the "Done" button releases the object and reverts to "Edit". Besides this, the edit mode will also cease as soon as another object is displayed, be it by clicking on an object link (the pencil icon), the top right navigation links, or a link in a search dialog.

Buttons vs. Links

The only way to interact with a HTTP-based application server is to click either on a HTML link, or on a submit button (see also *Control Flow*). It is essential to understand the different effects of such a click on data entered or modified in the current form.

- A click on a link will leave or reload the page. Changes are discarded.
- A click on a button will commit changes, and perform the associated action.

For that reason the layout design should clearly differentiate between links and buttons. Image buttons are not a good idea when in other places images are used for links. The standard button components should be preferred; they are usually rendered by the browser in a non-ambiguous three-dimensional look and feel.

Note that if JavaScript is enabled in the browser, changes will be automatically committed to the server.

The enabled or disabled state of a button is an integral part of the application logic. It must be indicated to the user with appropriate styles.

21.5.2 The Data Model

Source Code

```
# 21jul11abu
# (c) Software Lab. Alexander Burger

### Entity/Relations ###
#
#           nr    nm                    nr    nm               nm
#            |    |                     |    |                |
#          +-*----*-+                 +-*----*-+           +--*-----+
#          |        |             sup |        |           |        |
#    str --*  CuSu  O-----------------*  Item  *-- inv     |  Role  @-- perm
#          |        |                 |        |           |        |
#          +-*-*--O-+                 +----O---+           +----@---+
#            | |  |                       |                   | usr
#   nm  tel -+ |  |                       |                   |
#    |         |  |                       | itm               | role
#  +-*-----+   |  |   +-------+        +---*---+           +---*---+
#  |       |   |  |   |       |    ord |       |           |        |
#  |  Sal  +---+  +---*  Ord  @--------*  Pos  |      nm --*  User  *-- pw
#  |       |      cus |       | pos    |       |           |        |
#  +-*---*-+          +-*---*-+        +-*---*-+           +--------+
#    |   |              |   |            |   |
#   hi   sex           nr  dat          pr   cnt


(extend +Role)

(dm url> (Tab)
   (and (may RoleAdmin) (list "app/role.l" '*ID This)) )
```

```
(extend +User)
(rel nam (+String))                     # Full Name
(rel tel (+String))                     # Phone
(rel em (+String))                      # EMail

(dm url> (Tab)
   (and (may UserAdmin) (list "app/user.l" '*ID This)) )


# Salutation
(class +Sal +Entity)
(rel nm (+Key +String))                 # Salutation
(rel hi (+String))                      # Greeting
(rel sex (+Any))                        # T:male, 0:female

(dm url> (Tab)
   (and (may Customer) (list "app/sal.l" '*ID This)) )

(dm hi> (Nm)
   (or (text (: hi) Nm) ,"Dear Sir or Madam,") )


# Customer/Supplier
(class +CuSu +Entity)
(rel nr (+Need +Key +Number))           # Customer/Supplier Number
(rel sal (+Link) (+Sal))                # Salutation
(rel nm (+Sn +Idx +String))             # Name
(rel nm2 (+String))                     # Name 2
(rel str (+String))                     # Street
(rel plz (+Ref +String))                # Zip
(rel ort (+Fold +Idx +String))          # City
(rel tel (+Fold +Ref +String))          # Phone
(rel fax (+String))                     # Fax
(rel mob (+Fold +Ref +String))          # Mobile
(rel em (+String))                      # EMail
(rel txt (+Blob))                       # Memo

(dm url> (Tab)
   (and (may Customer) (list "app/cusu.l"  '*Tab Tab  '*ID This)) )

(dm check> ()
   (make
      (or (: nr) (link ,"No customer number"))
      (or (: nm) (link ,"No customer name"))
      (unless (and (: str) (: plz) (: ort))
         (link ,"Incomplete customer address") ) ) )
```

```
# Item
(class +Item +Entity)
(rel nr (+Need +Key +Number))          # Item Number
(rel nm (+Fold +Idx +String))          # Item Description
(rel sup (+Ref +Link) NIL (+CuSu))     # Supplier
(rel inv (+Number))                    # Inventory
(rel pr (+Ref +Number) NIL 2)          # Price
(rel txt (+Blob))                      # Memo
(rel jpg (+Blob))                      # Picture

(dm url> (Tab)
   (and (may Item) (list "app/item.l" '*ID This)) )

(dm cnt> ()
   (-
      (or (: inv) 0)
      (sum '((This) (: cnt))
         (collect 'itm '+Pos This) ) ) )

(dm check> ()
   (make
      (or (: nr) (link ,"No item number"))
      (or (: nm) (link ,"No item description")) ) )


# Order
(class +Ord +Entity)
(rel nr (+Need +Key +Number))          # Order Number
(rel dat (+Need +Ref +Date))           # Order date
(rel cus (+Ref +Link) NIL (+CuSu))     # Customer
(rel pos (+List +Joint) ord (+Pos))    # Positions

(dm lose> ()
   (mapc 'lose> (: pos))
   (super) )

(dm url> (Tab)
   (and (may Order) (list "app/ord.l" '*ID This)) )

(dm sum> ()
   (sum 'sum> (: pos)) )

(dm check> ()
   (make
      (or (: nr) (link ,"No order number"))
      (or (: dat) (link ,"No order date"))
      (if (: cus)
         (chain (check> @))
         (link ,"No customer") )
      (if (: pos)
         (chain (mapcan 'check> @))
         (link ,"No positions") ) ) )
```

```
(class +Pos +Entity)
(rel ord (+Dep +Joint)                  # Order
   (itm)
   pos (+Ord) )
(rel itm (+Ref +Link) NIL (+Item))      # Item
(rel pr (+Number) 2)                    # Price
(rel cnt (+Number))                     # Quantity

(dm sum> ()
   (* (: pr) (: cnt)) )

(dm check> ()
   (make
      (if (: itm)
         (chain (check> @))
         (link ,"Position without item") )
      (or (: pr) (link ,"Position without price"))
      (or (: cnt) (link ,"Position without quantity")) ) )


# Database sizes
(dbs
   (3 +Role +User +Sal)                          # 512 Prevalent objects
   (0 +Pos)                                      # A:64 Tiny objects
   (1 +Item +Ord)                                # B:128 Small objects
   (2 +CuSu)                                     # C:256 Normal objects
   (2 (+Role nm) (+User nm) (+Sal nm))           # D:256 Small indexes
   (4 (+CuSu nr plz tel mob))                    # E:1024 Normal indexes
   (4 (+CuSu nm))                                # F:1024
   (4 (+CuSu ort))                               # G:1024
   (4 (+Item nr sup pr))                         # H:1024
   (4 (+Item nm))                                # I:1024
   (4 (+Ord nr dat cus))                         # J:1024
   (4 (+Pos itm)) )                              # K:1024

# vi:et:ts=3:sw=3
```

Discussion

The data model for this mini application consists of only six entity classes (see the E/R diagram at the beginning of "app/er.l"):

- The three main entities are `+CuSu` (Customer/Supplier), `+Item` (Product Item) and `+Ord` (Order).

- A `+Pos` object is a single position in an order.
- `+Role` and `+User` objects are needed for authentication and authorization.

The classes `+Role` and `+User` are defined in "@lib/adm.l". A `+Role` has a name, a list of permissions, and a list of users assigned to this role. A `+User` has a name, a password and a role.

In "app/er.l", the `+Role` class is extended to define an `url>` method for it. Any object whose class has such a method is able to display itself in the GUI. In this case, the file "app/role.l" will be loaded - with the global variable `*ID` pointing to it - whenever an HTML link to this role object is activated.

The `+User` class is also extended. In addition to the login name, a full name, telephone number and email address is declared. And, of course, the ubiquitous `url>` method.

The application logic is centered around orders. An order has a number, a date, a customer (an instance of `+CuSu`) and a list of positions (`+Pos` objects). The `sum>` method calculates the total amount of this order.

Each position has an `+Item` object, a price and a quantity. The price in the position overrides the default price from the item.

Each item has a number, a description, a supplier (also an instance of `+CuSu`), an inventory count (the number of these items that were counted at the last inventory taking), and a price. The `cnt>` method calculates the current stock of this item as the difference of the inventory and the sold item counts.

The call to `dbs` at the end of "app/er.l" configures the physical database storage. Each of the supplied lists has a number in its CAR which determines the block size as (64 ¡¡ N) of the corresponding database file. The CDR says that the instances of this class (if the element is a class symbol) or the tree nodes (if the element is a list of a class symbol and a property name) are to be placed into that file. This allows for some optimizations in the database layout.

21.5.3 Usage

When you are connected to the application (see *Getting Started*) you might try to do some "real" work with it. Via the "Data" menu (see *Navigation*) you can create or modify customers, suppliers, items and orders, and produce simple overviews via the "Report" menu.

Customer/Supplier

Source Code

```
# 05nov09abu
# (c) Software Lab. Alexander Burger

(must "Customer/Supplier" Customer)

(menu ,"Customer/Supplier"
   (ifn *ID
      (prog
         (<h3> NIL ,"Select" " " ,"Customer/Supplier")
         (form 'dialog (choCuSu)) )
      (<h3> NIL ,"Customer/Supplier")
      (form NIL
         (<h2> NIL (<id> (: nr) " -- " (: nm)))
         (panel T (pack ,"Customer/Supplier" " @1") '(may Delete) '(choCuSu) 'nr '+CuSu)
         (<hr>)
         (<tab>
            (,"Name"
               (<grid> 3
                  ,"Number" NIL (gui '(+E/R +NumField) '(nr : home obj) 10)
                  ,"Salutation"
                  (gui '(+Hint) ,"Salutation"
                     '(mapcar '((This) (cons (: nm) This)) (collect 'nm '+Sal)) )
                  (gui '(+Hint2 +E/R +Obj +TextField) '(sal : home obj) '(nm +Sal) 20)
                  ,"Name" NIL (gui '(+E/R +Cue +TextField) '(nm : home obj) ,"Name" 40)
                  ,"Name 2" NIL (gui '(+E/R +TextField) '(nm2 : home obj) 40) ) )
            (,"Address"
               (<grid> 2
                  ,"Street" (gui '(+E/R +TextField) '(str : home obj) 40)
                  NIL NIL
                  ,"Zip" (gui '(+E/R +TextField) '(plz : home obj) 10)
                  ,"City" (gui '(+E/R +TextField) '(ort : home obj) 40) ) )
            (,"Contact"
               (<grid> 2
                  ,"Phone" (gui '(+E/R +TelField) '(tel : home obj) 40)
                  ,"Fax" (gui '(+E/R +TelField) '(fax : home obj) 40)
                  ,"Mobile" (gui '(+E/R +TelField) '(mob : home obj) 40)
                  ,"EMail" (gui '(+E/R +MailField) '(em : home obj) 40) ) )
            ((pack (and (: obj txt) "@ ") ,"Memo")
               (gui '(+BlobField) '(txt : home obj) 60 8) ) )
         (<hr>)
         (<spread> NIL (editButton T)) ) ) )

# vi:et:ts=3:sw=3
```

Discussion

The Customer/Supplier search dialog (`choCuSu` in "app/gui.l") supports a lot of search criteria. These become necessary when the database contains a large number of customers, and can filter by zip, by phone number prefixes, and so on.

In addition to the basic layout (see *Editing*), the form is divided into four separate tabs. Splitting a form into several tabs helps to reduce traffic, with possibly better GUI response. In this case, four tabs are perhaps overkill, but ok for demonstration purposes, and they leave room for extensions.

Be aware that when data were modified in one of the tabs, the "Done" button has to be pressed before another tab is clicked, because tabs are implemented as HTML links (see *Buttons vs. Links*).

New customers or suppliers will automatically be assigned the next free number. You can enter another number, but an error will result if you try to use an existing number. The "Name" field is mandatory, you need to overwrite the "¡Name¿" clue.

Phone and fax numbers in the "Contact" tab must be entered in the correct format, depending on the locale (see *Telephone Numbers*).

The "Memo" tab contains a single text area. It is no problem to use it for large pieces of text, as it gets stored in a database blob internally.

Item

Source Code

```
# 09aug10abu
# (c) Software Lab. Alexander Burger

(must "Item" Item)

(menu ,"Item"
   (ifn *ID
      (prog
         (<h3> NIL ,"Select" " " ,"Item")
         (form 'dialog (choItem)) )
      (<h3> NIL ,"Item")
      (form NIL
         (<h2> NIL (<id> (: nr) " -- " (: nm)))
         (panel T (pack ,"Item" " @1") '(may Delete) '(choItem) 'nr '+Item)
         (<grid> 4
            ,"Number" NIL (gui '(+E/R +NumField) '(nr : home obj) 10) NIL
            ,"Description" NIL (gui '(+E/R +Cue +TextField) '(nm : home obj) ,"Item" 30) NIL
            ,"Supplier" (gui '(+ChoButton) '(choCuSu (field 1)))
            (gui '(+E/R +Obj +TextField) '(sup : home obj) '(nm +CuSu) 30)
            (gui '(+View +TextField) '(field -1 'obj 'ort) 30)
            ,"Inventory" NIL (gui '(+E/R +NumField) '(inv : home obj) 12)
            (gui '(+View +NumField) '(cnt> (: home obj)) 12)
            ,"Price" NIL (gui '(+E/R +FixField) '(pr : home obj) 2 12) )
         (--)
         (<grid> 2
            ,"Memo" (gui '(+BlobField) '(txt : home obj) 60 8)
            ,"Picture"
            (prog
               (gui '(+Able +UpField) '(not (: home obj jpg)) 30)
               (gui '(+Drop +Button) '(field -1)
                  '(if (: home obj jpg) ,"Uninstall" ,"Install")
                  '(cond
                     ((: home obj jpg)
                        (ask ,"Uninstall Picture?"
                           (put!> (: home top 1 obj) 'jpg NIL) ) )
                     ((: drop) (blob! (: home obj) 'jpg @)) ) ) ) )
         (<spread> NIL (editButton T))
         (gui '(+Img)
            '(and (: home obj jpg) (allow (blob (: home obj) 'jpg)))
            ,"Picture") ) ) )

# vi:et:ts=3:sw=3
```

Discussion

Items also have a unique number, and a mandatory "Description" field.

To assign a supplier, click on the "+" button. The Customer/Supplier search dialog will appear, and you can pick the desired supplier with the "@" button in the first column. Alternatively, if you are sure to know the exact spelling of the supplier's name, you can also enter it directly into the text field.

In the search dialog you may also click on a link, for example to inspect a possible supplier, and then return to the search dialog with the browser's back button. The "Edit" mode will then be lost, however, as another object has been visited (this is described in the last part of *Editing*).

You can enter an inventory count, the number of items currently in stock. The following field will automatically reflect the remaining pieces after some of these items were sold (i.e. referenced in order positions). It cannot be changed manually.

The price should be entered with the decimal separator according to the current locale. It will be formatted with two places after the decimal separator.

The "Memo" is for an arbitrary info text, like in *Customer/Supplier* above, stored in a database blob.

Finally, a JPEG picture can be stored in a blob for this item. Choose a file with the browser's file select control, and click on the "Install" button. The picture will appear at the bottom of the page, and the "Install" button changes to "Uninstall", allowing the picture's removal.

Order

Source Code

```
# 03sep09abu
# (c) Software Lab. Alexander Burger

(must "Order" Order)

(menu ,"Order"
   (ifn *ID
      (prog
         (<h3> NIL ,"Select" " " ,"Order")
         (form 'dialog (choOrd)) )
      (<h3> NIL ,"Order")
      (form NIL
         (<h2> NIL (<id> (: nr)))
         (panel T (pack ,"Order" " @1") '(may Delete) '(choOrd) 'nr '+Ord)
         (<grid> 4
            ,"Date" NIL
            (gui '(+E/R +DateField) '(dat : home obj) 10)
            (gui '(+View +TextField)
               '(text ,"(@1 Positions)" (length (: home obj pos))) )
            ,"Customer" (gui '(+ChoButton) '(choCuSu (field 1)))
            (gui '(+E/R +Obj +TextField) '(cus : home obj) '(nm +CuSu) 30)
            (gui '(+View +TextField) '(field -1 'obj 'ort) 30) )
         (--)
         (gui '(+Set +E/R +Chart) '((L) (filter bool L)) '(pos : home obj) 8
            '((Pos I)
               (with Pos
                  (list I NIL (: itm) (or (: pr) (: itm pr)) (: cnt) (sum> Pos)) ) )
            '((L D)
               (cond
                  (D
                     (put!> D 'itm (caddr L))
                     (put!> D 'pr (cadddr L))
                     (put!> D 'cnt (; L 5))
                     (and (; D itm) D) )
                  ((caddr L)
                     (new! '(+Pos) 'itm (caddr L)) ) ) ) ) )
```

```
         (<table> NIL NIL
            '((align) (btn) (NIL ,"Item") (NIL ,"Price") (NIL ,"Quantity") (NIL ,"Total"))
            (do 8
               (<row> NIL
                  (gui 1 '(+NumField))
                  (gui 2 '(+ChoButton) '(choItem (field 1)))
                  (gui 3 '(+Obj +TextField) '(nm +Item) 30)
                  (gui 4 '(+FixField) 2 12)
                  (gui 5 '(+NumField) 8)
                  (gui 6 '(+Sgn +Lock +FixField) 2 12)
                  (gui 7 '(+DelRowButton))
                  (gui 8 '(+BubbleButton)) ) )
            (<row> NIL NIL NIL (scroll 8 T) NIL NIL
               (gui '(+Sgn +View +FixField) '(sum> (: home obj)) 2 12) ) )
         (<spread>
            (gui '(+Rid +Button) ,"PDF-Print"
               '(if (check> (: home obj))
                  (note ,"Can't print order" (uniq @))
                  (psOut 0 ,"Order" (ps> (: home obj))) ) )
            (editButton T) ) ) ) )

# vi:et:ts=3:sw=3
```

Discussion

Oders are identified by number and date.

The number must be unique. It is assigned when the order is created, and cannot be changed for compliance reasons.

The date is initialized to "today" for a newly created order, but may be changed manually. The date format depends on the locale. It is YYYY-MM-DD (ISO) by default, DD.MM.YYYY in the German and YYYY/MM/DD in the Japanese locale. As described in *Time & Date*, this field allows input shortcuts, e.g. just enter the day to get the full date in the current month.

To assign a customer to this order, click on the "+" button. The Customer/Supplier search dialog will appear, and you can pick the desired customer with the "@" button in the first column (or enter the name directly into the text field), just as described above for *Item*s.

Now enter order the positions: Choose an item with the "+" button. The "Price" field will be preset with the item's default price, you may change it manually. Then enter a quantity, and click a button (typically the "+" button

to select the next item, or a scroll button go down in the chart). The form will be automatically recalculated to show the total prices for this position and the whole order.

Instead of the "+" or scroll buttons, as recommended above, you could of course also press the "Done" button to commit changes. This is all right, but has the disadvantage that the button must be pressed a second time (now "Edit") if you want to continue with the entry of more positions.

The "x" button at the right of each position deletes that position without further confirmation. It has to be used with care!

The "$\hat{?}$" button is a "bubble" button. It exchanges a row with the row above it. Therefore, it can be used to rearrange all items in a chart, by "bubbling" them to their desired positions.

The "PDF-Print" button generates and displays a PDF document for this order. The browser should be configured to display downloaded PDF documents in an appropriate viewer. The source for the postscript generating method is in "app/lib.l". It produces one or several A4 sized pages, depending on the number of positions.

Reports

Source Code

```
# 08mar10abu
# (c) Software Lab. Alexander Burger

(must "Inventory" Report)

(menu ,"Inventory"
   (<h3> NIL ,"Inventory")
   (form NIL
      (<grid> "-.-"
         ,"Number" NIL
         (prog
            (gui '(+Var +NumField) '*InvFrom 10)
            (prin " - ")
            (gui '(+Var +NumField) '*InvTill 10) )
         ,"Description" NIL (gui '(+Var +TextField) '*InvNm 30)
         ,"Supplier" (gui '(+ChoButton) '(choCuSu (field 1)))
         (gui '(+Var +Obj +TextField) '*InvSup '(nm +CuSu) 30) )
      (--)
      (gui '(+ShowButton) NIL
         '(csv ,"Inventory"
            (<table> 'chart NIL
               (<!>
                  (quote
                     (align)
                     (NIL ,"Description")
                     (align ,"Inventory")
                     (NIL ,"Supplier")
                     NIL
                     (NIL ,"Zip")
                     (NIL ,"City")
                     (align ,"Price") ) )
               (catch NIL
                  (pilog
                     (quote
                        @Rng (cons *InvFrom (or *InvTill T))
                        @Nm *InvNm
                        @Sup *InvSup
                        (select (@Item)
                           ((nr +Item @Rng) (nm +Item @Nm) (sup +Item @Sup))
                           (range @Rng @Item nr)
                           (tolr @Nm @Item nm)
                           (same @Sup @Item sup) ) )
```

```
                     (with @Item
                        (<row> (alternating)
                           (<+> (: nr) This)
                           (<+> (: nm) This)
                           (<+> (cnt> This))
                           (<+> (: sup nm) (: sup))
                           (<+> (: sup nm2))
                           (<+> (: sup plz))
                           (<+> (: sup ort))
                           (<-> (money (: pr))) ) )
                     (at (0 . 10000) (or (flush) (throw))) ) ) ) ) ) ) )
# vi:et:ts=3:sw=3
```

```
# 08mar10abu
# (c) Software Lab. Alexander Burger

(must "Sales" Report)

(menu ,"Sales"
   (<h3> NIL ,"Sales")
   (form NIL
      (<grid> "-.-"
         ,"Date" NIL
         (prog
            (gui '(+Var +DateField) '*SalFrom 10)
            (prin " - ")
            (gui '(+Var +DateField) '*SalTill 10) )
         ,"Customer" (gui '(+ChoButton) '(choCuSu (field 1)))
         (gui '(+Var +Obj +TextField) '*SalCus '(nm +CuSu) 30) )
      (--)
      (gui '(+ShowButton) NIL
         '(csv ,"Sales"
            (<table> 'chart NIL
               (<!>
                  (quote
                     (align)
                     (NIL ,"Date")
                     (NIL ,"Customer")
                     NIL
                     (NIL ,"Zip")
                     (NIL ,"City")
                     (align ,"Total") ) )
```

```
               (catch NIL
                  (let Sum 0
                     (pilog
                        (quote
                           @Rng (cons *SalFrom (or *SalTill T))
                           @Cus *SalCus
                           (select (@Ord)
                              ((dat +Ord @Rng) (cus +Ord @Cus))
                              (range @Rng @Ord dat)
                              (same @Cus @Ord cus) ) )
                        (with @Ord
                           (let N (sum> This)
                              (<row> (alternating)
                                 (<+> (: nr) This)
                                 (<+> (datStr (: dat)) This)
                                 (<+> (: cus nm) (: cus))
                                 (<+> (: cus nm2))
                                 (<+> (: cus plz))
                                 (<+> (: cus ort))
                                 (<-> (money N)) )
                              (inc 'Sum N) ) )
                        (at (0 . 10000) (or (flush) (throw))) )
                     (<row> 'nil
                        (<strong> ,"Total") - - - - -
                        (<strong> (prin (money Sum))) ) ) ) ) ) ) ) )

# vi:et:ts=3:sw=3
```

Discussion

The two reports ("Inventory" and "Sales") come up with a few search fields and a "Show" button.

If no search criteria are entered, the "Show" button will produce a listing of the relevant part of the whole database. This may take a long time and cause a heavy load on the browser if the database is large.

So in the normal case, you will limit the domain by stating a range of item numbers, a description pattern, and/or a supplier for the inventory report, or a range of order dates and/or a customer for the sales report. If a value in a range specification is omitted, the range is considered open in that direction.

At the end of each report appears a "CSV" link. It downloads a file with the TAB-separated values generated by this report.

Part IV

PicoLisp Community Articles

22

VizReader's distributed word index

Henrik Sarvell

hsarvell@gmail.com

Summary. This article is about managing a simple distributed index with the *id* function.

22.1 Introduction

After having used VizReader[1] as a single local application for a while it quickly became clear that the full word index that are mapping words to articles - in order to enable word searches - was growing at an alarming rate. Something needed to be done and I started looking into the ext[2] functionality. It is overkill for managing a simple distributed index though, the id[3] function however is a good fit.

22.2 Setup

Currently the remotes all reside on the same machine so the speedup is achieved by querying a bunch of smaller files in parallel as opposed to having only one process go through one big file. Had this been done "properly" by using different machines the speedup would probably be much bigger, especially if said machines were located in the same data center.

[1] http://vizreader.com
[2] http://www.software-lab.de/doc/refE.html#ext
[3] http://www.software-lab.de/doc/refI.html#id

22.3 Implementation

Before we start looking at the actual code let's first list what is happening from start to finish:

1. When an article is imported all words are counted and the count, the resultant numbers from calling id on the article and the word object, and the article's date are all sent to and saved in a remote. Which remote to send to is inferred from the aid number of the article which is not to be confused with the result returned by the id function, aid is created explicitly as an auto incremented key.
2. When a search is performed all remotes are queried at the same time. Since we are storing the article's date in the index too it is possible for each remote to return results sorted by date. The logic that is responsible for querying the remotes will then store the first result from each remote in a hash that is continuously sorted by date in order to link only the newest articles from the parallel query, more on that later.
3. To build the result sent for display in the GUI is now just a simple matter of calling id on our list of numbers in order to fetch the real articles on disc.

```
(class +Aword +Entity)
(rel word (+Need +Key +String))

(class +Article +Entity)
(rel aid       (+Key +Number))
(rel title     (+String))
(rel htmlUrl   (+Key +String))
(rel body      (+Blob))
(rel picoStamp (+Ref +Number))

. . .
```

```
(dm setArticleWords> (A Ws)
   (let Idx (idx> (; A words))
      (put> A 'common
         (eval> '+Agent A 'setArticleWords
            (lit
               (make
                  (for Wstr Ws
                     (unless (common?> This Wstr)
                        (link
                           (list
                              (id A)
                              (id (request '(+Aword) 'word Wstr))
                              (val (car (idx 'Idx (lowc Wstr))))
                              (; A picoStamp)))))))))))
```

The +Agent class that is responsible for communicating with the remotes will call 'setArticleWords on the remote that is inferred from the article A in question, that's why it's the first argument to the eval> function. Ws is a list of all words in the article, Idx is an index tree[4] mapping words to their counts, this tree has been generated earlier and is now simply rebuilt. We loop through each word and filter out common words. Each element in the list we then send to the remote is generated by calling id on the article and the word, getting the count from the index tree and the date from the article.

I will not explain the eval> method of my +Agent class since it contains a lot of logic that doesn't have anything to do with what this article is about, the main thing happening there is simply using pr[5] to send (setArticleWords Lst) to the remote which simply uses eval[6] to execute.

```
(class +WordCount +Entity)
(rel article   (+Ref +Number))
(rel word      (+Aux +Ref +Number) (article))
(rel count     (+Number))
(rel picoStamp (+Ref +Number))
```

[4] http://www.software-lab.de/doc/refI.html#idx
[5] http://www.software-lab.de/doc/refP.html#pr
[6] http://www.software-lab.de/doc/refE.html#eval

```
(de setArticleWords (Lst)
   (dbSync)
   (let Res
      (mapcar id
         (tail 20
            (by '((Wc)(; Wc count)) sort
               (make
                  (for Wc Lst
                     (link
                        (request
                           '(+WordCount)
                           'article (car Wc)
                           'word (cadr Wc)
                           'count (caddr Wc)
                           'picoStamp (last Wc))))))))
      (commit 'upd)
      (pr Res)))
```

The above is the remote ER and the function we just called, note that in addition to storing the words here we are also returning the 20 most common words for local storage, yet again using id but here on the remote instead.

```
(dm getAsByWd> (W Start)
   (mapcar '((A)(id (db: +Article) A))
      (extArticles> '+Agent Start 25 'getArticles (lit (id W)))))

. . .

(dm rd1> (Sock)
   (or
      (in Sock (rd))
      (nil
         (close Sock))))

(dm extArticles> (Start Count . @)
   (let (End (+ Start Count) Socks (getSocks> This))
      (for S Socks
         (out S (pr (peel> '+Gh (rest))))))
```

```
(let Q (new '(+Hash) (extract '((S)(let A (rd1> This S) (when A (list S A)))) Socks))
   (tail Count
      (make
         (until (empty?> Q)
            (let Cur (car (sort> Q T 'cdadr))
               (link (caadr Cur))
               (let A (rd1> This (car Cur))
                  (if A
                     (set> Q (car Cur) A)
                     (del> Q (car Cur))))
               (when (<= End (length (made)))
                  (empty> Q)))))))))
```

Here we query the remotes for all articles containing a certain word, the first result set will use a Start value of 1, the Count will always be 25. We use id to get the word's number for the remote and then id again to fetch the local articles for the result we send to the GUI. In extArticles> we first figure out when to stop by adding Start and Count, in our case it will be 26. We then get all the sockets to the remotes and send the code for execution by way of pr. Next we build a hash using our sockets as keys with the article info in the value. Then we continue with reading from the sockets until the hash is empty which can happen if all the remotes have finished sending articles or if we have reached our goal, in this case fetching 25 articles. The hash is repeatedly sorted by date and the newest article is linked, the spot where that article came from will then be filled by the next article from the remote in question and so on until we meet one of the end conditions.

```
(de go ()
. . .
   (rollback)
   (task (port (+ *IdxNum 4040))
      (let? Sock (accept @)
         (unless (fork)
            (in Sock
               (while (rd)
                  (sync)
                  (out Sock
                     (eval @))))
            (bye))
         (close Sock)))
   (forked))
```

```
(de getArticles (W)
   (let Goal
      (goal
         (quote
            @Word W
            @Date (cons (- (stamp> '+Gh) (* 6 31 86400)) (stamp> '+Gh))
            (select (@Wcs)
               ((picoStamp +WordCount @Date) (word +WordCount @Word))
               (same @Word @Wcs word)
               (range @Date @Wcs picoStamp))))
      (do 25
         (NIL (prove Goal))
         (bind @
            (pr (cons (; @Wcs article) (; @Wcs picoStamp)))
            (unless (flush) (bye)))))
   (bye))
```

The above is all on the remote, note that the go method is built to be able to receive repeated use of pr from the local/master of which there is no example yet in VizReader, anyway because of that fact we need to finish getArticles with a bye in order to stop execution. The getArticles function will repeatedly pr all articles that are newer than half a year and which at the same time contain the word in question, we will stop after having printed 25 of them. It won't make sense printing more since the receiving end only wants a maximum of 25 anyway.

23

Asynchronous Programming in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

23.1 Introduction

If you've been using PicoLisp for some time doing nitty gritty stuff like interfacing with such horrible things as PHP and other abominations you can probably infer from the headline that the pitfall here is quite trivial. However, it's the journey that is the important part, not the somewhat anticlimactic end to this whole adventure. Having said that, let's get to it.

23.2 Asynchronous Evaluation in PicoLisp

In PicoLisp there is a clever way of asynchronous evaluation, it can be useful if you:

1. Have a very computationally heavy problem, you fork the execution to utilize multiple cores and all that cheap RAM.
2. Need to query X services and don't care to wait for each call to finish before making the next one.

This "strategy" involves

later[1] and wait[2], see documents for later[3] in particular.

[1] http://software-lab.de/doc/refL.html#later
[2] http://software-lab.de/doc/refW.html#wait
[3] http://software-lab.de/doc/refL.html#later

My interest is web development so #2 is standard for me and up until now I've used sockets[4] in combination with pr[5] and rd[6]. The "standard" PicoLisp way to do these things if you will.

23.3 HTTP only

However, at the moment I'm working on something new and I want to work solely with HTTP, not with PLIO, despite it being faster/more efficient than HTTP. Let's just say that HTTP is the only allowed way of communication in my "spec".

23.3.1 Using `call`

So I went happily on my way coding up all this communication with the help of `CURL` and the call[7] function.

Let's have a code listing:

```
(de callAll (Func Data)
   (let Result
      (make
         (for S (collect 'id '+Server)
            (later (chain (cons "void"))
               (list (; S id) (callOne Func Data S)))))
      (wait 30000 (not (memq "void" Result)))
      Result))
```

Here you can see the core of the strategy, we loop through a list of nodes (`+Server`) to query each and every for some arbitrary data. What you don't see in the above listing is the contents of `callOne` but it doesn't matter, we'll get to that soon enough. In essence what we're doing here is getting all the nodes in my massive distributed database (there are at the time of this writing two of them running on my laptop but whatever).

These nodes might be more or less busy doing other stuff so response times might vary and can you imagine if the first one takes 10 seconds to reply and the second one takes 5 seconds? In a non-asynch world that would add up to a grand total of 15 seconds for the query.

[4] http://software-lab.de/doc/refC.html#connect
[5] http://software-lab.de/doc/refP.html#pr
[6] http://software-lab.de/doc/refR.html#rd
[7] http://software-lab.de/doc/refC.html#call

Luckily we use later and wait to avoid that and query them in parallel for a grand total of 10 seconds. What's happening here is that we add the results to a list, each entry in the list will start with "void", then we fork[8]. Each fork will return a result (or not but we'll get to that very very soon).

The second (list ...) argument there is supposed to return a list with the `id` of the node in the car and the result of the query in the cdr. If a node times out one or more positions in the list will still be "void" instead of something useful.

Finally the wait call will wait for 30 seconds or until all nodes have returned something (i.e not timed out). After that wait we return the list which I expected would look something like this for my specific case (counting the objects in each node): ((1 "1") (2 "0")).

The problem was, the result didn't look like that, it looked more like `(NIL NIL)`. I was using call like this:

```
(call 'curl '-m 30 '-d
   ``key1=value1\&key2=value2`` ''http://localhost:8081/@exec`` )
```

The reason for that is ye old copy paste problem, not even that in this case actually. Just checking code from an old project[9] and seeing call in action without realizing that in that case it was actually used correctly (I wasn't interested in the returned content).

23.3.2 Using `in`

Yep, `call` returns `T`, not the actual result which makes it useless in this case. I now use `in`, like this:

```
(in (list ``curl'' ``-m`` Timeout ''-d`` ''key1=value1\&key2=value2``
      ''http://localhost:8081/@exec`` ) (till NIL T) )
```

and all is well.

So now you don't have make the same mistake and spend two hours of your life before you actually RTFM. However, by now you should've realized that asynchronous logic can be indispensible when you don't want to wait for too long for something good.

---

[8] http://software-lab.de/doc/refP.html#pipe

[9] http://vizreader.com

24

PicoLisp Ticker

Alexander Burger

abu@software-lab.de

Summary. This article describes how a *PicoLisp Ticker* page is set up to produce an endless stream of pseudo-text, and how the *Googlebot* reacts to such "nonsensical" data.

24.1 Producing an endless stream of pseudo-text

Around end of May, I was playing with an algorithm I had received from *Bengt Grahn*, many years ago. A small program - it was even part of the PicoLisp distribution ("misc/crap.l") for many years - which when given an arbitrary sample text in some language, produces an endless stream of pseudo-text which strongly resembles that language.

It was fun, so I decided to set up a PicoLisp "Ticker" page, producing a stream of "news": http://ticker.picolisp.com[1]

24.2 Implementing a ticker page

The source for the server is simple:

```
(allowed ()
   *Page "!start" "@lib.css" "ticker.zip" )

(load "@lib/http.l" "@lib/xhtml.l")
(load "misc/crap.l")

(one *Page)
```

[1] http://ticker.picolisp.com

```
(de start ()
   (seed (time))
   (html 0 "PicoLisp Ticker" "@lib.css" NIL
      (<h2> NIL "Page " *Page)
      (<div> 'em50
         (do 3 (<p> NIL (crap 4)))
         (<spread>
            (<href> "Sources" "ticker.zip")
            (<this> '*Page (inc *Page) "Next page") ) ) ) )

(de main ()
   (learn "misc/ticker.txt") )

(de go ()
   (server 21000 "!start") )
```

The sample text for the learning phase, "misc/ticker.txt", is a plain text version of the PicoLisp FAQ[2]. The complete source, including the text generator, can be downloaded via the "Sources" link as "ticker.zip".

Now look at the "Next page" link, appearing on the bottom right of the page. It always points to a page with a number one greater than the current page, providing an unlimited supply of ticker pages.

I went ahead, and installed and started the server. To get some logging, I inserted the line

```
(out 2 (prinl (stamp) " {" *Url "} Page " *Page "  [" *Adr "] " *Agent))
```

at the beginning of the `start` function.

24.3 Googlebot in action

On June 18th I announced it on Twitter, and watched the log files. Immediately, within one or two seconds (!), Googlebot accessed it:

```
2011-06-18 11:22:04  Page 1  [66.249.71.139] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

Wow, I thought, that was fast! Don't know if this was just by chance, or if Google always has such a close watch on Twitter.

[2] http://software-lab.de/doc/faq.html

Anyway, I was curious about what the search engine would do with such nonsense text, and how it would handle the infinite number of pages. During the next seconds and minutes, other bots and possibly human users accessed the ticker:

```
2011-06-18 11:22:08 Page 1 [65.52.23.76] Mozilla/4.0 (compatible;
MSIE 7.0; Windows NT 6.0)

2011-06-18 11:22:10 Page 1 [65.52.4.133] Mozilla/4.0 (compatible;
MSIE 7.0; Windows NT 6.0)

2011-06-18 11:22:20 Page 1 [50.16.239.111] Mozilla/5.0 (compatible;
Birubot/1.0) Gecko/2009032608 Firefox/3.0.8

2011-06-18 11:29:52 Page 1 [174.129.42.87] Python-urllib/2.6

2011-06-18 11:30:34 Page 1 [174.129.42.87] Python-urllib/2.6

2011-06-18 11:33:54 Page 1 [89.151.99.92] Mozilla/5.0 (compatible;
MSIE 6.0b; Windows NT 5.0) Gecko/2009011913 Firefox/3.0.6
TweetmemeBot

2011-06-18 11:33:5n4 Page 1 [89.151.99.92] Mozilla/5.0 (compatible;
MSIE 6.0b; Windows NT 5.0) Gecko/2009011913 Firefox/3.0.6
TweetmemeBot

2011-06-18 13:47:21 Page 1 [190.175.174.220] Mozilla/5.0 (X11; U;
Linux i686; en-US; rv:1.9.2.17) Gecko/20110428 Fedora/3.6.17-1.fc14
Firefox/3.6.17

2011-06-18 13:49:13 Page 2 [190.175.174.220] Mozilla/5.0 (X11; U;
Linux i686; en-US; rv:1.9.2.17) Gecko/20110428 Fedora/3.6.17-1.fc14
Firefox/3.6.17

2011-06-18 13:49:21 Page 3 [190.175.174.220] Mozilla/5.0 (X11; U;
Linux i686; en-US; rv:1.9.2.17) Gecko/20110428 Fedora/3.6.17-1.fc14
Firefox/3.6.17

2011-06-18 19:43:36 Page 1 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.n742.91
Safari/534.30
```

```
2011-06-18 19:43:54 Page 2 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30

2011-06-18 19:44:11 Page 3 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30

2011-06-18 19:44:13 Page 4 [24.167.162.218] Mozillan/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30

2011-06-18 19:44:16 Page 5 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30

2011-06-18 19:44:18 Page 6 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30

2011-06-18 19:44:20 Page 7 [24.167.162.218] Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91
Safari/534.30
```

Mr. Google came back the following day:

```
2011-06-19 00:25:57 Page 2 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)

2011-06-19 01:03:13 Page 3 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)

2011-06-19 01:35:57 Page 4 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)

2011-06-19 02:39:19 Page 5 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)

2011-06-19 03:n43:39 Page 6 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)

2011-06-19 04:17:02 Page 7 [66.249.67.197] Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html)
```

In between (not shown here) were also some accesses, probably by non-bots, who usually gave up after a few pages.

Mr. Google, however, assiduously went through "all" pages. The page numbers increased sequentially, but he also re-visited page 1, going up again. Now there were several indexing threads, and by June 23rd the first one exceeded page 150.

I felt sorry for poor Googlebot, and installed a "robots.txt" the same day, disallowing the ticker page for robots. I could see that several other bots fetched "robots.txt". But not Google. Instead, it kept following the pages of the ticker.

Then, finally, on July 5th, Googlebot looked at "robots.txt":

```
"robots.txt" 2011-07-05 07:03:05 Mozilla/5.0 (compatible;
Googlebot/2.1; +http://www.google.com/bot.html) ticker.picolisp.com
"robots.txt: disallowed all"
```

The indexing, however, went on. Excerpt:

```
2011-07-05 04:27:46 {!start} Page 500 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 04:58:50 {!start} Page 501 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 05:30:24 {!start} Page 502 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 06:02:10 {!start} Page 503 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 06:32:14 {!start} Page 504 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 07:02:41 {!start} Page 505 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 08:02:31 {!start} Page 506 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 08:45:52 {!start} Page 507 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

```
2011-07-05 09:20:06 {!start} Page 508 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-05 09:51:49 {!start} Page 509 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

Strange. I would have expected the indexing to stop after Page 505.

In fact, all other robots seem to obey "robots.txt". Mr. Google, however, even started a new thread five days later again:

```
2011-07-10 02:22:52 {!start} Page 1 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

I should feel flattered, if the PicoLisp news ticker is so interesting!

How will that go on? As of today, we have reached

```
2011-07-15 09:42:36 {!start} Page 879 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

I'll stay tuned ...

Now ... 8 hours later (17:10):

As I wrote in the mailing list, I've extended "robots.txt/default" to exclude also "/21000/" from "picolisp.com". This was 14:58, two hours ago. Meanwhile,

```
2011-07-15 15:29:59 {!start} Page 800 [74.125.94.84] Mozilla/5.0
(X11; Linux x86_64) AppleWebKit/534.24 (KHTML, like Gecko; Google
Web Preview) Chrome/11.0.696 Safari/534.24

2011-07-15 15:43:58 {!start} Page 889 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-15 16:13:54 {!start} Page 890 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)

2011-07-15 16:27:24 {!start} Page 1 [66.249.71.203] Mozilla/5.0
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

we still have accesses, and even a restart from page one (which I wouldn't have expected now).

As I wrote, "robots.txt/default" now looks like this:

```
(prinl "User-Agent: *")

(prinl "Disallow:"
   (cond
      ((= *Host '`(chop "ticker.picolisp.com")) " /")
      ((= *Host '`(chop "picolisp.com")) " /21000/") ) )
```

Looking at the returned contents:

```
: (client "ticker.picolisp.com" 80 "robots.txt" (out NIL (echo)))
User-Agent: *
Disallow: /
```

and

```
: (client "picolisp.com" 80 "robots.txt" (out NIL (echo)))
User-Agent: *
Disallow: /21000/
```

... next morning

Good! This helped. Googlebot seems to have stopped all traversals.

The log entry at 16:27 yesterday is the last one.

In summary, I can't blame Google. It was actually my fault not to explicitly disallow /21000/, because for a bot the links looks like pointing to a different site. Just disabling the "root" of the traversal is not enough; there is no garbage collector involved.

25

The many uses of @ in PicoLisp

Thorsten Jolitz

tjolitz@gmail.com

Summary. This article gives an overview over the many uses of `@` in PicoLisp.

25.1 The `@` mark in PicoLisp

The *AT-mark* `@` is everywhere in PicoLisp source code, and sometimes it is not obvious, at least for beginners, what the meaning of `@` in the context at hand is.

Here is a table that summarizes all uses of `@` in PicoLisp, giving examples and explanations, as well as links to related docs with more information. It is probably necessary to read the docs first to understand the compact information in the table. This summary serves only as a quick overview, helping to find out the context and meaning of an otherwise mysterious `@` mark in some PicoLisp code.

All (?) possible uses of @ with examples and explanations

context	use	meaning	reference
CAR of a lambda expression	(de foo @ ...)	all arguments are evaluated and kept internally in a list	http://software-lab.de/doc/ref.html
read-eval-loops	(- @ @@ @@@)	the result of the last (3) evaluation (s) stored in the VAL of symbol	http://software-lab.de/doc/ref.html
flow- and logic functions with conditional expressions	(while (read) (println @)), (and and (@ (min @ 5) (prinl @) (gt0 (dec @)) .))	store result of (the last) conditional expression	http://software-lab.de/doc/ref.html
flow- and logic functions with controlling expressions	(case @ ("^M" NIL) ("^J" "^M") (T @))	store result of controlling expression	http://software-lab.de/doc/ref.html
'match' and 'fill'	(match '(@A Zeit) '(Keine))	Pattern Wildcard	http://software-lab.de/doc/ref.html
'text'	(text "abc @1 def @2" 'XYZ 123)	replacing all occurrences of an at-mark "@", followed by one of the letters "1" through "9", and "A" through "Z", with the corresponding any argument.	http://software-lab.de/doc/refT.html#text
path names	(load "@lib/misc.l")	home directory substitution	http://software-lab.de/doc/tut.html
Pilog	(be likes (John @X))	Pilog variable	http://software-lab.de/doc/ref.html
Pilog	(be likes (John @))	Anonymous Pilog variable	http://software-lab.de/doc/ref.html
shared object libraries	(native "@" "getenv" 'S "TERM") # Same as (sys "TERM")	(64-bit version only) Calls a native C function. The first argument should specify a shared object library, e.g. "@" (here @ as transient symbol stands for the current main program).	http://software-lab.de/doc/refN.html#native

26

Wacky Stuff with circular Lists

José Ignacio Romero

jir@2.71828.com.ar

Summary. This article demonstrates and explains *cirular lists* in PicoLisp.

26.1 Example 1 with walk-through

```
(and and (@ (min @ 5) (prinl @) (gt0 (dec @)) .))
```

I'm `eval`. I see a list, look inside, I see a symbol called `'and`. I look at it's `val`. It's a number, thus a function pointer, I call it with the rest of the list unevaluated.

I'm `doAnd`, I look at the list I was passed, I see a symbol called `'and`, I evaluate it, a number came out, it's not `NIL`, so I shove it in `@` and look at the next `cadr`. It's a list, I evaluate it, has a symbol called `@` in it's `car`, that symbol resolves to a number, a pointer to `doAnd`, i call it with the rest of the list unchanged.

I'm `doAnd`, I look at the list I was passed, the first argument, evaluating it results in a function call that returns `5`, it's not `NIL`, so I shove it to `@` and go on. The next element is another list, a call to `prinl` happens, it returned `5`, it's not `NIL`, so I shove it to `@` and go on. Look at the next element, a call to `(gt0 (dec @))`, returns `4`, that is not `NIL`, so I shove the `4` in `@` and go on. Looking at the next `cadr` I see `@` again (but I don't realize, because I don't know wether a list is circular or not), it evaluates to `4`, so I shove it to `@` keep going.

```
I see (min @ 5) again.........
 ... 3
 ... 2
 ... 1
 ... (gt0 (dec @)) returns NIL here
```

so I stop evaluating and return `NIL`

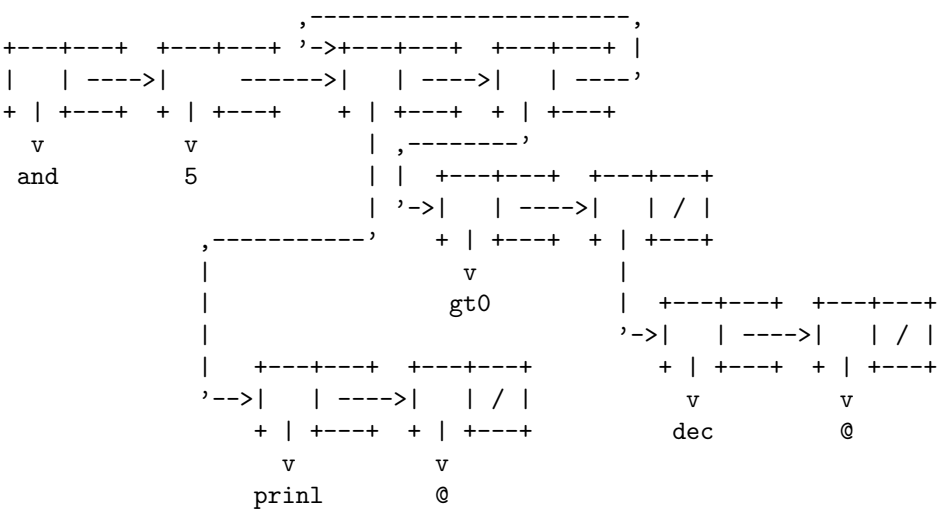
Back in the first `doAnd`, I see the previous call returned `NIL`, so I stop evaluating right there and return `NIL`.

26.2 Example 2 with graphical depiction

Example 1 could be written with only one `'and` using the right syntax (and losing part of it's *rube-goldberg* appeal):

```
(and 5 . ((prinl @) (gt0 (dec @)) .))
```

This is how that *S-expression* should look in memory, I omitted the technically correct representation of numbers and symbols to keep it simple:



27

Speedtest PicoLisp vs Elisp

Thorsten Jolitz[1] and José Romero[2]

[1] tjolitz@gmail.com
[2] jir@2.71828.com.ar

Summary. This article compares the speed of (compiled and interpreted) *Emacs Lisp* with (always interpreted) *PicoLisp*, testing the costs of *function calls/arithmetic* as well as *list manipulation*.

27.1 The Tests

27.1.1 Function Call/Arithmetic Cost

Shell Script Approach

The classic *Fibonacci function* was used for measuring function call/arithmetic cost.

Here is the PicoLisp script:

```
#!/usr/bin/picolisp
(de fibo (N)
   (if (> 2 N)
      1
      (+ (fibo (dec N)) (fibo (- N 2))) ) )
(fibo 35)
(bye)
```

Here is the (uncompiled) Emacs Lisp script:

```
#!/usr/bin/emacs --script

(defun fibo (N)
   (if (> 2 N)
      1
      (+ (fibo (1- N)) (fibo (- N 2))) ) )

(fibo 35)
```

Here is the script that calls a byte-compiled Emacs Lisp file with the above function definition and call:

```
#! /bin/sh
":"; exec emacs --no-site-file --script
"/home/tj/shellscripts/tj-fibo-compiled.elc" # -*-emacs-lisp-*-
```

The following shell command was used to measure the performance:

```
[tj@arch  ]$ time script
```

with *script* being one of the three scripts above.

Command Line Approach

This is an alternative, more elegant and efficient way to run the tests. Just produce these two files:

```
$ cat > fibo.el << .
 (defun fibo (N)
    (if (> 2 N)
       1
       (+ (fibo (1- N)) (fibo (- N 2))) ) )

 (fibo 35)
 .


$ cat > fibo.l << .
 (de fibo (N)
    (if (> 2 N)
       1
       (+ (fibo (dec N)) (fibo (- N 2))) ) )
 (fibo 35)
 .
```

Then byte-compile fibo.el and run the following commands:

```
$ time emacs --no-site-file --script fibo.el
$ time emacs --no-site-file --script fibo.elc
$ time pil fibo.l -bye
```

As a side note: Emacs can be invoked noninteractively from the shell to do byte compilation with the aid of the function batch-byte-compile. In this case, the files to be compiled are specified with command-line arguments. Use a shell command of the form

```
emacs -batch -f batch-byte-compile files...
```

for example

```
$ emacs --no-site-file -batch -f batch-byte-compile fibo.el
```

27.1.2 List Manipulation Cost

The costs of list manipulation were tested with the "extensive list manipulations" code from Alexander Burger:

```
$ cat > tst.l << .
 (de tst ()
   (mapcar
    (quote (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)) ) )
  (do 1000000 (tst))
 .
```

```
$ cat > tst.el << .
 (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)) ) )
 (dotimes (i 1000000) (tst))
 .
```

27.2 Results

27.2.1 32bit

System Information

```
 $ uname -a
 Linux icz 2.6.32-5-686 #1 SMP Mon Jan 16 16:04:25 UTC 2012 i686
 GNU/Linux
 $ cat /proc/cpuinfo |grep "model name" | cut -d: -f2
  Pentium(R) Dual-Core CPU       T4200  @ 2.00GHz
  Pentium(R) Dual-Core CPU       T4200  @ 2.00GHz
```

Function Calls

These are the results for running `fibo (N)` with `N=35`:

```
           | PicoLisp         | 0m5.662s  | 1x      |
           | Elisp            | 0m13.854s | ca 2.5x |
           | Elisp (compiled) | 0m5.882s  | ca 1x   |
```

PicoLisp is 2.5x faster than interpreted Emacs Lisp and as fast as compiled Emacs Lisp.

List Manipulation

These are the results for running `tst` with `(do 1000000 (tst)` or `(dotimes (i 1000000) (tst))`:

```
| PicoLisp         | 0m1.208s | 1x      |
| Elisp            | 0m8.311s | ca 7x   |
| Elisp (compiled) | 0m5.622s | ca 4.5x |
```

PicoLisp is 7x faster than interpreted Emacs Lisp and 4.5x faster than compiled Emacs Lisp. Looks like the Emacs compiler can't improve much in that function and it's still 4.6-6.9x slower than PicoLisp.

27.2.2 64bit

System Information

```
$ uname -a
Linux arch 3.3.2-1-ARCH #1 SMP PREEMPT Sat Apr 14 09:48:37 CEST 2012
x86_64 AMD Athlon(tm) 64 X2 Dual Core Processor 5000+ AuthenticAMD
GNU/Linux
$ cat /proc/cpuinfo |grep "model name" | cut -d: -f2
 AMD Athlon(tm) 64 X2 Dual Core Processor 5000+
 AMD Athlon(tm) 64 X2 Dual Core Processor 5000+
```

Function Calls

These are the results for running `fibo (N)` with `N=35`:

```
| PicoLisp         | 0m3.191s  | 1x    |
| Elisp            | 0m12.731s | ca 4x |
| Elisp (compiled) | 0m6.635s  | ca 2x |
```

PicoLisp is 4x faster than interpreted Emacs Lisp and 2x faster than compiled Emacs Lisp.

These are the results for running `fibo (N)` with `N=40`:

```
| PicoLisp         | 0m35.982s | 1x    |
| Elisp            | 2m14.352s | ca 4x |
| Elisp (compiled) | 1m13.304s | ca 2x |
```

Again PicoLisp is ca. 2x faster than compiled Elisp and 4x faster than interpreted Elisp.

List Manipulation

These are the results for running `tst` with `(do 1000000 (tst))` or `(dotimes (i 1000000) (tst))`:

```
| PicoLisp         | 0m1.635s | 1x      |
| Elisp            | 0m9.582s | ca 6x   |
| Elisp (compiled) | 0m7.129s | ca 4.5x |
```

PicoLisp is 6x faster than interpreted Emacs Lisp and 4.5x faster than compiled Emacs Lisp.

Just to remind you - PicoLisp is always interpreted, but the interpreter is designed with the need for speed[3].

27.2.3 32bit vs 64bit

All other things equal, 64-bit PicoLisp is usually slower than the 32-bit version, due to a poorer memory cache performance (the cells are twice as large size). On the other hand, arithmetics are faster, due to the additional short number type in pil64.

[3] http://picolisp.com/5000/!wiki?needforspeed

Part V

PicoLisp Community Tutorials

28

PicoLisp at first glance

Henrik Sarvell

hsarvell@gmail.com

Summary. This is the first in a series of articles for absolute PicoLisp beginners. The series will contrast PicoLisp against PHP in the examples to make it easier for C-people to understand.

28.1 PicoLisp at first glance

As I announced earlier, the plan was to create some small proof of concept web-thing in c-lisp but it didn't work out. Instead I ended up doing just that in PicoLisp instead, I guess me and c-lisp was not meant to be. Anyway, PicoLisp is created by Alex Burger without whose help and patience I wouldn't have gone very far.

Actually at first PicoLisp seemed too good to be true, just a few things:

- Good documentation (rare in the Lisp world).
- Object persistence/database.
- Totally dynamically interpreted (no need for macros).
- UTF–8 support out of the box.
- Built in webserver.
- GUI framework to render HTML.
- Mailing function with attachments (SMTP).
- File uploads.

OK so maybe it was a little too good to be true, the documentation is good but covers far from everything, and the reference is what it is, a reference for people who already know the language to some extent. The GUI framework is

tailored to Alex's work which is administrative programs for big corporations which differs from what we do here. Most of the stuff you might want to change is made in PicoLisp though so it won't be very hard to change, no need to touch C. As far as the documentation goes; I will try and remedy the situation somewhat in the near future, the more I learn the more I can teach.

For some time now I've struggled with PicoLisp and it gets easier every day, my C-mind is slowly expanding. It has been painful, and still is, but it is worth it http://www.prodevtips.com/wp-includes/images/smilies/icon_smile.gif

So having said the above I could go directly to fast-explaining what I've done so far which is ye old registration form. Just like I do with my PHP stuff. That would, understandably, be totally useless since we're talking about a language with an extremely small adoption, even counting all the people who are fluent in other Lisps.

This will instead be a new series for absolute PicoLisp beginners, just like I was. I will contrast PicoLisp against PHP in the examples to make it easier for c-people to understand. At the end of this series will be the explanation of how the registration form works, hopefully by then it will be easily understood.

Disclaimer: This series will only be about PicoLisp, the content in the tutorials might or might not be applicable to other Lisps.

29

Registers and Quoting in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

29.1 Install and Start

By now you should have compiled and installed PicoLisp as per the install instructions. We will start the interpreter with **./dbg**. You can create a file and just copy paste the tutorial snippets and run them in the interpreter with : (load "tutorial.l"). To rerun you hit "Esc" followed by "k" to step through prior commands, hit "Enter" when you see the load command.

29.2 The car and the cdr

As in other Lisps the `Contents of Address Register (CAR)` and `Contents of Decrement Register (CDR)` is at the heart of the language. You might also want to check out Lisp on Wikipedia before you continue. Also be sure to check out the naming conventions.

Let's try it out:

```
(setq *Greeting (list "Hello" "how" "are" "you" "doing?"))
(prin (car *Greeting))
```

`Setq` will put the list created with list in the variable `Greeting`. Now run the above code again but substitute `car Greeting` with `cdr Greeting`. As you see this is basically the *key =>value* system of PHP.

Let's create a simple table/2D array:

```
(setq *Fruits
      (list
       (list "green" "apple" "guava" "avocado")
       (list "red" "cherry" "apple")))

(prin (car *Fruits))
```

As you can see you will get the whole first list from the last `car` command, not really surprising if we follow the logic from the first example, this whole list will basically be the key that was represented by "Hello" above. Try writing

```
(prin (car (car *Fruits)))
```

on the last line instead. As you see this will return "green" which is the `car` of the first list which in turn is the car of the whole table. These double, triple and quadruple car/cdr calls have shortcuts. Try

```
(prin (caar *Fruits))
```

instead and you will get the same result. See what you get when you try `cdr`, `cdar`, `cadr`, `caadr` and `cdadr`. Which combinations are they shortcuts for? Take your time to learn how these register functions applies to various list configurations, the time will be well spent because these things are used all the time. They're everywhere.

As it happens there is a shortcut for retrieving the contents of any key even if it's in place 100 down the line. Try substituting the last line with

```
(prin (assoc "red" *Fruits))
```

instead. As you can see the whole list with red fruits and the key (car) is returned. This is normally not what you want when you make a call like that, you don't want the key too, only the fruits. But armed with our knowledge of how the whole `car` and `cdr` thing works we quickly do the following:

```
(prin (cdr (assoc "red" *Fruits)))
```

That's better, we now get all red fruits, and only the fruits.

29.3 Quoting

Try this:

```
(set '*Greeting '("Hello" "how" "are" "you" "doing?"))
(prin (car *Greeting))
```

Different than above but the result is the same, it's easy to see that `setq var` is just a shortcut for `set var`. The is a shortcut for **quote**, everything quoted is taken literally. Variable names inside a quoted list will of course not expand into their values - except when when evaluated or passed to a family of special functions, **mapcar** (described below) is one of them.

Evaluation example:

```
(setq *Prin 'prin)
(eval '(*Prin "hello"))
```

However, when not evaluated:

```
(setq *Prin 'prin)
(prin '(*Prin "hello"))
```

When thinking about quoting it helps - at least for me - to think about the turing machine that accepts instructions in the form of these long paper strips that you feed into it. On the other side you get the result of the computation. When feeding the machine a variable it will of course expand into the strip it contains. When quoting though you simply feed the machine the raw/literal strip, telling it to treat it as such.

There are however exceptions, some functions work on data by reference, just like with the `&` in PHP. Let's look at such an example:

```
(setq *Greeting '("Hello" "how" "are" "you" "doing?"))
(prinl (pop '*Greeting))
(prinl *Greeting)
```

In this case, as you can see the will make the pop function treat Greeting as something passed by reference. Quoted lists can of course be created dynamically and then be passed around and executed to create yet other lists that will be executed in other places in absurdity. If done properly you can in this way utilize the power of Lisp.

As it happens there is a whole category of functions that will accept a function literal (quoted list) and use that function on another list. Let's look at a simple example:

```
(de getSomething (Lst R)
    (mapcar '((Element) (R Element)) Lst) )

(setq *Fruits
      '(("green" "apple" "guava" "avocado")
        ("red" "cherry" "apple")))

(prin (getSomething *Fruits 'car))
```

In this case we will retrieve all keys, basically the same functionality as the PHP function array_keys. If you do cdr you will instead get the values which corresponds to array_values(). What if you want to get the first fruit in each sub-list? Yep you guessed it, just pass cadr instead. Pretty dynamic isn't it? If we return to the Turing machine analogy, this would constitute using a placeholder on a raw strip that will get it's value when it's time to execute the strip/list.

What's happening here is that the function mapcar takes a function literal as it's first parameter, the second parameter is the list the function literal will operate on, actually mapcar can accept several lists and use their contents in the function literal, however in this case we just keep it simple. Element will be each element in the list, in our case the two sub-lists. The result will be a new list whose elements are the results of each operation our function literal performs. That is why we get a list with "green" and "red" in it if we pass car to getSomething.

Notice also that there is no return keyword, in PicoLisp (and all Lisps I think) all expressions return something, hence no need for a return keyword. And the (de keyword is the same as function in PHP, we simply define a new function to be used later. That's all for this time, the next tutorial will probably cover more advanced list manipulation examples.

30

Working with tables in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

30.1 Example Data

Let's examine a common example, a list of people forming a table without key access to each person could look like this:

```
(setq *People
      '(((name . John) (phone . 123456) (age . 56))
        ((name . Fred) (phone . 654321) (age . 35))
        ((name . Fred) (phone . 236597) (age . 38))
        ((name . Hank) (phone . 078965) (age . 23))))
```

30.2 Retrieving data from the table

What you see is they way to specify pairs, each pair is a `car` and a `cdr`. If you recall the way `assoc` worked in the prior tutorial you realize that each sublist can be accessed with that function - since each key is a `car`. With that in mind a solution for retrieving a person by key and value could look like this:

```
(de assoc2d (Lst K V)
    (filter '((Sub)
              (let CurValue (cdr (assoc K Sub))
                (= V CurValue))) Lst ) )

(println (assoc2d *People 'name 'Fred))
```

The above example will retrieve a new list with all persons called Fred. The way this works is through **filter** which is a member of the same family as

mapcar. Filter will return a new list with all items that the callback/literal function returned true for. As in the example in the prior tutorial Sub will be each element, in this case each person.

Next we initiate a temporary variable with `(let CurValue (cdr (assoc K Sub)` *logic using CurValue here*), **let** is a cousin of `setq` but will create it's own little space. Had we already had a variable called `CurValue` with some value in it in the above example - that variable would've gotten a new value inside the let expression. However after the let expression has finished executing, `CurValue` would revert back to it's original value. It's a convenient way of using a variable name temporarily without having to worry about wrecking something else.

Next we use the equal (`=`) function to test if our current value is equal to the passed value in V, since `=` will return `T` (Pico Lisp's equivalent to true) if they are equal, or `NIL` (the equivalent of false) we are done. All sublists without the wanted key/value combination will return `NIL` and will therefore not get a place in the return array.

What if you want to create your own table system, maybe you think the above way of defining a table is too verbose, you want it to look like this instead:

```
(setq *People
      '((name John phone 123456 age 56)
        (name Fred phone 654321 age 35)
        (name Fred phone 236597 age 38)
        (name Hank phone 078965 age 23)))
```

No problem, then you could define the lookup logic like this instead:

```
(de assoc1d (Lst Key)
    (loop
     (NIL Lst NIL)
     (T (= Key (pop 'Lst)) (pop 'Lst))
     (pop 'Lst)))

(de assoc2d (Lst K V)
    (filter '((Sub)
              (= V (assoc1d Sub K))) Lst ) )

(println (assoc2d *People 'name 'Fred))
```

Our custom `assoc1d` function is basically a replacement of `assoc` tailored to our custom table system. It will return the value of the passed key or `NIL` if the key can't be found. You could try it out in isolation:

```
(println (assoc1d '(phone 123456 name John age 56) 'name))
```

The **loop** function will let us loop infinitely and have an arbitrary amount of conditional exits at the same time. We begin by checking if the list (`Lst`) is `NIL` (all elements have been popped off), if that is the case we return `NIL`. If not then we pop an element off and check if it matches the passed key, if yes then we return the next element (the value) by popping it off. If the key didn't match we will continue down the loop conditionals and simply pop the value off without returning it. The equivalent would look something like this in PHP:

```
function assoc1d($lst, $key){
  while(!empty($lst)){
    if($key == array_shift($lst))
      return array_shift($lst);
    else
      array_shift($lst);
  }
  return false;
}
```

In this case the Lisp equivalent isn't really any shorter. However, the alternative to loop would be the use of temporary variables and so on and that is a road we don't want to start walking down.

The `assoc2d` function doesn't present us with any surprises, we simply check each returned value from the `assoc1d` function with the passed value (V).

30.3 Sort the table

So what if we want to sort our table of persons? The solution is similar to sorting tables in PHP. We will extract the values (column) we want to sort by - with the help of the key:

```
(de getCol (Lst K)
    (mapcar '((Sub)(assoc1d Sub K)) Lst ) )

(println (getCol *People 'phone))
```

Let's put it all together:

```
(de getSorting (Sorted Original)
    (make
     (while Original
            (let Value (pop 'Original)
              (setq Sorted
                    (place
                     (link (index Value Sorted)) Sorted NIL ))))))

(de sort2d (L Key)
     (let Col (getCol L Key)
       (mapcar '((Pos)(get L Pos)) (getSorting (sort (copy Col)) Col) )))

(println (sort2d *People 'name))
```

When contemplating problems in Lisp it helps to think in terms of how to generate intermediate lists that are needed to solve the problem. In this case we generate a list describing how to sort our table through the `getSorting` function. That list will be used in the `Sort2d` function to do the actual sorting.

In `Sort2d` we begin by fetching the column represented by the key, in this case we get a flat list of names as they appear in the table (a column). Note that we use **copy** before we sort the first argument to `getSorting`, the reason being that sort will also sort in place, therefore to get two distinct columns in `getSorting` we have to copy one of them, in this case the sorted version. Otherwise they would both be sorted and that would ruin everything.

Right at the beginning of `getSorting` we start with **make** which will initiate a make environment. Inside a make environment there are a few special functions that can be used, the most common one is **link** which we use here. In this case make will return a list created with all arguments to link. It doesn't matter how or where the link is called, whenever and wherever it is called within the make environment will cause it to append another item to the list being made.

Here we will loop through the Original column with unsorted names and pop a name off each time until it's empty. Each name is stored in Value followed by a call to `setq` to change the Sorted column but why? The reason is duplicate names, we need some way of marking already fetched names by setting them to `NIL`, otherwise the **index** function would return the same position for Fred. This would later result in us getting a copy of the first Fred in the sorted table instead of two unique Freds, that is really, really unwanted behavior.

This way of preventing duplicate Freds feels a little bit ugly. In Lisp there is not one way of doing something, in fact there are probably an infinite number of ways of doing something, one better than the other, only the imagination and cleverness of the programmer sets the limits. And unfortunately my limit

was reached here, but somehow I realize that there probably is a better way ...

Anyway, it's the **place** function that is responsible for replacing names whose positions we have already linked with `NIL`, the above way of doing this is possible because link returns the value it links as well as linking it, thus it can be used with place at the same time. Check out both index, place, make and link in the reference.

The main point with `getSorting` is that we compare the sorted names with the unsorted names and return the position of the original name in the sorted column. These are then used in `((Pos)(get L Pos))` as each `Pos` with **get**. `Get` can be used in a variety of situations, in this case it's simply used as an index lookup, in PHP it might have looked like $L[$Pos].

Don't like the way the table got sorted? No problem, try calling `Sort2d` like this instead:

```
(println (flip (sort2d *People 'name)))
```

Yep, **flip** will reverse the list.

Maybe a bit late but: This is probably not the way you would be working with records in a "sharp" situation. Every person would then be a database object and sorted upon retrieval, don't get mad though. The above was a good exercise in any case.

Note also the use of **name**, as you can see it seems like it's a reserved word, still we are able to use it by quoting it.

31

Simple OO in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

31.1 Defining classes

PicoLisp has a very nice object system which will take some time to explore, let's begin with simple examples and work towards more complex scenarios.

```
(class +Person)

(dm hello> ()
    (prin "hello"))

(hello> '+Person)
```

In this example we do not instantiate an object, in PHP the last call would correspond to `Person::hello()`.

```
(class +Person)

(dm setHello> (HelloPhrase)
    (=: hello HelloPhrase))

(dm hello> ()
    (prin (: hello)))

(setHello> '+Person "Hello how are you?")

(hello> '+Person)
```

There are two shortcuts at work here, **=:** will **put** the contents of HelloPhrase into the member variable hello, **:** will in turn **get** it.

31.2 Creating instances

Let's start creating instances:

```
(class +Person)

(dm T (Age Name)
    (=: age Age)
    (=: name Name))

(setq *John (new '(+Person) 65 'John))

(show *John)
```

So the first argument to **new** is a quoted list with the class we want to use. The constructor is the `T` method, that is a requirement, it always has to be called `T` for PicoLisp to notice it correctly.

Add the above hello functions to the John code above and try:

```
(setHello> *John "Hello how are you?")

(hello> *John)
```

As you can see the result is the same, in this regard PicoLisp is similar to PHP:

```
class Person{

  static $hello;

  function __construct($age, $name){
    $this->age = $age;
    $this->name = $name;
  }

  static function setHello($HelloPhrase){
    self::$hello = $HelloPhrase;
  }

  static function hello(){
    echo self::$hello;
  }
}
```

```
$john = new Person(65, "John");

$john->setHello("Hello how are you?");

$john->hello();

Person::hello();
```

The only difference is that PHP requires us to declare the `$hello` variable in order for it to be used in subsequent functions. A requirement which makes static usage less useful in PHP than in PicoLisp.

31.3 Fetch from and sort a list of objects

Let's see how we can fetch from and sort a list of persons:

```
(setq *Persons (list
               (new '(+Person) 65 'John)
               (new '(+Person) 38 'Fred)
               (new '(+Person) 41 'Annie)
               (new '(+Person) 42 'Sam)))
```

This is actually how it could look after you get a list of people from a database which makes this example more useful than the stuff we did in the prior tutorial in this series.

```
(de assoc2d (Lst Key Value)
    (filter '((Sub)(= (get Sub Key) Value)) Lst))

(show (car (assoc2d *Persons 'name 'John)))
```

As you can see **get** can be used to get member variables in objects as well as by index as in the prior tutorial.

Sorting is similar to what we have already done in the prior part, in fact `getSorting` doesn't have to be changed at all, neither does `sort2d`:

```
(de getCol (Lst K)
    (mapcar '((Sub)(get Sub K)) Lst ) )

(de getSorting...

(de sort2d...

(show (car (sort2d *Persons 'name)))
```

Try removing the car in the last call, you will get a list looking something like `($34567855 $68904356 $21345679 $56854378)`. Show will only print their addresses when accessing objects indirectly as is the case here when they are in a list. I leave it as an exercise to write a function that loops through the list and shows each person.

32

More OO in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

32.1 Simple single inheritance

Let's first look at a simple single inheritance example:

```
(class +Species)

(dm T (Species)
    (=: species Species))

(class +Person +Species)

(dm T (Age Name)
    (=: age Age)
    (=: name Name)
    (super "H. sapiens"))

(setq *John (new '(+Person) 65 'John))

(prin (get *John 'species))
```

Nothing really surprising here, the hierarchy is set from left to right in the class definition, that's why `+Person` comes before `+Species`: `(class +Person +Species)`.

```
(class +Animal +Species)

(dm T (Age Name . @)
    (=: age Age)
    (=: name Name)
    (super (car (rest))))

(setq *John (new '(+Animal) 25 'John "G. gorilla"))

(prin (get *John 'species))
```

So John is now a gorilla instead. It's starting to get interesting, the above way of doing the argument list will enable us to pass a variable amount of arguments, the ones ending up in the `@` can be retrieved with **rest**, in this case it's the third one, "G. gorilla". Note that rest will retrieve a list, even if there is only one argument left in it, hence the use of car in this case to get only the first element. In this case "G. gorilla" will be passed to the constructor of `+Species` through **super**.

There is however at better way of doing the argument handling than `(car (rest))`:

```
(class +Species)

(dm T (Species . @)
    (=: species Species)
    (=: description (next)))

(class +Animal +Species)

(dm T (Age Name . @)
    (=: age Age)
    (=: name Name)
    (super (next) (car (rest))))

(setq *John (new '(+Animal) 25 'John "G. gorilla"
      "Big leaf eating primate"))

(prin (get *John 'description))
```

That's right, **next** will retrieve the next value from **rest** and in the process remove the value by reference as demonstrated by the fact that `(car (rest))` gives the proper result in this case. You would however want to use just another `(next)` instead in a situation like the one above.

32.2 Multiple inheritance

Let's take a look at a more complex "horizontal" example, for another example check out the OOP section in Alex's Pico Lisp tutorial. I found it helpful, even in the very beginning.

```
(class +Species)
(dm T (@)
    (=: species (next)))

(dm show> ()
    (pack " Species:" (: species)))

(class +Body)
(dm T (Age Weight Height . @)
    (=: age Age)
    (=: weight Weight)
    (=: height Height)
    (pass extra))

(dm show> ()
    (pack " Age:" (: age) " Weight(kg):
     " (: weight) " Height(cm):" (: height) (extra)))

(class +Person)
(dm T (Name Occupation . @)
    (=: occupation Occupation)
    (=: name Name)
    (pass extra))

(dm show> ()
    (pack " Name:" (: name) " Occupation:" (: occupation) (extra)))

(setq *John (new '(+Person +Body +Species)
      "John" "Teacher" 65 85 180 "H. Sapiens"))

(prin (show> *John))
```

In this case John is the combination of three different classes at once and the way to call the next function in the horizontal hierarchy (from left to right) is to use **extra**. In this case **pass** is a shortcut for sending **rest** to the next constructor. Let's introduce a new class:

```
(class +Location)
(dm T (Location . @)
    (=: location Location)
    (pass extra))

(dm show> ()
    (pack " Location:" (: location) (extra)))

(setq *John (new '(+Person +Body +Location +Species)
 "John" "Teacher" 65 85 180 "New York" "H. Sapiens"))

(prin (show> *John))
```

In this case the two middle classes `+Body` and `+Location` are interchangeable:

```
(setq *John (new '(+Person +Location +Body +Species)
 "John" "Teacher" "New York" 65 85 180 "H. Sapiens"))
```

This is basically the same thing since it's not a hierarchy in the traditional sense, the two middle classes do not have to know what is behind and after in the chain.

This way of using chained relations is important, it is used for instance in the GUI framework to validate forms by simply having a `chk>` function that uses `(pass extra)` to walk the chain, each check is of course unique for each input type, `+TextField` and `+NumField` are two examples.

32.3 Class extension on demand

Classes can be extended on demand:

```
(setq *John (new '(+Person +Body +Location +Species)
"John" "Teacher" 65 85 180 "New York" "H. Sapiens"))

(extend +Body)
(dm bmi> ()
    (*/ (: weight) 10000 (** (: height) 2) ))

(prin (bmi> *John))
```

The `**/*` function is necessary to handle cases like this in order to get the proper result by first multiplying the weight with 10000 and then dividing

that result with 180*180. PicoLisp doesn't handle intermediate floating point numbers automatically. If you wanted an output with one decimal for instance you could do:

```
(dm bmi> ()
    (format (*/ (: weight) 100000 (** (: height) 2)) 1))
```

In this case **format** will take the number 262 and turn it into 26.1. A more on the fly method of accomplishing the above would be:

```
(setq *John (new '(+Person +Body +Location +Species)
"John" "Teacher" 65 85 180 "New York" "H. Sapiens"))

(push *John
      '(bmi> () (format (*/ (: weight) 100000 (** (: height) 2)) 1)))

(prin (bmi> *John))
```

Or maybe the `bmi>` method is already part of some old class in some library and now our program discovers that John needs that class too:

```
(class +WeightHandler)
(dm bmi> ()
    (format (*/ (: weight) 100000 (** (: height) 2)) 1))

(setq *John (new '(+Person +Body +Location +Species)
"John" "Teacher" 65 85 180 "New York" "H. Sapiens"))

(unless (method 'bmi> *John) (push *John '+WeightHandler))

(prin (bmi> *John))
```

In this case **method** will return `NIL` if John doesn't already have the ability to calculate his BMI, in that case we simply push the `WeightHandler` class in front of his other classes.

I don't think I've ever experienced a more flexible object system.

33

Simple OODB in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

33.1 Walk through a simple example

Let's walk through a simple and usual example, what would constitute a user table in MySQL:

```
(class +User +Entity)
(rel username (+Need +Key +String))
(rel password (+Need +String))

(pool "users.db")

(new! '(+User) 'username "sam" 'password "parrotno5")
(new! '(+User) 'username "fred" 'password "MegaPizza")
(new! '(+User) 'username "anna" 'password "swooosh")
(new! '(+User) 'username "fred" 'password "yiiihaaa")
(new! '(+User) 'username NIL 'password "asdf")

(mapcar show (collect 'username '+User))
```

Output:

```
{6} (+User)
   password "swooosh"
   username "anna"
{5} (+User)
   password "MegaPizza"
   username "fred"
```

```
{2} (+User)
   password "parrotno5"
   username "sam"
-> ({6} {5} {2})
```

That was quite a lot at the same time. All classes that are to generate objects stored in the database needs to be children of `+Entity`. Furthermore some relations are needed in the form of prefix classes, **rel** will in this case take the name of the relation, username, and the list of classes that will define the behavior of the relation. Prefix classes has been explained in the prior tutorial.

In the case of the username we have `+Need` which denotes that this relation is needed for successful creation of the persistent object. As you can see in the output the last call to **new!** (only difference from **new** is that we create the object in a file instead of in the RAM) never resulted in an object on disc since no key was created. In our case the username will be used as key and needs to be unique, hence no second "fred" in the output. Of course both username and password will both be strings. There is a short description of more relations in the reference.

33.2 External symbols

Instead of the above run this (don't delete `users.db`!):

```
(class +User +Entity)
(rel username (+Need +Key +String))
(rel password (+Need +String))

(pool "users.db")

(show '{2})
```

This will give us "sam" which means that he is directly accessible through {2} which Alex explains better than me:

> External symbol names are surrounded by braces ('{' and '}'). The characters of the symbol's name itself identify the physical location of the external object. This is currently the number of the starting block in the database file, encoded in base–64 notation (characters '0 through 9, :' through ;', 'A' through 'Z' and 'a' through 'z').

Instead of `(show {2})` try:

```
(show (db 'username '+User "sam"))
```

This is basically the equivalent of:

```
SELECT * FROM `user` WHERE BINARY username = 'sam'
```

And

```
(show (db 'username '+User "fred" 'password "MegaPizza"))
```

is of course the same as the login SQL:

```
SELECT * FROM `user` WHERE BINARY username = 'fred'
AND BINARY password = 'MegaPizza'
```

34

Advanced OODB in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

34.1 Assumptions

In this tutorial I will assume you've already glanced at the documents I linked to in the prior article. I hope you still have cms.db intact.

```
# Copy paste relations and classes from the prior
# article here, nothing has changed

(pool "cms.db")

(?
 (select (@A)
         ((tag +Tag "pc" (tags +Article)))
         (tolr "computer" @A body)
 (show @A)))
```

34.2 Using select

A lot easier than the approach we employed in the prior tutorial, **select** will first take generators, in our case only `(tag +Tag ‘‘pc’’ (tags +Article))`. It will go through the tags and when we find one with tag "pc" we will continue and retrieve all articles connected through the tags reference. Next comes an arbitrary amount of filter clausfes, in our case only one: `(tolr ‘‘computer’’ @A body)`. **tolr** is a shortcut for tolerant which means we do partials too. Try replacing "computer" with "comp" and the result will be the same.

The result is of course a list with all articles who are tagged with "pc" and contain the substring "computer" in their article bodies.

```
(?
 (select (@A)
         ((tag +Tag "pc" (tags +Article)))
         (tolr "computer" @A body)
         (same "sam" @A author username)
         (same "tech" @A folder slug)
 (show @A)))
```

The only addition here is more filtering through the author and folder references. We now get a list of all articles tagged with "pc", written by Sam, containing "computer" in their bodies and located in the "tech" folder, feel free to contemplate the equivalent SQL ...

34.3 Pilog example

34.3.1 Select and insert

As you might know already this all works through Pilog which is a PicoLisp implementation of Prolog. To understand how it works let's play around a little with **be** and a Pilog version of the SWI-Prolog tutorial:

```
(be Indian (vindaloo))
(be Indian (dahl))
(be Indian (tandoori))
(be Indian (kurma))
(be mild (dahl))
(be mild (tandoori))
(be mild (kurma))
(be Chinese (chow-mein))
(be Chinese (chop-suey))
(be Chinese (sweet-and-sour))
(be Italian (pizza))
(be Italian (spaghetti))

(be likes (Sam @F) (Indian @F) (mild @F))
(be likes (Sam @F) (Chinese @F))
(be likes (Sam @F) (Italian @F))
(be likes (Sam chips))

(? (likes Sam @F))
```

Yep, Sam likes Indian food but only the mild curries, vindaloo doesn't fall into that category, that's why it's missing in the output. This is the mechanism

behind our OODB queries. The **same** and **tolr** keywords we use above are in fact set with **be** in pilog.l.

Let's continue with some simple pagination:

```
(new! '(+Article) 'slug "new-pcs-in-2008" 'headline "New PC's in 2008"
'body "An article about all the new PC's in 2008." 'author (db 'username '+User "sam"))

(setq *Query (goal '(@Headline "2008" (db headline +Article @Headline @A))))

(do 2 (bind (prove *Query) (println (get @A 'headline))))
```

There are two new things here **goal** and **prove**. Until now we have used the shortcut **?** to do both at the same time. Goal will prepare a Lisp statement by turning it into a valid query that the Pilog engine can prove or disprove like we are doing above with Sam and his food. Try printing `*Query` to see what it looks like. In this case repeated calls to the last line will retrieve the results two by two because prove will return the next result which makes it ideal to call repeatedly to get the next two and then the next two and so on. Try this instead:

```
(setq *Query (goal '(@Headline "2008" (db headline +Article @Headline @A))))
(do 1 (bind (prove *Query) (println (get @A 'headline))))
(do 1 (bind (prove *Query) (println (get @A 'headline))))
```

In a "sharp" situation we could have called that last line to fetch the next result when our user presses a next button for instance. Notice also the necessary "preparation" of 2008 with `@Headline` at the beginning of the quoted list we pass to goal.

34.3.2 Updating and Deleting

Until now we have only selected and inserted things, let's look at ways to change and delete our data. As you know most of our articles are tagged with "fun", this is how we could remove that tag from our tech folder/article:

```
(del!> (db 'slug '+Article "tech") 'tags (db 'tag '+Tag "fun"))
(mapcar show (collect 'slug '+Article))
```

Note how **del!>** automatically deletes the fun tag from the reference list in the tech article. Updating a pure value is just a matter of putting again:

```
(put!> (db 'slug '+Article "tech") 'headline "The technology folder")
(mapcar show (collect 'slug '+Article))
```

Let's get rid of the fun tag altogether:

```
(lose!> (db 'tag '+Tag "fun"))
(mapcar show (collect 'slug '+Article))
```

The tag is gone but the references are still there, in my case the fun tag was {P} and the {P} still shows in the tag list of each article. So we have a case of orphaning, sometimes it's a wanted behavior, not now though so let's get rid of the reference:

```
(for Article (collect 'tags '+Article '{P})
     (put!> Article 'tags (delete '{P} (get Article 'tags))))

(mapcar show (collect 'slug '+Article))
```

The **for** loop is the PicoLisp version of the old "for in" or "for each". We collect all articles that are referring to the fun tag ({P}). After that we get the tag list in question, delete the fun reference and finally put it back. With that in mind we could create a custom **lose** method:

```
(extend +Entity)
(dm loseref!> ()
    (for Child (var: Cascade)
         (let (ChildClass (car Child) ChildRef (cdr Child))
           (for Element (collect ChildRef ChildClass This)
                (put!> Element ChildRef (delete This (get Element ChildRef))))))
    (lose!> This))

(class +User +Entity)
(rel username (+Need +Key +String))
(rel password (+Need +String))

(class +Article +Entity)
(rel slug     (+Need +Key +String))
(rel headline (+Need +Idx +String))
(rel body     (+Need +String))
(rel author   (+Ref +Link) NIL (+User))
(rel folder   (+Ref +Link) NIL (+Article))
(rel tags     (+List +Ref +Link) NIL (+Tag))
```

```
(class +Tag +Entity)
(rel tag (+Need +Key +String))
(var Cascade . ((+Article . tags)))

(pool "cms.db")

(loseref!> (db 'tag '+Tag "pc"))

(mapcar show (collect 'slug '+Article))
```

This is just repetition of the above with the addition of a Cascade list that we loop through to find which classes are affected (in our case only `+Article`) and the name of the reference to use `(tags)`. Note the use of *class variables* (which I forgot to mention in the simple OO tutorial). We initiate a class variable with **var** and retrieve it with **var:**.

That was one way of doing it, another is to inspect the relations and use that information to do the cleanup. The problem with this is that it will delete all references in all tagged objects. Pretend you had something else in the system that you are tagging, `+Novel`(s) for instance. If you only wanted to remove the specific tag for articles, not novels you would have to specifically state that somewhere and you are back to something like the above. However, if this is not a problem you could do like this instead:

```
(extend +Entity)
(dm loseref!> ()
    (for Child (getRefs> This)
         (let (ChildClass (car Child) ChildRef (cdr Child))
           (for Element (collect ChildRef ChildClass This)
                (put!> Element ChildRef (delete This (get Element ChildRef))))))
    (lose!> This))

(dm getRefs> ()
    (make
     (for Class (all)
          (when (isa '+Entity Class)
            (for El (getl Class)
                 (and
                  (isa '(+Ref +Link) (car El))
                  (= (list *Class) (get El 1 'type))
                  (link (cons Class (cdr El)))))))))
```

```
# Relations here without the (var Cascade . ((+Article . tags))) line.

(pool "cms.db")

(loseref!> (db 'tag '+Tag "scuba"))

(mapcar show (collect 'slug '+Article))
```

`GetRefs>` will loop through all symbols currently loaded, when the symbol is an `+Entity` we fetch the whole property list from the symbol.

We loop through all properties and check if they have `+Ref +Link`, if yes we check if the current class accessed through the `*Class` global is equal to the type we fetch from the `car` of `El`, yes `(get (car El) type)` would have worked too. If they are equal we move on and **link** a **cons** pair to the list.

We get the name of the relation with `(cdr El)`, the result is identical to the explicitly set `((+Article . tags))` in the prior example.

35

Registration Form in PicoLisp

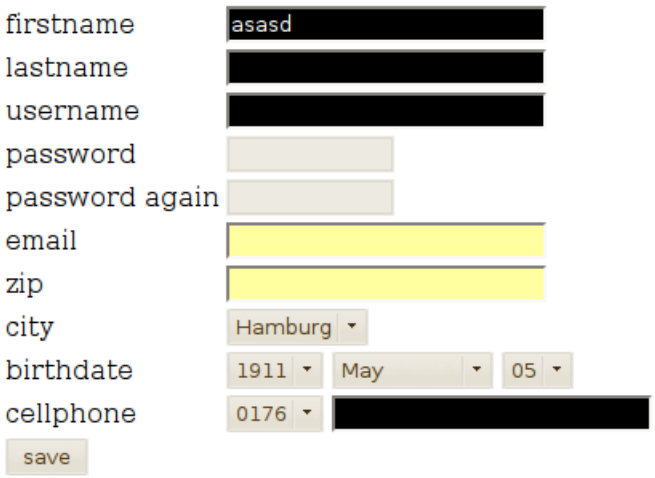
Henrik Sarvell

hsarvell@gmail.com

35.1 Prerequisites

Finally, the registration form! Not really for beginners but anyway

If you are new to Pico Lisp you might want to check out the first article in the series and move "upwards". Apart from that this tutorial builds upon two earlier articles. Regular expressions and templating.



I've changed the background color to black and the text to white just as a test to see if the CSS was loading properly. Let's walk in order of execution, first out is **main.l**:

35.2 Walk through the `main.l` library

```
(load "lib/http.l" "lib/xhtml.l" "lib/form.l" "lib/ps.l"
 "lib/adm.l" "lib/misc.l" "lib/rgx.l" "lib/tpl.l")
(setq *BP "projects/tpl-test/")
(setq *Css (pack *BP "css/styles.css"))
(load
 (pack *BP "models/er.l")
 (pack *BP "helpers/global-helpers.l"))

(de main ()
  (pool (pack *BP "db/test.db")))

(de start ()
  (app)
  (setq Tpl (new '(+Tpl) *BP))
  (assign> Tpl 'title "Registration Form")
  (parse> Tpl 'index)
  (compRun> Tpl)
  (out> Tpl))

(de go ()
   (server 8080 "@start"))
```

So apart from the libraries we load **er.l** and **global-helpers.l**, **rgx.l** and **tpl.l** are the two libraries whose explanations I link to above.

We load the database with pool initially because we start the server with: **.p/dbg.l projects/tpl-test/main.l -main -go.**

And go is of course responsible for starting the server on `port 8080` and running start.

The start function will first start the session with app, create the template object with our base path `*BP` and assign a title, parse the template, compile it, run it and finally print it to the browser.

Let's take a look at the template:

```
<html>
<head>
<title> <% get title %> </title>
<base href="<% bPath %>"/>
<link rel="stylesheet" href="<% path cssDir %>styles.css" type="text/css" >
</head>
<body>
<% gui reg-form %>
</body>
</html>
```

There are some new things here since we went through the template class, but not much:

```
(dm path> (Var)
   (pack (srcUrl) (get This Var)))

(dm bPath> ()
   (baseHRef))
```

The result could look like this:

```
<base href="http://localhost:44148/"/>
<link rel="stylesheet"
      href="http://localhost:8080/projects/tpl-test/css/styles.css"
type="text/css" >
```

So `baseHRef` outputs http://localhost:44148 and *srcUrl*http://localhost:8080/, great. The reason for the base tag is that the GUI framework needs it. The port number keeps track of each session and is unique for that nsession.

35.3 Walk through the `er.l` library

Before we start with the form itself let's go through the **er.l** file and **global-helpers.l**, first `er.l`:

```
(extend +Entity)
(dm asSelect> ()
    (collect (: lbl) This NIL T (: lbl)))

(class +Member +Entity)
(rel fname    (+Need +Sn +Idx +String))
(rel lname    (+Need +Sn +Idx +String))
(rel uname    (+Need +Key +Sn +Idx +String))  #min 6 chars
(rel pwd      (+Need +String))                #min 6 chars
(rel zip      (+Need +Idx +String))           #min 5 chars, numerical
(rel city     (+Link)(+City))                 #lives in a city
# min 7 digits for validation but we store min 11 digits
# (as is, no loc formatting)
(rel cellnr   (+Need +Ref +String))
# has to validate as proper email address
(rel email    (+Need +Key +Idx +String))
(rel bdate    (+Need +Ref +Date))             #birthdate



(class +CellPrefix +Entity)
# will be used in the registration to create a complete cell number
(rel nr (+Ref +String))
(var lbl . nr)

(class +City +Entity)
(rel nm (+Ref +String))
(var lbl . nm)
```

We've got a link to `+City` in the `+Member`. The `asSelect>` method is responsible for fetching a list to be used as a drop down. In the `+CellPrefix` case it is **nr** of course, and in the `+City` case it's **nm**. Actually pretty redundant since they only have one relation but that could quickly change.

35.4 Walk through the `global-helpers.l` library

`Global-helpers.l` contains some extra validation logic and generators:

```
(class +Gh)

(dm range> (Start End Pad)
    (make
     (for (N Start (>= End N) (inc N)) (link (pad Pad N)))))

(dm getMonths> ()
    (range> This 1 12 2))

(dm getDays> ()
    (range> This 1 31 2))

(dm getYears> (Min-age)
    (let curYear (curYear> This)
      (range> This (- curYear 100) (- curYear Min-age) 0 )))

(dm curYear> () (car (date (date))))
```

Trivial stuff to generate various drop downs. Note `pad` to get `01`, `02` etc.

```
(class +EmailField +TextField)

(dm chk> ()
  (ifn
     (match> '+Rgx (super) '((word > 0) "{at}" (dmn > 0) "." (ltr > 2 < 5)))
     ,"email-expected"
     (super)))

(class +AlNum +TextField)

(dm chk> ()
  (ifn (alnum> '+Rgx (val> This)) ,"alnum-expected" (super)))
```

So we create a new `+EmailField` and `+AlNum` based on the basic `+TextField`. The main thing here is the `chk>` method that is called automatically by the GUI logic in order to check if a value is OK or not. The `+Rgx` stuff has already been covered. Note the comma (`,`) It's a shortcut for the localization logic, it will lookup the keys for translations in translation files if they are provided, currently not though. Check out the the OO tutorial for more info on `super` above, and `extra` used below.

```
(class +MinLen)

(dm T (MinLen . @)
  (=: minLen MinLen)
  (pass extra))

(dm chk> ()
  (ifn
     (>= (length (val> This)) (: minLen))
     (pack ,"minlen-1" (: minLen) ,"minlen-2")
     (extra)))
```

The minimum length logic, it does not inherit from something else but is setup to work with other classes in a horizontal fashion. Note the use of `,minlen--1 (: minLen) ,minlen--2`. We need this to account for differing minimum lengths in the error message.

```
(class +PwdCheck)

(dm T (PwdGet . @)
  (=: pwdGet PwdGet)
  (pass extra))

(dm chk> ()
  (ifn (= (eval (: pwdGet)) (val> This)) ,"password-mismatch" (extra)))
```

Also designed to be prefix class only. We simply check if the passwords match.

35.5 The registration form

It's time for the registration form which is a biggie, I've uploaded it here if you want to see the whole thing at once. It will be chopped up below in a from top to bottom fashion.

```
(action
    (let E (loc "*Err" err)
       (set E (head 1 (val E))))
    (form NIL
       (unless *Post (=: obj (new! '(+Member))))
       (<table> NIL NIL NIL
          (<row> NIL ,"firstname" (gui '(+E/R +TextField)
                '(fname : home obj) 10))
          (<row> NIL ,"lastname"  (gui '(+E/R +TextField)
                 '(lname : home obj) 10))
          (<row> NIL ,"username"  (gui '(+E/R +MinLen +AlNum)
                 '(uname : home obj) 6 10))
```

Having everything be an argument to the action function is a requirement for the GUI to work.

The first thing we do is override the default behavior of displaying validation errors for all fields at once. We get the `*Err` list from the err function defined in `form.l` line 207. After that we extract and use the first error (if there are any).

The second thing we do is to build an empty `+Member` object which will be available through `(: home obj)`. You might notice the use of **new**!, what happens if the user just navigates away you might think, there will be a lot of empty bullshit in the database won't it? Sure, and they can be cleared out by running a cron job every day or continuously although it consumes more resources. Since we work with a new! object there will be no need for future commits, from now on everything that happens happens on disk, not memory.

We build the layout in a table where each call to `<row>` will create new rows and tds. Notice the calls to `gui`, apparently `+E/R` needs the `(fname home obj)` list and `+TextField` the 10 number (the size of the field).

Our first custom prefix classes comes in the form of `+MinLen` and `+AlNum` which takes 6 and 10 respectively (remember that +AlNum is just a `+TextField` with extra validation attached to it).

```
(<row> NIL ,"password"
   (gui
      '(+E/R +PwdCheck +MinLen +PwField)
      '(pwd : home obj)
      '(val> (: home pwd2)) 6 10))
(<row> NIL ,"password again" (gui 'pwd2 '(+PwField) 10 ))
(<row> NIL ,"email" (gui '(+E/R +EmailField)       '(email : home obj) 10))
(<row> NIL ,"zip"   (gui '(+E/R +MinLen +NumField) '(zip : home obj) 6 10))
(<row> NIL ,"city"  (gui '(+E/R +TextField)        '(city : home obj) (asSelect> '+City)))
```

Note that the code has been indented to save horizontal space, in reality it's just a continuation of the above code.

`+PwdCheck` will work with `(val> (: home pwd2))` as input, yes it's the value of the "password again" field.

The city `+TextField` will get the result of the `(asSelect> +City)` call as it's input, note the lack of a quote (). So we evaluate and get the list of cities which will turn the field into a drop down instead of a normal text field.

```
(<row> NIL ,"birthdate"
   (<div>
      (gui '(+E/R +Fmt +Chart) '(bdate : home obj)
         '((Dat) (list (date Dat)))
         '((Lst) (and (caar Lst) (cadar Lst) (caddr (car Lst)) (date (car Lst))))
         3 )
      (gui 1 '(+TextField) (getYears> '+Gh 18))
      (gui 2 '(+Map +TextField)
         (make
            (for (I . M) *MonFmt
               (link (cons M I))))
         *MonFmt)
      (gui 3 '(+TextField) (getDays> '+Gh))))
```

It's starting to get tricky. Remember that we use `+Date` for the birth date so we need some way of getting it to prepopulate three consecutive drop downs, and be inserted by getting the data in all of them. Actually we are just updating the empty object we created at the top all the time but insert might be a more proper wording since we are replacing nothing with something.

The `+Fmt` is class will take care of getting and setting the date, the first function `((Dat)...)` will `set>` and the second one `((Lst)...)` will be our `val>` function.

The `+Chart` prefix is responsible for handling our 3 date drop downs and will therefore take 3 as its single argument. The constituent gui elements need to know where they are in this list, hence 1, 2 and 3 as the first argument. The second drop down will display the months of the year by working with the global `*MonFmt` which will contain the full names based on which location we have selected. Since we haven't chosen a location they will default to their English names. Come to think of it this piece of code should have been put in global helpers since month drop downs are pretty common.

```
(<row> NIL ,"cellphone"
   (<div>
      (gui '(+E/R +Chart) '(cellnr : home obj) 2
         '((Str)
            (let Len
               (length
                  (Find '((P) (pre? P Str)) (asSelect> '+CellPrefix)))
               (setq Str (chop Str))
               (list
                  (list (pack (cut Len 'Str)) (format (pack Str))))))
         '((Lst) (pack (caar Lst) (cadar Lst))))
      (gui 1 '(+TextField) (asSelect> '+CellPrefix))
      (gui 2 '(+MinLen +NumField) 7 10 )))
(<row> NIL (gui '(+Button) ,"save" '(url "@start"))))))
```

Here we pass two extra set and get functions to the `+Chart` class directly instead. The first one is needed due to the fact that it will recieve for instance a string looking like `01766587436`. It then needs to cut the string up to get a list looking like this: `(0176 6587436)`, in order to repopulate on for instance validation failure. The problem is that some mobile phone prefix numbers can have five digits in Germany, ouch. That's why we need to determine the length first, otherwise we could end up with the wrong numbers.

Finally the `+Button` will post the form back to the start function which in turn contains this function. The error logic at the top of the form could be used to confirm a successful post by displaying a new page or message. No error == success.

36

Explicit Scope Resolution in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

Summary. This article describes how to use explicit scope resolution with `run` and `eval` when extending the `html` function with some arbitrary code as final argument.

36.1 Extending the `html` function

Today I felt like extending the **html** function that is responsible for rendering the bare bones of a HTML document in the GUI framework.

Normally a call to (`html`) looks something like:

```
(html 0 "Pico Admin" *CSS NIL
  (Some code here)
  (Some more code here) ... )
```

The important thing is the fact that the function accepts some arbitrary code(s) as the final argument. I wanted to simply hard code my stuff into an "extension", like this:

```
(de pa-html Prg
  (html 0 "Pico Admin" *CSS NIL Prg))
```

Calling the above is shorter than the original with the same arguments over and over again and `Prg` is now the arbitrary code I referred to above.

36.2 FEXPRs and scoping rules

It wouldn't work though, what was going on? Well the answer is obvious when you get some help and think a little about it. In our **(pa-html)** above the **Prg** variable is a FEXPR; unevaluated code. With our addition we add a new scope, the local scope of (pa-html). It turns out that (html) doesn't expect the code to come from this local scope, a lot of variables that are defined in the scope calling (pa-html) are probably undefined within the local scope of (pa-html) for instance. Hence the massive crashing.

To understand the solution, let's start over from the beginning with the **(up)** function:

```
(let Hello "Hello"
  (println Hello)
  (let Hello "Hi!"
     (println Hello)
     (let Hello "Greetings"
        (println Hello)
        (println (up Hello)))))
```

Output:

```
"Hello"
"Hi!"
"Greetings"
"Hi!"
```

So the call to (up) will check what **Hello** contained in the scope above the scope that it was called in. In this case "Hi!" was the contents. If I say that the default value for (up) is 1 you can probably figure out what the last (println) would've printed had we passed it `(up 2 Hello)` instead.

36.3 Explicit scoping with run and eval

36.3.1 Using run

Both (**run**) and (**eval**) can accept a last optional argument that will define their scope explicitly, let's start with (run):

```
(de Lvl1 (Arg . Prg)
  (let Lvl "lvl: 1"
     (run Prg 1)))

(de Lvl2 ()
  (let Lvl "lvl: 2"
     (Lvl1 "some argument"
        (println Lvl) (println "Something more here") ) ) )

(Lvl2)
```

Output:

```
"lvl: 2"
"Something more here"
```

As you can see **(run Prg 1)** will run **Prg** within the scope of (**Lvl2**), "one up". I think you can guess what the output would be if we just did **(run Prg)**.

By this point you should be able to understand that

```
(de pa-html Prg
  (html 0 "Pico Admin" *CSS NIL (run Prg 1)))
```

is how (`pa-html`) should look.

36.3.2 Using `eval`

Let's finish off with (`eval`) for good measure:

```
(de Lvl1 (Arg Lst)
  (let Lvl "lvl1"
     (eval Lst 1)))

(de Lvl2 ()
  (let Lvl "lvl2"
     (Lvl1 "I'm Arg" '(pack Arg " and I'm: " Lvl))))

(Lvl2)
```

Output:

```
" and I'm: lvl2"
```

Since **Arg** is undefined within the scope of (**Lvl2**) we won't get "I'm Arg" in the beginning of the output. Just calling `(eval Lst)` would of course result in `I'm Arg and I'm: lvl1`.

I think it's a bad idea to rely on explicit scope resolution out of laziness, as you can imagine the code could easily become unintelligible if not used sparingly. Use only when absolutely necessary if you can not see any other solution.

37

Pilog Solve and the +Aux Relation

Henrik Sarvell

hsarvell@gmail.com

Summary. This article uses the 'Doctrine' example (known from the PHP world) to demonstrate the usage (and advantages) of Pilog `solve` and the `+Aux` relation.

37.1 'Doctrine for dummies' example

I just set out to duplicate the Doctrine for dummies example, but this time for real, with a real OODB system, not some silly ORM. Thanks goes to Alex for helping me out with the queries.

```
(class +Member +Entity)
(rel uname  (+Need +Key +String))
(rel pwd    (+Need +String))
(rel email  (+String))
(rel city   (+Ref +Link) NIL (+City))

(class +City +Entity)
(rel name   (+Key +String))

(class +Message +Entity)
(rel subject (+Idx +String))
(rel body    (+String))
(rel from    (+Ref +Link) NIL (+Member))
(rel to      (+Aux +Ref +Link) (from) NIL (+Member))

(pool "dbtest.db")
```

So we create the relations, and the database file. Take special note of the **+Aux** relation from **to** to **from**. It will play a central role later.

```
(setq Mbrs
  (list
     (list "member1" "password1" "member1@members.com" "Berlin")
     (list "member2" "password2" "member2@members.com" "Stuttgart")
     (list "member3" "password3" "member3@members.com" "Hamburg")))

(for Mbr Mbrs
  (request '(+Member)
     'uname (get Mbr 1)
     'pwd   (get Mbr 2)
     'email (get Mbr 3)
     'city  (request '(+City) 'name (get Mbr 4))))
```

We input some members in the database, note the use of **request**. The major thing is that it will create an object if there is none with the given key(s) already, however if there is an object with the given keys it will return that object, quite handy.

```
(setq Mbr1 (db 'uname '+Member "member1"))
(setq Mbr2 (db 'uname '+Member "member2"))

(new T '(+Message)
  'subject "from mbr1 to mbr2"
  'body "Hello mbr2 this is mbr1"
  'to Mbr2
  'from Mbr1)

(new T '(+Message)
  'subject "from mbr2 to mbr1"
  'body "Hello mbr1 this is mbr2"
  'to Mbr1
  'from Mbr2)

(new T '(+Message)
  'subject "from mbr2 to mbr1, again"
  'body "Hello mbr1 this is mbr2, again"
  'to Mbr1
  'from Mbr2)
```

```
(new T '(+Message)
  'subject "from mbr1 to mbr3"
  'body "Hello mbr3 this is mbr1"
  'to (db 'uname '+Member "member3")
  'from Mbr1)

(commit)
```

Note the use of **(new T...)** without the `T` we won't get database objects.

37.2 Querying

37.2.1 Simple queries

```
(mapc show (collect 'to '+Message Mbr1 Mbr1 'from))
(mapc show (collect 'from '+Message Mbr1 Mbr1 'to))
(mapc show (collect 'from '+Message Mbr1 Mbr1))
(mapc show (collect 'to '+Message Mbr1 Mbr1))
```

Test that everything is there by calling the above statements, the first one will get all people who sent a message to member 1. The second one will instead get all people who member 1 sent messages to. The two last ones will do the same but will get messages, not people. And that was that, this is as complex as the PHP Doctrine example got, let's move on to more complex stuff relating to the +Aux relation:

37.2.2 Using the `+Aux` relation

```
(scan (tree 'to '+Message))
(scan (tree 'from '+Message))
```

Output:

```
({5} {:} . {H}) {H}
({5} {:} . {I}) {I}
({:} {5} . {B}) {B}
({A} {5} . {L}) {L}

({5} . {B}) {B}
({5} . {L}) {L}
({:} . {H}) {H}
({:} . {I}) {I}
-> {F}
```

Notice the difference, the **to** relation is utilizing **+Aux** so we have extra references there between the message receiver and sender. Together the combination can be used as a key to speed up things.

```
(mapc show (collect 'to '+Message (list Mbr1 Mbr2)))
```

This is an example of how to use the `+Aux` relation to get all messages to member 1 from member 2 with **collect**.

37.2.3 Pilog `solve` with parallel scanning

```
(mapc show
   (solve
     (quote @M1 Mbr1 @M2 Mbr2 @S "mbr2"
        (select (@Msgs)
           ((subject +Message @S) (to +Message @M1) (from +Message @M2))
           (same @M1 @Msgs to)
           (same @M2 @Msgs from)
           (tolr @S @Msgs subject)))
     @Msgs ))
```

The above code will retrieve all messages from member 2 to member 1 that also contain the word `mbr2` in the subject. It is however not using the `+Aux` relation to make the lookup of who is sending a message to who, we scan in parallel here.

37.2.4 Pilog `solve` using the `+Aux` relation

```
(mapc show
   (solve
      (quote
         @M1 Mbr1
         @M2 Mbr2
         @M (list Mbr1 Mbr2)
         @S "mbr2"
         (select (@Msgs)
            ((subject +Message @S) (to +Message @M))
            (tolr @S @Msgs subject)
            (same @M1 @Msgs to)
            (same @M2 @Msgs from)))
      @Msgs ))
```

This example is however using the `+Aux` relation. The **@M** list will look for for example the **{5} {:}** combo above in the **(to +Message @M)** clause. Next we filter as usual.

After benchmarking the above examples I got a 3% speed increase in favor of the second one using 6000 messages to test on, when using 11000 the difference increased to 11% so careful planning will pay off more and more the bigger the database becomes.

38

PicoLisp and JSON

Henrik Sarvell

hsarvell@gmail.com

Summary. This article discusses libraries and tests for converting PicoLisp to JSON and vice versa.

38.1 Introduction

Yet again I have to do some documenting so I know what the hell I'm doing since I'm all over the place at the moment. Doing something here and then moving over to do something over there and then coming back to coding this and that. If you're this unstructured you need crutches and this documentation is that, it will enable me to get back to this and do easy debugging in the future, it will trigger memories.

Therefore this code is very rough and untested, a work in progress, you have been warned...

If you somehow landed on this page without any background on PicoLisp or Lisp you probably need to start from the beginning.

38.2 The tests

38.2.1 PicoLisp to JSON

Let's start with the tests for a change. This is an example of converting a proper database object to JSON. The code will determine the relations and use them to build the JSON, nothing else that might be in there:

```
(load "lib/str.l")
(load "lib/json.l")

(class +Product +Entity)
(rel name (+Need +String))
(rel id (+Need +Number))
(rel descr (+String))
(rel attributes (+List +String))

(setq Product
   (new '(+Product) 'name
      "A \"PC\"" 'id 123 'attributes '("black" "laptop") ) )

(println (to> '+Json 'encObj> Product))
```

Output:

```
"{\"attributes\": [\"black\", \"laptop\"], \"id\": 123,
  \"name\": \"A \\\"PC\\\"\", \"descr\": false}"
```

Associative structure, it will also be encoded as object:

```
(setq Pairs '((key1 . hello) (key2 . world) (false . NIL)
              (someArr . (1 2 "hello quote: \"quote\"" 4)) ) )
(println (to> '+Json 'encPair> Pairs))
```

```
"{\"key1\": \"hello\", \"key2\": \"world\", \"false\": false,
  \"someArr\": [1, 2, \"hello quote: \\\"quote\\\"\", 4]}"
```

2D structure will be an array of objects:

```
(setq Pair1 '((key1 . hello) (key2 . world) (false . NIL) (someArr . (1 2 "hello" 4))))
(setq Pair2 '((key1 . hello) (key2 . world) (false . NIL) (someArr . (1 2 "world" 4))))
(setq Tst (list Pair1 Pair2))
(println (to> '+Json 'encTable> Tst))
```

```
"[{\"key1\": \"hello\", \"key2\": \"world\", \"false\": false,
  \"someArr\": [1, 2, \"hello\", 4]},
{\"key1\": \"hello\", \"key2\": \"world\", \"false\": false,
  \"someArr\": [1, 2, \"world\", 4]}]"
```

Simple case of **nested list**:

```
(setq Tst (list 1 2 "hello" (1 2 3) 3 4 "world"))
(println (to> '+Json 'encArr> Tst))
```

```
"[1, 2, \"hello\", [1, 2, 3], 3, 4, \"world\"]"
```

38.2.2 JSON to PicoLisp

```
(setq Json "{\"hello1\": {\"subObj\": [123, 456, true, NIL]}, \"b\": true}")
(setq Result (from> '+Json Json))
(show Result)
(show (get Result 'hello1))
```

```
$385543015 NIL
   b
   hello1 $385543062
$385543062 NIL
   subObj (123 456 T NIL)
```

38.3 The library

38.3.1 JSON to PicoLisp

```
(class +Json)

(dm from> (J)
  (=: L (chop J))
  (let C (pop (:: L))
     (let R (if (= C "[") (pre> This 'pArr>) (pre> This 'pObj>))
        (parse> This R))))
```

This is where the coding from JSON to PicoLisp structure begins.

```
(dm pre> (Type)
  (let (R (list Type) InStr NIL)
     (catch NIL
        (while (: L)
           (let C (pop (:: L))
              (cond
                 ((= C "[") (queue 'R (pre> This 'pArr>)))
                 ((= C "{") (queue 'R (pre> This 'pObj>)))
                 ((and (or (= C "]") (= C "}")) (nT InStr)) (throw))
                 (T (when (= C "\"")
                          (setq InStr (not InStr)))
                       (queue 'R C)))))) R ))
```

Here we are creating an intermediary list that will be easy to execute. We do this by inserting the names of functions to use in later steps into this list. But what is really happening? Well as you saw from the prior listing we begin with either **pArr>** or **pObj>** depending on if we are to begin parsing to an object or an array. **InStr** will keep track of whether the characters **{**, **}**, [,] are inside a string or not, if they are they should not count of course.

So while we still have characters in our chopped up list we will loop through them by destructively popping, if we have a "[" we will put **pArr>** on the list instead of the caracter, if we have "{" we put **pObj>**. If we have the respective closing character, and it is not inside a string, we exit by throwing `NIL`.

```
(dm any> (L)
   (let R (any (pack L))
      (if (= R "true") T (if (= R "false") NIL R))))

(dm parse> (L)
  (apply (car L) (list This (cdr L))))

(dm pObj> (L)
  (let (R (new) L (split L ","))
     (for El L
        (let Pair (split El ":")
           (put R
              (any (any (pack (car Pair))))
              (let Value (cdadr Pair)
                 (if (lst? (car Value))
                    (parse> This (car Value))
                    (any> This Value)))))) R ))
```

We begin by applying either **pObj>** or **pArr>** in **parse>**.

The `pObj>` method will begin with creating the empty result object, **R** and a list of sublists looking something like this: ("k" "e" "y" ":" "v" "a" "l" "u" "e") in **L** by splitting by ",".

We continue by splitting by ":" to get the key and the value. The key is then retrieved, the value will be further examined to determine if we should apply recursion to get a sub-object/array or simply return the result through **any>**.

```
(dm pArr> (L)
   (make
      (for El (mapcar 'clip (split L ","))
         (if (lst? (car El))
            (link (parse> This (car El)))
            (link (any> This El))))))
```

38.3.2 PicoLisp to JSON

```
(dm to> (F L)
   (pack (make (apply F (list This L)))))
```

As you know from the tests above the behavior has to be explicitly set by passing the function name to be used to generate the result when going from PicoLisp to Javascript. It's pretty obvious actually since it's impossible to determine from the structure of various types of lists how to treat them. We can't infer whether a list is a normal nested list or a paired list, for all intents and purposes they are identical. However the output will be radically different. Note **make**, that is why we are able to use **link** all over the place below.

```
(dm encTable> (Tbl)
  (link "[")
  (let F T
     (mapc
        '((L)
            (link (comma> This F))
            (encPair> This L)
            (setq F NIL)) Tbl))
  (link "]"))

(dm encPair> (L)
  (link "{")
  (let F T
     (mapc
     '((El)
         (link (pack (comma> This F) "\"" (car El) "\"" ":" " "))
         (setq F NIL)
         (enc> This (if (pair El) (cdr El) NIL))) L ))
  (link "}"))

(dm comma> (First)
   (unless First ", "))
```

Some redundant code here, it might benefit from refactoring, or we could just leave it like it is and call it a day, yeah let's do that. When encoding a table each element will in turn be encoded with **encPair>**, if we have the first element we do not prepend the ",".

A paired list will be encoded as an object with **encPair>**.

```
(dm encArr> (L)
  (link "[")
  (let F T
     (mapc
     '((El)
         (link (pack (comma> This F)))
         (setq F NIL)
         (enc> This El)) L ))
  (link "]"))
```

Redundancy again! `List -> array` is easier though than `paired list -> object`.

```
(dm encObj> (O)
  (encPair> This
     (make
        (mapc
           '((Prop)
               (when (isa '+Relation (car Prop))
                  (let Key (cdr Prop)
                     (link (cons Key (get O Key))))))) (getl (car (type O)))))))
```

Finally something clever, in case of object we will get all the properties of the object through the `getl` function. Every property that is a relation will get the "treatment". We get the name of the relation as **Key** and use that name on the original object to retrieve the value. The resultant array is now a paired list that can be encoded with **encObj>**.

```
(dm enc> (L)
   (cond
      ((=T L)    (link "true"))
      ((not L)   (link "false"))

      ((num? L)  (link L))
      ((lst? L)  (encArr> This L))
      ((or (box? L) (ext? L)) (encObj> This L))))
```

accordingly, notice the escaping with **+Str**. It's the genesis of some kind of general string library, not much in there yet though:

```
(dm fChr> (Lst Chr)
  (find '((C)(= C Chr)) Lst))

#S = hello, Lst = '("\"")
(dm esc> (S Lst)
  (pack
     (mapcar
        '((C)
            (if (fChr> This Lst C) (pack "\\" C) C )) (chop S))))
```

Every character in the passed list (in this case only one) will be escaped.

39

Factorials, Permutations and Recursion in PicoLisp

Henrik Sarvell

hsarvell@gmail.com

Summary. This article describes *factorials*, *permutations* and *recusrion* can be used in the simulation of stock trading strategies.

39.1 Simulating stock trading strategies

Currently I'm simulating trading strategies on historical stock data. Yes I know according to Nassim this is complete bullshit but I might beg to differ. At least I feel the need to determine if it's bullshit on my own than just take his word for it.

I have bought trading data from the SET which includes the years 1975–2008, the period 1997–2001 could possibly closely resemble what we are up against at the moment so any simulated strategy that returns more than simply sitting on the sidelines and doing nothing is a winner and might be worth testing at the moment.

39.2 Factorials and Permutation

39.2.1 First try

However in order to simulate these strategies we need to be able to do permutations and I couldn't find anything already created to this effect, neither could I find a core factorial function, so here they are:

```
(de fac (Num)
   (let Res 1
      (for (N 1 (>= Num N) (inc N))
         (setq Res (* Res N)))))
```

So this one is basically the same as ye old "N!".

```
(de switch (Lst P1 P2)
  (let (V1 (get Lst P1) V2 (get Lst P2))
      (place P1 (place P2 Lst V1) V2)))
```

This one is used in the permutate function below, however it might be of use in a standalone fashion, hence it having its own definition. Anyway the end result is a switch of the values indicated by the numbers in P1 and P2.

39.2.2 Using `recur` and `recurse`

```
(de permutate (Lst)
   (let (Result (list) Count 1 Start 1)
      (recur (Lst Start Result Count)
         (when (>= (fac (length Lst)) Count)
            (push Result Lst)
            (when (= Start (length Lst)) (setq Start 1))
            (recurse
               (switch Lst Start (inc Start))
               (inc Start)
               Result
               (inc Count)))
         (car Result))))
```

Note **recur** and **recurse** here, we might just have created a different non-recursive entry function instead but using these two is a more lazy approach that lets us dispense with the need to create two different definitions.

The end result is a list of lists with all different permutations.

Anyway, I will put this stuff up for inspection on the Pico Lisp mailing list and let more knowledgeable people give feedback, updates with new code and comments will most likely appear here in the near future.

Update: OK so it didn't work, I created the above based on an (1 2 3) example list, however in my sharp application I work with 4 numbers and it didn't manage that, I'll leave it though as an example of how recursion can be done in PicoLisp.

39.2.3 Second try

I ended up stealing one of the algorithms from the permutation Wikipedia article, and this is the result:

```
(de permutation (N Lst)
   (for (J 2 (>= (length Lst) J) (inc J))
      (setq N (/ N (- J 1)))
      (setq Lst (switch Lst (inc (% N J)) J))))

(de permutate (Lst)
   (let Rslt (list)
      (for (N 1 (>= (fac (length Lst)) N) (inc N))
         (push Rslt (permutation N Lst)))
      (uniq (car Rslt))))
```

Everything else equal.

39.2.4 Using `permute`

Update: So I got my answer from Alex on the mailing list:

> Well, there is the 'permute' function in "lib/simul.l". Does it what you intend?

```
(de permute (Lst)
   (ifn (cdr Lst)
      (cons Lst)
      (mapcan
         '((X)
            (mapcar
               '((Y) (cons X Y))
               (permute (delete X Lst)) ) )
         Lst ) ) )
```

Indeed, and very nice, excellent solution, I wish my mind was lispy enough to come up with these things myself. If you encapsulate the recursive permute call in a println you will get a feeling for how it works.

40

Prolog as a Dating Aid

Henrik Sarvell

hsarvell@gmail.com

Summary. This article describes how `Pilog`, PicoLisp's implementation of `Prolog`, can be used to find out which women in a flirt&dating database have similar tastes like the male candidate.

40.1 A `Prolog` presentation

This Saturday I had a small presentation. It wasn't really well prepared so I'll try and make up for it here instead.

I wanted to demonstrate how Prolog, or in my case Pilog (same thing but different syntax), could be used to solve problems and query object databases. If you've been following my stuff on PicoLisp you won't find much new.

40.2 Set up a Prolog environment

First I began by setting up a simple Prolog environment to demonstrate how Pilog can be used in a setting free of object databases. The goal is to find a compatible woman. This is basically the same thing as the food example.

```
(be actionMovies (Jane))
(be actionMovies (Yoko))
(be durian (Kwan))
(be somTum (Kwan))
(be diving (Anna))
(be Japanese (Yoko))
(be blond (Anna))
(be petite (Yoko))
(be petite (Kwan))
(be sporty (Anna))
(be funny (Yoko))
(be likeBeer (Jane))
(be likeBeer (Anna))
(be speaksThai (Kwan))

(be likes (Tum @F) (actionMovies @F))
(be likes (Tum @F) (Japanese @F))
(be likes (Tum @F) (blond @F))
(be likes (Tum @F) (petite @F))
(be likes (Tum @F) (funny @F))
(be likes (Tum @F) (sporty @F))
(be likes (Tum @F) (likeBeer @F))
(be likes (Tum @F) (speaksThai @F))
(be likes (Tum @F) (durian @F))
(be likes (Tum @F) (somTum @F))
(be likes (Tum @F) (diving @F))

(let L NIL
        (solve '((likes Tum @F))
           (accu 'L @F 1) )
        (flip (by cdr sort L)) )

(println L)
```

The last sequence will output the woman Tum likes the most first and then in descending order. Try to fool around with it, starting with `(? (likes Tum @F))` and then adding the rest step by step and see how the output changes for each step.

40.3 The database

40.3.1 Generate the database

Let's move on to the database stuff. First we need to generate it, you can do that by running the following:

```
(class +Woman +Entity)
(rel id       (+Key +Number))
(rel age      (+Number))
(rel country  (+Ref +Link) NIL (+Country))
(rel hair     (+Ref +Link) NIL (+Color))
(rel smoking  (+Number))
(rel tattoo   (+Number))

(class +Country +Entity)
(rel name (+Key +String))

(class +Color +Entity)
(rel color (+Key +String))

(class +Likes +Entity)
(rel name  (+Key +String))

(class +LikesCon +Entity)
(rel woman  (+Aux +Ref +Link) (likes) NIL (+Woman))
(rel likes  (+Ref +Link) NIL (+Likes))

(pool "bcamp_phuket.db")

(de randEls (Cls Key Amount)
   (make
      (let Lst (collect Key Cls)
         (do Amount
            (let Nth (rand 1 (length Lst))
               (link (get Lst Nth)))))))
```

```
(de setup()
   (mapc '((Col) (new!  '(+Color)   'color Col))
      '("red" "brown" "blond" "black") )
   (mapc '((Like)(new!  '(+Likes)   'name Like))
      '("diving" "skiing" "partying" "pop" "rock" "alternative" "cars"
          "beer" "tennis" "wine" "golf" "geeks" "computers" "som tam"
          "gaeng som" "larb moo" "gaeng aum" ) )
   (mapc '((Con) (new!  '(+Country) 'name Con))
      '("Sweden" "Thailand" "Japan") ) )

(de createWomen ()
   (let N 0
      (do 10000
         (new! '(+Woman)
            'age (rand 18 65)
            'country (car (randEls '+Country 'name 1))
            'hair (car (randEls '+Color 'color 1))
            'smoking (rand 0 1)
            'tattoo (rand 0 1)
            'id (inc 'N)))))

(de createLikes ()
   (for W (collect 'id '+Woman)
      (for L (randEls '+Likes 'name 10)
         (new! '(+LikesCon) 'woman W 'likes L))))

(setup)
(createWomen)
(createLikes)
```

This example uses **new!**, it takes quite a while to generate the database like this. You could try **new** followed by a **commit** when all the objects have been created if you want this to go faster.

40.3.2 Query the database

Let's move on to the final piece, you run this after the above code, don't forget to delete the three last function calls or comment them out first. The end could look like this:

```
#(setup)
#(createWomen)
#(createLikes)

(setq Start (usec))
(setq Women
   (uniq
      (mapcar '((Con)(; Con woman))
         (solve
            (quote
               @C1 "Japan"
               @C2 "Thailand"
               @L1 "beer"
               @L2 "diving"
               @L3 "geeks"
               @Tat 1
               @Smo 0
               (select (@Links)
                  (((name +Likes @L1 name +Likes @L2 name +Likes @L3) (likes +LikesCon)))
                  (or
                     ((same @C1 @Links woman country name))
                     ((same @C2 @Links woman country name)))
                  (same @Tat @Links woman tattoo)
                  (same @Smo @Links woman smoking)))
            @Links ))))

(setq End (format (/ (- (usec) Start) 100000) 1))

(mapc
   '((W)
       (println
          (; W country name)
          (; W smoking)
          (; W tattoo)
          (collect 'woman '+LikesCon W W 'likes 'name))) Women)

(println
   (pack
      "There were "
      (length Women)
      " women matching the query out of 10000. The query took "
      End
      " seconds."))
```

The above will fetch all women from Thailand and Japan who like one of beer, diving or geeks, she also needs to sport a tattoo and not smoke. We work through the connection of women to what they like, when we have them we proceed by extracting the women with `((Con)(; Con woman))` and cutting out duplicates with **uniq**.

We proceed by printing each woman's relevant info and finally we print some statistics on how many hits we got and how long the fetch took.

41

jQuery and PicoLisp

Henrik Sarvell

hsarvell@gmail.com

Summary. This article describes problems arising when trying to make `jQuery` and `PicoLisp` work together, as well as possible solutions and their implementation.

41.1 Problem

The heart of the problem with making `jQuery.post` work with the PicoLisp server is simple once found (thanks to Alex for helping me find it). Apache seems to use the Content-Length header to determine the length of the argument sent by `XMLHttpRequest.send()`, the PicoLisp server doesn't bother with determining the the length. It looks for a `newline` at the end instead.

41.2 Solution

41.2.1 Description

How do we solve this problem considering that `$.ajax` does not append a newline? With my abysmal knowledge of OO programming in Javascript (all that prototyping makes my head hurt) I've opted for the simplest and dirtiest solution. A simple copy paste of `$.post` and `$.ajax` implemented as a separate plugin to enable effortless upgrades of the core, the new stuff can be called with `$.pico.post` and works just like the original.

The only change in the plugin compared with the original is line 2806 in `jquery.js`, it looks like this in the original: **xhr.send(s.data);**, I've changed that to **xhr.send(s.data + ' ');**.

41.2.2 Implementation

My current application renders the HTML like this:

```
(de js (JS)
   (prinl "<script type=\"text/javascript\" src=\""
      (pack *BP "js/" JS) "\"></script>" ) )

(de rss-html Prg
   (html 0 "RSS Reader" *Css NIL
      (js "jquery.js")
      (js "jquery.pico.js")
      (js "rss-reader.js")
      (<div> 'page_margins
         (<div> 'page
            (<div> 'header "header")
            (<div> 'main
               (<div> 'left
                  (<div> 'left_content (leftContent)))
               (<div> 'middle
                  (<div> 'middle_content (run Prg 1)))
               (<div> 'right
                  (<div> 'right_content "right"))
               (<div> 'clear)))
         (<div> 'footer "footer"))))
```

So we include the new plugin in the form of **(js "jquery.pico.js")**, after the base library.

The rss-reader.js file now contains this:

```
$(document).ready(function(){
  $(".articles_link").css('cursor', 'pointer').click(function(){
    $.pico.post("@ajaxTest", {jquerytest: "test"}, function(res){
      $(".middle_content").html(res);
    });
  });
});
```

When one of the links are clicked we call **@ajaxTest:**

```
(de ajaxTest ()
  (httpHead "text/plain; charset=utf-8")
  (ht:Out T
     (ht:Prin (pack "Result: " (get 'jquerytest 'http)))))
```

And the contents of the div with the class attribute of "middle_content" changes to **Result: test**, great.

Source: jquerypico.js

Note that we make use of the httpGate here, the base url in this example is http://localhost, not http://localhost:8080.

Part VI

PicoLisp FAQ

42

Frequently Asked Questions (FAQ)

Alexander Burger

abu@software-lab.de

Monk: "If I have nothing in my mind, what shall I do?"
Joshu: "Throw it out."
Monk: "But if there is nothing, how can I throw it out?"
Joshu: "Well, then carry it out."
(Zen koan)

Summary. These are the official FAQ for PicoLisp.

42.1 Why did you write yet another Lisp?

Because other Lisps are not the way I'd like them to be. They concentrate on efficient compilation, and lost the one-to-one relationship of language and virtual machine of an interpreted system, gave up power and flexibility, and impose unnecessary limitations on the freedom of the programmer. Other reasons are the case-insensitivity and complexity of current Lisp systems.

42.2 Who can use PicoLisp?

PicoLisp is for programmers who want to control their programming environment, at all levels, from the application domain down to the bare metal. Who want use a transparent and simple - yet universal - programming model, and want to know exactly what is going on. This is an aspect influenced by `Forth`.

It does *not* pretend to be easy to learn. There are already plenty of languages that do so. It is not for people who don't care what's under the hood, who just want to get their application running. They are better served with some

standard, "safe" black-box, which may be easier to learn, and which allegedly better protects them from their own mistakes.

42.3 What are the advantages over other Lisp systems?

42.3.1 Simplicity

PicoLisp is easy to understand and adapt. There is no compiler enforcing special rules, and the interpreter is simple and straightforward. There are only three data types: Numbers, symbols and lists ("LISP" means "List-, Integer- and Symbol Processing" after all ;-). The memory footprint is minimal, and the tarball size of the whole system is just a few hundred kilobytes.

42.3.2 A Clear Model

Most other systems define the language, and leave it up to the implementation to follow the specifications. Therefore, language designers try to be as abstract and general as possible, leaving many questions and ambiguities to the users of the language.

PicoLisp does the opposite. Initially, only the single-cell data structure was defined, and then the structure of numbers, symbols and lists as they are composed of these cells. Everything else in the whole system follows from these axioms. This is documented in the chapter about the The PicoLisp Machine in the reference manual.

42.3.3 Orthogonality

There is only one symbolic data type, no distinction (confusion) between symbols, strings, variables, special variables and identifiers.

Most data-manipulation functions operate on the value cells of symbols as well as the CARs of list cells:

```
: (let (N 7  L (7 7 7)) (inc 'N) (inc (cdr L)) (cons N L))
-> (8 7 8 7)
```

There is only a single functional type, no "special forms". As there is no compiler, functions can be used instead of macros. No special "syntax" constructs are needed. This allows a completely orthogonal use of functions. For example, most other Lisps do not allow calls like

```
: (mapcar if '(T NIL T NIL) '(1 2 3 4) '(5 6 7 8))
-> (1 6 3 8)
```

PicoLisp has no such restrictions. It favors the principle of "Least Astonishment".

42.3.4 Object System

The OOP system is very powerful, because it is fully dynamic, yet extremely simple:

- In other systems you have to statically declare "slots". In PicoLisp, classes and objects are completely dynamic, they are created and extended at runtime. "Slots" don't even exist at creation time. They spring into existence purely dynamically. You can add any new property or any new method to any single object, at any time, regardless of its class.
- The multiple inheritance is such that not only classes can have several superclasses, but each individual object can be of more than one class.
- Prefix classes can surgically change the inheritance tree for any class or object. They behave like Mixins in this regard.
- Fine-control of inheritance in methods with `super` and `extra`.

42.3.5 Pragmatism

PicoLisp has many practical features not found in other Lisp dialects. Among them are:

- Auto-quoting of lists when the CAR is a number. Instead of `'(1 2 3)` you can just write `(1 2 3)`. This is possible because a number never makes sense as a function name, and has to be checked at runtime anyway.
- The `quote` function returns all unevaluated arguments, instead of just the first one. This is both faster (`quote` does not have to take the CAR of its argument list) and smaller (a single cell instead of two). For example, 'A expands to `(quote . A)` and `'(A B C)` expands to `(quote A B C)`.
- The symbol `@` is automatically maintained as a local variable, and set implicitly in certain flow- and logic-functions. This makes it often unnecessary to allocate and assign local variables.
- Functional I/O is more convenient than explicitly passing around file descriptors.

- A well-defined ordinal relationship between arbitrary data types facilitates generalized comparing and sorting.
- Uniform handling of `var` locations (i.e. values of symbols and CARs of list cells).
- The universality and usefulness of symbol properties is enforced and extended with implicit and explicit bindings of the symbol `This` in combination with the access functions `=: : ` and `::`.
- A very convenient list-building machinery, using the `link`, `yoke`, `chain` and `made` functions in the `make` environment.
- The syntax of often-used functions is kept non-verbose. For example, instead of `(let ((A 1) (B 2) C 3) ..)` you write `(let (A 1 B 2 C 3) ..)`, or just `(let A 1 ..)` if there is only a single variable.
- The use of the hash (`#`) as a comment character is more adequate today, and allows a clean hash-bang (`#!`) syntax for stand-alone scripts.
- The interpreter is invoked with a simple and flexible syntax, where command line arguments are either files to be interpreted or functions to be directly executed. With that, many tasks can be performed without writing a separate script.
- A sophisticated system of interprocess communication, file locking and synchronization allows multi-user access to database applications.
- A Prolog interpreter is tightly integrated into the language. Prolog clauses can call Lisp expressions and vice versa, and a self-adjusting depth-first search predicate `select` can be used in database queries.

42.3.6 Persistent Symbols

Database objects ("external" symbols) are a primary data type in PicoLisp. They look like normal symbols to the programmer, but are managed in the database (fetched from, and stored to) automatically by the system. Symbol manipulation functions like `set`, `put` or `get`, the garbage collector, and other parts of the interpreter know about them.

42.3.7 Application Server

It is a stand-alone system (it does not depend on external programs like Apache or MySQL) and it provides a "live" user interface on the client side, with an application server session for each connected client. The GUI layout and behavior are described with S-expressions, generated dynamically at runtime, and interact directly with the database structures.

42.3.8 Localization

Internal exclusive and full use of UTF–8 encoding, and self-translating transient symbols (strings), make it easy to write country- and language-independent applications.

42.4 How is the performance compared to other Lisp systems?

Despite the fact that PicoLisp is an interpreted-only system, the performance is quite good. Typical Lisp programs operating on list data structures are executed in (interpreted) PicoLisp at about the same speed as in (compiled) CMUCL, and about two or three times faster than in CLisp or Scheme48. Programs with lots of numeric calculations, however, may be slower on a 32-bit system, due to PicoLisp's somewhat inefficient implementation of numbers. The 64-bit version improved on that.

But in practice, speed was never a problem, even with the first versions of PicoLisp in 1988 on a Mac II with a 12 MHz CPU. And certain things are cleaner and easier to do in plain `C` or `asm` anyway. It is very easy to write `C` functions in PicoLisp, either in the kernel, as shared object libraries, or even inline in the Lisp code.

PicoLisp is very space-effective. Other Lisp systems reserve heap space twice as much as needed, or use rather large internal structures to store cells and symbols. Each cell or minimal symbol in PicoLisp consists of only two pointers. No additional tags are stored, because they are implied in the pointer encodings. No gaps remain in the heap during allocation, as there are only objects of a single size. As a result, consing and garbage collection are very fast, and overall performance benefits from a better cache efficiency. Heap and stack grow automatically, and are limited only by hardware and operating system constraints.

42.5 What means "interpreted"?

It means to directly execute Lisp data as program code. No transformation to another representation of the code (e.g. compilation), and no structural modifications of these data, takes place.

Lisp data are the "real" things, like numbers, symbols and lists, which can be directly handled by the system. They are *not* the textual representation of

these structures (which is outside the Lisp realm and taken care of the `read` ng and `print` ng interfaces).

The following example builds a function and immediately calls it with two arguments:

```
: ((list (list 'X 'Y) (list '* 'X 'Y)) 3 4)
-> 12
```

Note that no time is wasted to build up a lexical environment. Variable bindings take place dynamically during interpretation.

A PicoLisp function is able to inspect or modify itself while it is running (though this is rarely done in application programming). The following function modifies itself by incrementing the '0' in its body:

```
(de incMe ()
   (do 8
      (printsp 0)
      (inc (cdadr (cdadr incMe))) ) )

: (incMe)
0 1 2 3 4 5 6 7 -> 8
: (incMe)
8 9 10 11 12 13 14 15 -> 16
```

Only an interpreted Lisp can fully support such "Equivalence of Code and Data". If executable pieces of data are used frequently, like in PicoLisp's dynamically generated GUI, a fast interpreter is preferable over any compiler.

42.6 Is there (or will be in the future) a compiler available?

No. That would contradict the idea of PicoLisp's simple virtual machine structure. A compiler transforms it to another (physical) machine, with the result that many assumptions about the machine's behavior won't hold any more. Besides that, PicoLisp primitive functions evaluate their arguments independently and are not suited for being called from compiled code. Finally, the gain in execution speed would probably not be worth the effort. Typical PicoLisp applications often use single-pass code which is loaded, executed and thrown away; a process that would be considerably slowed down by compilation.

42.7 Is it portable?

Yes and No. Though we wrote and tested PicoLisp originally only on Linux, it now also runs on FreeBSD, Mac OS X (Darwin), Cygwin/Win32, and probably other POSIX systems. The first versions were even fully portable between DOS, SCO-Unix and Macintosh systems. But today we have Linux. Linux itself is very portable, and you can get access to a Linux system almost everywhere. So why bother?

The GUI is completely platform independent (Browser), and in the times of Internet an application server does not really need to be portable.

42.8 Is PicoLisp a web server?

Not really, but it evolved a great deal into that direction.

Historically it was the other way round: We had a plain X11 GUI for our applications, and needed something platform independent. The solution was obvious: Browsers are installed virtually everywhere. So we developed a protocol which persuades a browser to function as a GUI front-end to our applications. This is much simpler than to develop a full-blown web server.

42.9 I cannot find the LAMBDA keyword in PicoLisp

Because it isn't there. The reason is that it is redundant; it is equivalent to the `quote` function in any aspect, because there's no distinction between code and data in PicoLisp, and `quote` returns the whole (unevaluated) argument list. If you insist on it, you can define your own `lambda`:

```
: (def 'lambda quote)
-> lambda
: ((lambda (X Y) (+ X Y)) 3 4)
-> 7
: (mapcar (lambda (X) (+ 1 X)) '(1 2 3 4 5))
-> (2 3 4 5 6)
```

42.10 Why do you use dynamic variable binding?

Dynamic binding is very powerful, because there is only one single, dynamically changing environment active all the time. This makes it possible (e.g. for

program snippets, interspersed with application data and/or passed over the network) to access the whole application context, freely, yet in a dynamically controlled manner. And (shallow) dynamic binding is the fastest method for a Lisp interpreter.

Lexical binding is more limited by definition, because each environment is deliberately restricted to the visible (textual) static scope within its establishing form. Therefore, most Lisps with lexical binding introduce "special variables" to support dynamic binding as well, and constructs like `labels` to extend the scope of variables beyond a single function.

In PicoLisp, function definitions are normal symbol values. They can be dynamically rebound like other variables. As a useful real-world example, take this little gem:

```
(de recur recurse
   (run (cdr recurse)) )
```

It implements anonymous recursion, by defining `recur` statically and `recurse` dynamically. Usually it is very cumbersome to think up a name for a function (like the following one) which is used only in a single place. But with `recur` and `recurse` you can simply write:

```
: (mapcar
   '((N)
      (recur (N)
         (if (=0 N)
            1
            (* N (recurse (- N 1))) ) ) )
   (1 2 3 4 5 6 7 8) )
-> (1 2 6 24 120 720 5040 40320)
```

Needless to say, the call to `recurse` does not have to reside in the same function as the corresponding `recur`. Can you implement anonymous recursion so elegantly with lexical binding?

42.11 Are there no problems caused by dynamic binding?

You mean the *funarg* problem, or problems that arise when a variable might be bound to *itself*? For that reason we have a convention in PicoLisp to use transient symbols (instead of internal symbols)

1. for all parameters and locals, when functional arguments or executable lists are passed through the current dynamic bindings
2. for a parameter or local, when that symbol might possibly be (directly or indirectly) bound to itself, and the bound symbol's value is accessed in the dynamic context

This is a form of lexical *scoping* - though we still have dynamic *binding* - of symbols, similar to the `static` keyword in `C`.

In fact, these problems are a real threat, and may lead to mysterious bugs (other Lisps have similar problems, e.g. with symbol capture in macros). They can be avoided, however, when the above conventions are observed. As an example, consider a function which doubles the value in a variable:

```
(de double (Var)
   (set Var (* 2 (val Var))) )
```

This works fine, as long as we call it as `(double 'X)`, but will break if we call it as `(double 'Var)`. Therefore, the correct implementation of `double` should be:

```
(de double ("Var")
   (set "Var" (* 2 (val "Var"))) )
```

If `double` is defined that way in a separate source file, and/or isolated via the `====` function, then the symbol `Var` is locked into a private lexical context and cannot conflict with other symbols.

Admittedly, there are two disadvantages with this solution:

1. The rules for when to use transient symbols are a bit complicated. Though it is safe to use them even when not necessary, it will take more space then and be more difficult to debug.
2. The string-like syntax of transient symbols as variables may look strange to alumni of other languages.

Fortunately, these pitfalls do not occur so very often, and seem more likely in utilities than in production code, so that they can be easily encapsulated.

42.12 But with dynamic binding I cannot implement closures!

This is not true. Closures are a matter of scope, not of binding.

For a closure it is necessary to build and maintain a separate environment. In a system with lexical bindings, this has to be done at *each* function call, and for compiled code it is the most efficient strategy anyway, because it is done once by the compiler, and can then be accessed as stack frames at runtime.

For an interpreter, however, this is quite an overhead. So it should not be done automatically at each and every function invocation, but only if needed.

You have several options in PicoLisp. For simple cases, you can take advantage of the static scope of transient symbols. For the general case, PicoLisp has built-in functions like `bind` or `job`, which dynamically manage statically scoped environments.

Environments are first-class objects in PicoLisp, more flexible than hard-coded closures, because they can be created and manipulated independently from the code.

As an example, consider a currying function:

```
(de curry Args
   (list (car Args)
      (list 'list
         (lit (cadr Args))
         (list 'cons ''job
            (list 'cons
               (list 'lit (list 'env (lit (car Args))))
               (lit (cddr Args)) ) ) ) ) )
```

When called, it returns a function-building function which may be applied to some argument:

```
: ((curry (X) (N) (* X N)) 3)
-> ((N) (job '((X . 3)) (* X N)))
```

or used as:

```
: (((curry (X) (N) (* X N)) 3) 4)
-> 12
```

In other cases, you are free to choose a shorter and faster solution. If (as in the example above) the curried argument is known to be immutable:

```
(de curry Args
   (list
      (cadr Args)
      (list 'fill
         (lit (cons (car Args) (cddr Args)))
         (lit (cadr Args)) ) ) )
```

Then the function built above will just be:

```
: ((curry (X) (N) (* X N)) 3)
-> ((X) (* X 3))
```

In that case, the "environment build-up" is reduced by a simple (lexical) constant substitution with zero runtime overhead.

Note that the actual `curry` function is simpler and more pragmatic. It combines both strategies (to use `job`, or to substitute), deciding at runtime what kind of function to build.

42.13 Do you have macros?

Yes, there is a macro mechanism in PicoLisp, to build and immediately execute a list of expressions. But it is seldom used. Macros are a kludge. Most things where you need macros in other Lisps are directly expressible as functions in PicoLisp, which (as opposed to macros) can be applied, passed around, and debugged.

42.14 Why are there no strings?

Because PicoLisp has something better: Transient symbols. They look and behave like strings in any respect, but are nevertheless true symbols, with a value cell and a property list.

This leads to interesting opportunities. The value cell, for example, can point to other data that represent the string's translation. This is used extensively for localization. When a program calls

```
(prinl "Good morning!")
```

then changing the value of the symbol `''Good morning!''` to its translation will change the program's output at runtime.

Transient symbols are also quite memory-conservative. As they are stored in normal heap cells, no additional overhead for memory management is induced. The cell holds the symbol's value in its CDR, and the tail in its CAR. If the string is not longer than 7 bytes, it fits (on the 64-bit version) completely into the tail, and a single cell suffices. Up to 15 bytes take up two cells, 23 bytes three etc., so that long strings are not very efficient (needing twice the memory on the average), but this disadvantage is made up by simplicity and uniformity. And lots of extremely long strings are not the common case, as they are split up anyway during processing, and stored as plain byte sequences in external files and databases.

Because transient symbols are temporarily interned (while `load` ng the current source file), they are shared within the same source and occupy that space only once, even if they occur multiple times within the same file.

42.15 What about arrays?

PicoLisp has no array or vector data type. Instead, lists must be used for any type of sequentially arranged data.

We believe that arrays are usually overrated. Textbook wisdom tells that they have a constant access time O(1) when the index is known. Many other operations like splits or insertions are rather expensive. Access with a known (numeric) index is not really typical for Lisp, and even then the advantage of an array is significant only if it is relatively long. Holding lots of data in long arrays, however, smells quite like a program design error, and we suspect that often more structured representations like trees or interconnected objects would be better.

In practice, most arrays are rather short, or the program can be designed in such a way that long arrays (or at least an indexed access) are avoided.

Using lists, on the other hand, has advantages. We have so many concerted functions that uniformly operate on lists. There is no separate data type that has to be handled by the interpreter, garbage collector, I/O, database and so on. Lists can be made circular. And lists don't cause memory fragmentation.

42.16 How to do floating point arithmetics?

PicoLisp does not support real floating point numbers. You can do all kinds of floating point calculations by calling existing library functions via `native`, inline-C code, and/or by loading the "@lib/math.l" library.

But PicoLisp has something even (arguably) better: Scaled fixpoint numbers, with unlimited precision.

The reasons for this design decision are manifold. Floating point numbers smack of imperfection, they don't give "exact" results, have limited precision and range, and require an extra data type. It is hard to understand what really goes on (How many digits of precision do we have today? Are perhaps 10-byte floats used for intermediate results? How does rounding behave?).

For fixpoint support, the system must handle just integer arithmetics, I/O and string conversions. The rest is under programmer's control and responsibility (the essence of PicoLisp).

Carefully scaled fixpoint calculations can do anything floating points can do.

42.17 What happens when I locally bind a symbol which has a function definition?

That's not a good idea. The next time that function gets executed within the dynamic context the system may crash. Therefore we have a convention to use an upper case first letter for locally bound symbols:

```
(de findCar (Car List)
   (when (member Car (cdr List))
      (list Car (car List)) ) )
```

;-)

42.18 Would it make sense to build PicoLisp in hardware?

At least it should be interesting. It would be a machine executing list (tree) structures instead of linear instruction sequences. "Instruction prefetch" would look down the CAR- and CDR-chains, and perhaps need only a single cache for both data and instructions.

Primitive functions like `set`, `val`, `if` and `while`, which are written in `C` or assembly language now, would be implemented in microcode. Plus a few I/O functions for hardware access. `EVAL` itself would be a microcode subroutine.

Only a single heap and a single stack is needed. They grow towards each other, and cause garbage collection if they get too close. Heap compaction is trivial due to the single cell size.

There would be no assembly-language. The lowest level (above the hardware and microcode levels) are s-expressions: The machine language is *Lisp*.

42.19 I get a segfault if I ...

PicoLisp is a pragmatic language. It doesn't check at runtime for all possible error conditions which won't occur during normal usage. Such errors are usually detected quickly at the first test run, and checking for them after that would just produce runtime overhead.

Catching the segfault signals is also not a good idea, because the Lisp heap is most probably be damanged afterwards.

It is recommended, though, to inspect the code periodically with `lint`.

42.20 Where can I ask questions?

The best place is the PicoLisp Mailing List (see also The Mail Archive), or the IRC `#picolisp` channel on FreeNode.net.

43

Some technical questions and answers

Alexander Burger

abu@software-lab.de

Summary. These are some technical questions about PicoLisp with answers, additional to the official FAQ.

43.1 Can there be more than one copy of the symbol T?

Question

NIL is a special symbol which exists exactly once in the whole system. Can there be more than one copy of the symbol **T**?

Answer

In this sense, *any* internal symbol is unique, and exists exactly once in the system. And any internal symbol (also 'NIL') could exist a second time, but it cannot be interned at the same time (and would thus be a transient symbol).

"interned" means no more or less that an entry in the "internal" symbol table points to that symbol.

'NIL' is special, however, because even if you would succeed to intern a new (transient) symbol "NIL" (it isn't possible, as the existing 'NIL' cannot be uninterned), it would not have the speciality of the old 'NIL', e.g. being returned from boolean functions, because these functions return a hard-compiled pointer to the old 'NIL', and conditionals check for this pointer.

Take a normal symbol, say "abc". You can create several transient symbols "abc" in the system, for example by loading them from different source files, or separating the input with (====), by 'pack'ing them and so on.

Now you could 'intern' one of those "abc" symbols. It will appear as 'abc'. The reader will always return that interned symbol when it sees 'abc'. A call to (intern "abc") also would return that already-interned symbol 'abc'. To change another one of the above "abc" symbols to 'abc', you'll first have to unintern (with 'zap') the existing 'abc'.

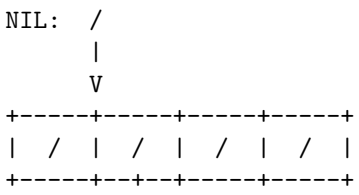
43.2 Why is the symbol T not protected like NIL?

Question

(Related to the one above) If one tries to put a property on the **NIL** symbol, one gets the message "NIL – Protected symbol". Why is the symbol **T** not protected likewise?

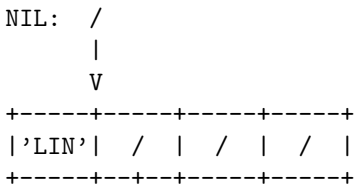
Answer

Perhaps is this error message not necessary, at least not in the 64-bit version. For a background, see the structure of 'NIL' in "doc/structures":



'NIL' is the only symbol that has a double nature: It is a symbol *and* a cons pair. It doesn't even have a name in the 32-bit version, the reader and printer simply know about it, and read and print 'N', 'I', and 'L' by themselves.

In the 64-bit version the situation is slightly different:



Here NIL *does* have a name. This field where you see the letters 'N', 'I' and 'L' here, is called the symbol's tail. Besides the name, it can also hold a property list.

Technically, there is perhaps no problem when storing also properties in that tail of 'NIL', and it might work (at least on the 64-bit version) if we removed the above error message.

However, I think it is wise to prohibit properties in 'NIL', as "NIL" also means "nothing", and storing properties in "nothing" sounds a bit sick.

Any opinions?

43.3 Why does the REPL exit when NIL is typed?

Question

If you type just "NIL" or "()" on the command line in the REPL, then the REPL exits. Why is that?

Answer

The REPL is basically

```
(while (read)
   (eval @) )
```

'read' returns 'NIL' upon end of file, but also when it indeed *reads* 'NIL'.

The same happens btw if you write the atom 'NIL' somewhere in a 'load'ed file. This is a convenient way to "comment" the rest of the file.

Even better is *conditional commenting* of the rest of a file, by using a back-quote read macro. When '(condition) is read, the rest of the file will be ignored if (condition) evaluates to 'NIL'. There are examples for that in the sources, e.g. at the end of "lib/form.l", where '*Dbg causes the rest of the file to be loaded only if in debugging mode.

Note: As of picoLisp-3.0.6, the interpreter does no longer exit when the top level REPL reads NIL. This is a bit inconsistent, but more what seems to be expected by most people.

43.4 PicoLisp indicated that 'be' was undefined - why?

Question

At this URL (http://rosettacode.org/wiki/Pascal%27s_triangle/Puzzle#PicoLisp[1]) there is some PicoLisp code for solving Pascal's triangle. I tried it out on my machine and PicoLisp indicated that 'be' was undefined. Where would I find it? I'm running version 3.0.4 on a Windows 7 Home Premium 64-bit system.

Answer

This is most probably because you didn't load the full system, but just the 'picolisp' binary perhaps.

Normally, PicoLisp is either started locally with the 'p' or 'dbg' scripts:

```
$ ./p +
:
```

or globally (e.g. when installed via some package) with the 'pil' command:

```
$ pil +
:
```

In both cases, the full system is loaded. The trailing '+' indicates debug mode.

```
: (pp 'be)
(de be CL
   (with (car CL)
      (if (== *Rule This)
         (=: T (conc (: T) (cons (cdr CL))))
         (=: T (cons (cdr CL)))
         (setq *Rule This) )
      This ) )
-> be
```

[1] http://rosettacode.org/wiki/Pascal%27s_triangle/Puzzle#PicoLisp

Part VII

PicoLisp 64-bit Version

44

README 64-bit

Alexander Burger

abu@software-lab.de

Summary. This is the README file from the 64-bit PicoLisp distribution.

44.1 64-bit PicoLisp

The 64-bit version of PicoLisp is a complete rewrite of the 32-bit version.

While the 32-bit version was written in C, the 64-bit version is implemented in a generic assembler, which in turn is written in PicoLisp. In most respects, the two versions are compatible (see "Differences" below).

44.1.1 Building the Kernel

No C-compiler is needed to build the interpreter kernel, only a 64-bit version of the GNU assembler for the target architecture.

The kernel sources are the "*.l" files in the "src64/" directory. The PicoLisp assembler parses them and generates a few "*.s" files, which the GNU assembler accepts to build the executable binary file. See the details for bootstrapping the "*.s" files in INSTALL.

The generic assembler is in "src64/lib/asm.l". It is driven by the script "src64/mkAsm" which is called by "src64/Makefile".

The CPU registers and instruction set of the PicoLisp processor are described in "doc64/asm", and the internal data structures of the PicoLisp machine in "doc64/structures".

Currently, x86-64/Linux, x86-64/SunOS and ppc64/Linux are supported. The platform dependent files are in the "src64/arch/" for the target architecture, and in "src64/sys/" for the target operating system.

44.1.2 Reasons for the Use of Assembly Language

Contrary to the common expectation: Runtime execution speed was not a primary design decision factor. In general, pure code efficiency has not much influence on the overall execution speed of an application program, as memory bandwidth (and later I/O bandwidth) is the main bottleneck.

The reasons to choose assembly language (instead of C) were, in decreasing order of importance:

1. Stack manipulations

   Alignment to cell boundaries: To be able to directly express the desired stack data structures (see "doc64/structures", e.g. "Apply frame"), a better control over the stack (as compared to C) was required.

   Indefinite pushs and pops: A Lisp interpreter operates on list structures of unknown length all the time. The C version always required two passes, the first to determine the length of the list to allocate the necessary stack structures, and then the second to do the actual work. An assembly version can simply push as many items as are encountered, and clean up the stack with pop's and stack pointer arithmetics.

2. Alignments and memory layout control

   Similar to the stack structures, there are also heap data structures that can be directly expressed in assembly declarations (built at assembly time), while a C implementation has to defer that to runtime.

   Built-in functions (SUBRs) need to be aligned to to a multiple of 16+2, reflecting the data type tag requirements, and thus allow direct jumps to the SUBR code without further pointer arithmetic and masking, as is necessary in the C version.

3. Multi-precision arithmetics (Carry-Flag)

   The bignum functions demand an extensive use of CPU flags. Overflow and carry/borrow have to emulated in C with awkward comparisons of signed numbers.

4. Register allocation

   A manual assembly implementation can probably handle register allocation more flexibly, with minimal context saves and reduced stack space, and multiple values can be returned from functions in registers. As mentioned above, this has no measurable effect on execution speed, but the binary's overall size is significantly reduced.

5. Return status register flags from functions

Functions can return condition codes directly. The callee does not need to re-check returned values. Again, this has only a negligible impact on performance.

6. Multiple function entry points

   Some things can be handled more flexibly, and existing code may be easier to re-use. This is on the same level as wild jumps within functions ('goto's), but acceptable in the context of an often-used but rarely modified program like a Lisp kernel.

It would indeed be feasible to write only certain parts of the system in assembly, and the rest in C. But this would be rather unsatisfactory. And it gives a nice feeling to be independent of a heavy-weight C compiler.

44.1.3 Differences to the 32-bit Version

Except for the following seven cases, the 64-bit version should be upward compatible to the 32-bit version.

1. Internal format and printed representation of external symbols

   This is probably the most significant change. External (i.e. database) symbols are coded more efficiently internally (occupying only a single cell), and have a slightly different printed representation. Existing databases need to be converted.

2. Short numbers are pointer-equal

   As there is now an internal "short number" type, an expression like

   ```
   (== 64 64)
   ```

   will evaluate to 'T' on a 64-bit system, but to 'NIL' on a 32-bit system.

3. Bit manipulation functions may differ for negative arguments

   Numbers are represented internally in a different format. Bit manipulations are not really defined for negative numbers, but (-15 -6) will give -6 on 32 bits, and 6 on 64 bits.

4. 'do' takes only a 'cnt' argument (not a bignum)

   For the sake of simplicity, a short number (60 bits) is considered to be enough for counted loops.

5. Calling native functions is different.

Direct calls using the 'lib:fun' notation is still possible (see the 'ext' and 'ht' libraries), but the corresponding functions must of course be coded in assembly and not in C. To call C functions, the new 'native' function should be used, which can interface to native C functions directly, without the need of glue code to convert arguments and return values.

6. New features were added, like coroutines or namespaces.
7. Bugs (in the implementation, or in this list ;-)

45

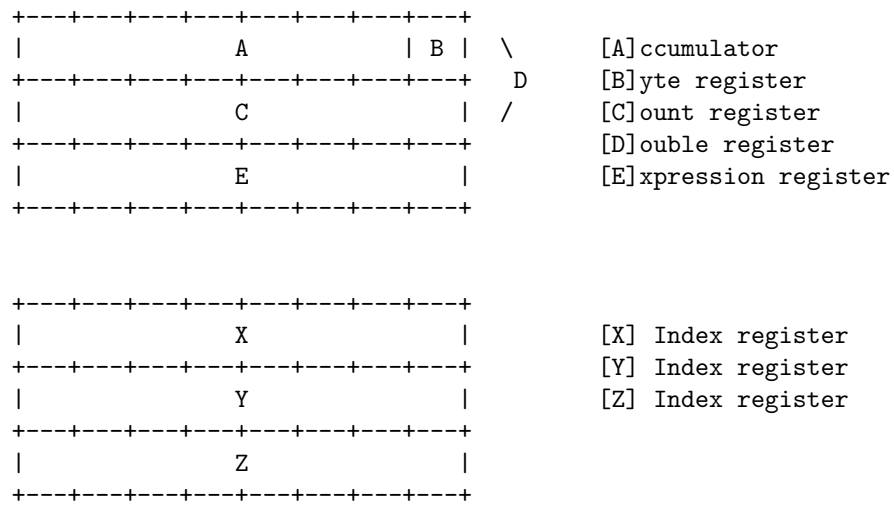
Generic VM/Assembler

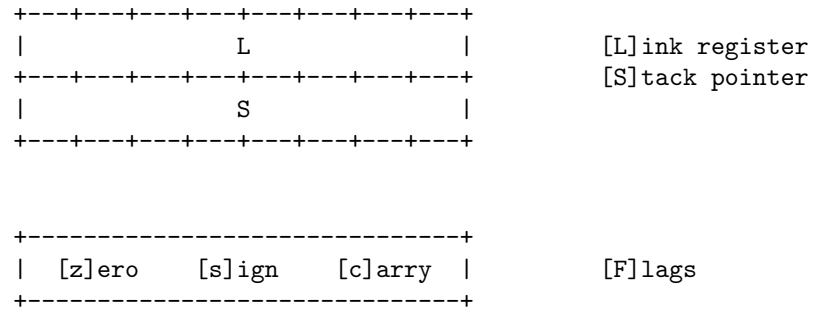
Alexander Burger

abu@software-lab.de

Summary. This is a detailled description of the 64-bit Generic VM/Assembler.

45.1 CPU Registers





Source Addressing Modes:

```
ld A 1234            # Immediate
ld A "(a+b-c)"
ld A R               # Register
ld A Global          # Direct
ld A (R)             # Indexed
ld A (R 8)           # Indexed with offset
ld A (R OFFS)
ld A (R Global)
ld A (Global)        # Indirect
ld A (Global OFFS)   # Indirect with offset
ld A ((R))           # Indexed indirect
ld A ((R 8))         # Indexed with offset indirect
ld A ((R 8) OFFS)
ld A ((R Global) OFFS)
ld A ((R OFFS) Global)
...
```

Destination Addressing Modes:

```
ld R A               # Register
ld (R) A             # Indexed
ld (R 8) A           # Indexed with offset
ld (R OFFS) A
ld (R Global) A
```

```
ld (Global) A        # Indirect
ld (Global OFFS) A   # Indirect with offset
ld ((R)) A           # Indexed indirect
ld ((R 8)) A         # Indexed with offset indirect
ld ((R 8) OFFS) A
ld ((R Global) OFFS) A
ld ((R OFFS) Global) A
...
```

Destination Addressing Modes: Target Addressing Modes:

```
jmp 1234             # Absolute
jmp Label
jmp (R)              # Indexed
jmp (R T)            # Indexed SUBR
jmp (Global)         # Indirect
```

45.2 Instruction Set

No Operation:

```
nop              # No operation
```

Move Instructions:

```
ld dst src       # Load 'dst' from 'src' [---]
ld2 src          # Load 'A' from two bytes 'src' (unsigned)
ld4 src          # Load 'A' from four bytes 'src' (unsigned)
ldc reg src      # Load if Carry 'reg' from 'src'
ldnc reg src     # Load if not Carry 'reg' from 'src'
ldz reg src      # Load if Zero 'reg' from 'src'
ldnz reg src     # Load if not Zero 'reg' from 'src'
lea dst src      # Load 'dst' with effective address of 'src' [---]
```

```
st2 dst           # Store two bytes from 'A' into 'dst'
st4 dst           # Store four bytes from 'A' into 'dst'
xchg dst dst      # Exchange 'dst's
movn dst src cnt  # Move 'cnt' bytes from 'src' to 'dst' (non-overlapping)
mset dst cnt      # Set 'cnt' bytes of memory to B
movm dst src end  # Move memory 'src'..'end' to 'dst' (aligned)
save src end dst  # Save 'src'..'end' to 'dst' (non-overlapping)
load dst end src  # Load 'dst'..'end' from 'src' (non-overlapping)
```

Arithmetics:

```
  add dst src       # Add 'src' to 'dst' [zsc]
  addc dst src      # Add 'src' to 'dst' with Carry [zsc]
  sub dst src       # Subtract 'src' from 'dst' [zsc]
  subc dst src      # Subtract 'src' from 'dst' with Carry [zsc]

  inc dst           # Increment 'dst' [zs.]
  dec dst           # Increment 'dst' [zs.]
  not dst           # One's complement negation of 'dst'
  neg dst           # Two's complement negation of 'dst'

  and dst src       # Bitwise AND 'dst' with 'src'
  or dst src        # Bitwise OR 'dst' with 'src'
  xor dst src       # Bitwise XOR 'dst' with 'src'
  off dst src       # Clear 'src' bits in 'dst'
  test dst src      # Bit-test 'dst' with 'src' [z._]


shl dst src       # Shift 'dst' left into Carry by 'src' bits
shr dst src       # Shift 'dst' right into Carry by 'src' bits
rol dst src       # Rotate 'dst' left by 'src' bits
ror dst src       # Rotate 'dst' right by 'src' bits
rcl dst src       # Rotate 'dst' with Carry left by 'src' bits
rcr dst src       # Rotate 'dst' with Carry right by 'src' bits

mul src           # Multiplication of 'A' and 'src' into 'D' [...]
div src           # Division of 'D' by 'src' into 'A', 'C' [...]

zxt               # Zero-extend 'B' to 'A'

setz              # Set Zero flag [z__]
clrz              # Clear Zero flag [z..]
setc              # Set Carry flag [--c]
clrc              # Clear Carry flag [--c]
```

Comparisons:

```
cmp dst src       # Compare 'dst' with 'src' [z.c]
cmp4 src          # Compare four bytes in 'A' with 'src'
cmpn dst src cnt  # Compare 'cnt' bytes 'dst' with 'src'
slen dst src      # Set 'dst' to the string length of 'src'
memb src cnt      # Find B in 'cnt' bytes of memory [z..]
null src          # Compare 'src' with 0 [zs_]
nul4              # Compare four bytes in 'A' with 0 [zs_]
```

Byte addressing:

```
set dst src       # Set 'dst' byte to 'src'
nul src           # Compare byte 'src' with 0 [zs_]
```

Types:

```
cnt src           # Non-'z' if small number
big src           # Non-'z' if bignum
num src           # Non-'z' if number
sym src           # Non-'z' if symbol
atom src          # Non-'z' if atom
```

Flow Control:

```
jmp adr           # Jump to 'adr'
jz adr            # Jump to 'adr' if Zero
jnz adr           # Jump to 'adr' if not Zero
js adr            # Jump to 'adr' if Sign
jns adr           # Jump to 'adr' if not Sign
jc adr            # Jump to 'adr' if Carry
jnc adr           # Jump to 'adr' if not Carry
call adr          # Call 'adr'
cc adr(src ..)    # C-Call to 'adr' with 'src' arguments
cc adr reg        # C-Call to 'adr' with top of stacked args in 'reg'
ldd               # Load double value pointed to by 'C'
ldf               # Load float value pointed to by 'C'
fixnum            # Convert double with scale 'E' to fixnum in 'E'
float             # Convert fixnum with scale 'A' pointed to by 'X'
std               # Store double value at address 'Z'
stf               # Store float value at address 'Z'
```

```
ret                # Return
begin              # Called from foreign function
return             # Return to foreign function
```

Stack Manipulations:

```
push src           # Push 'src' [---]
pop dst            # Pop 'dst' [---]
link               # Setup frame
tuck src           # Extend frame
drop               # Drop frame [---]
```

Evaluation:

```
eval               # Evaluate expression in 'E'
eval+              # Evaluate expression in partial stack frame
eval/ret           # Evaluate expression and return
exec reg           # Execute lists in 'reg', ignore results
prog reg           # Evaluate expressions in 'reg', return last result
```

System:

```
initData           # Init runtime system
initCode
initMain
```

45.3 Naming Conventions

Lisp level functions, which would be all of the form `doXyzE_E`, are written as `doXyz` for brevity.

46

Internal Structures 64-bit Version

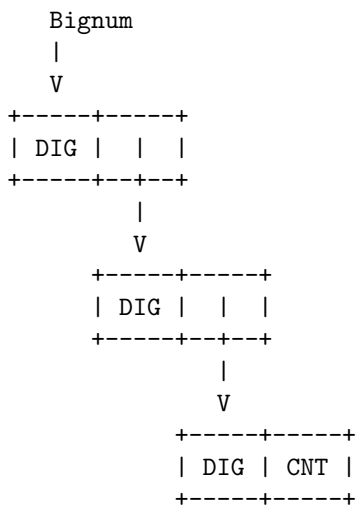
Alexander Burger

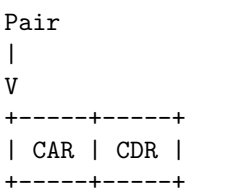
abu@software-lab.de

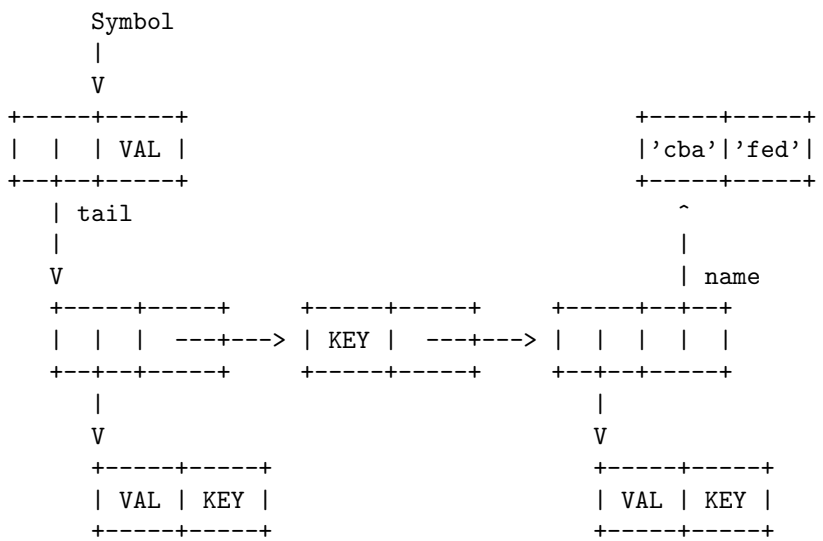
Summary. This article describes the internal structures of the 64-bit PicoLisp Version.

46.1 Primary Data Types

```
cnt   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxS010
big   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxS100
sym   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1000
pair  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0000
```

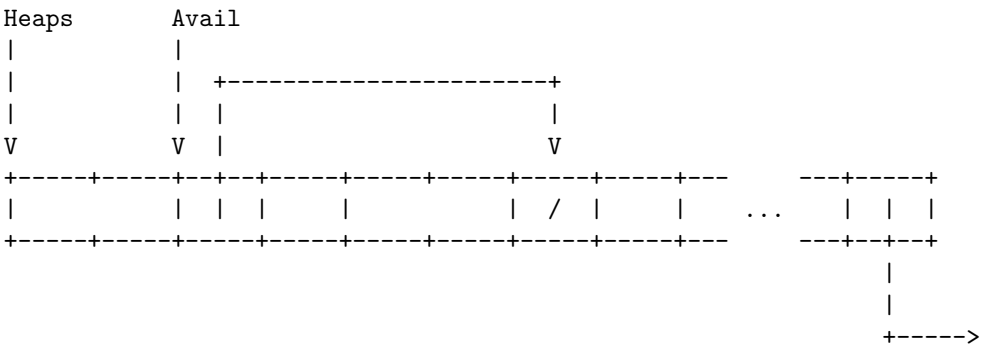

Bignum
DIG
DIG
DIG
CNT


Pair
CAR
CDR

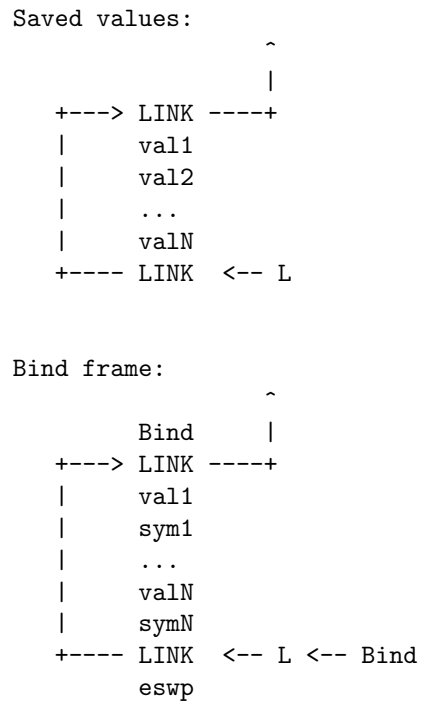

Symbol
VAL
'cba'
'fed'
tail
name
KEY
KEY
VAL
KEY
VAL
KEY

```
   NIL:  /
         |
         V
   +-----+-----+-----+-----+
   |'LIN'|  /  |  /  |  /  |
   +-----+--+--+-----+-----+

Symbol tail
   Internal/Transient
      0010 Short name
      0100 Long name
      0000 Properties

   External
      1010 Short name
      1000 Properties

Name final short
   Internals, Transients
      0000.xxxxxxx.xxxxxxx.xxxxxxx.xxxxxxx.xxxxxxx.xxxxxxx.xxxxxxx0010
        60      52      44      36      28      20      12       4


   Externals
      42 bit Object (4 Tera objects)
      16 bit File (64 K files)
       2 bit Status
         Loaded    01........
         Dirty     10........
         Deleted   11........

      1+2 Bytes: 1 file, 64K objects                  {177777}
      1+3 Bytes: 16 files, 1M objects                {03777777}
      1+4 Bytes: 256 files, 16M objects             {0077777777}
      1+5 Bytes: 256 files, 4G objects           {0037777777777}
      1+6 Bytes: 65536 files, 4G objects       {000037777777777}
      1+8 Bytes: 65536 files, 4T objects  {0000777777777777777}
      (2 + 10 + 8 + 12 + 8 + 20)
      xx.xxxxxxxx.xxxxxxx.xxxxxxxxxx.xxxxxxx.xxxxxxxxxxxxxxxxxxE010
        obj       file    obj         file    obj
                  ^6      ^5      ^4      ^3      ^2
```

46.2 Heap



46.3 Stack

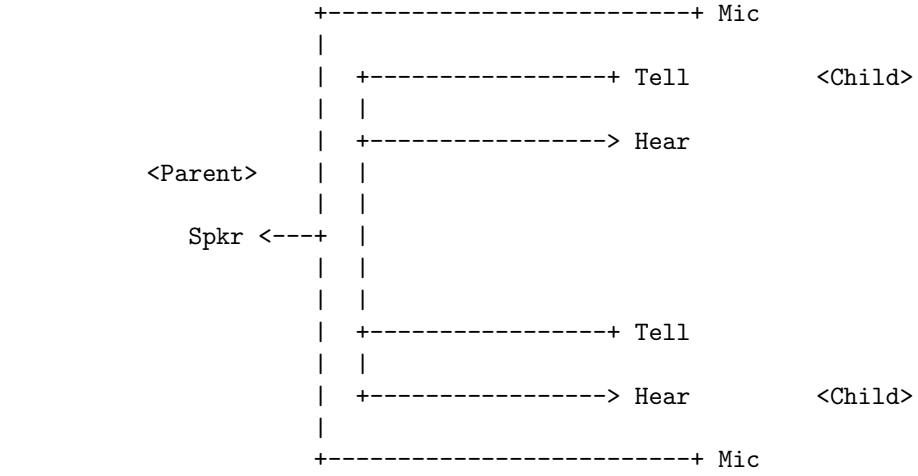


```
VarArgs frame:
                    ^
          Bind      |
   +---> LINK -----+
   |     val1
   |     sym1
   |     ...
   |     valN
   |     symN
   +---- LINK <---+  <-- Bind
          eswp    |
          Next    |
          Args    |
   +---> LINK ----+  <-- Next
   |     arg1
   |     ...
   |     argN        <-- Args
   +---- LINK  <-- L


Apply args:
                    ^
                    |
   +---> LINK -----+
   |     ...
   |     fun         <-- Y
   |     arg1
   |     ...
   |     argN        <-- Z
   |     ...
   +---- LINK <-- L


Apply frame:
                    ^
          Apply     |
   +---> LINK -----+
   |     ...
   |     valN <-+       (gc)
   |     zero   |
   |     NIL    |       (gc)
   |     carN --+ <-+
   |     ...        |
   |     val1 <-+   |   (gc)
   |     zero   |   |
   |     cdr1 --|---+   (gc)
   | +-> car1 --+
   | +-- cdr             (gc)
   |     fun         <-- exe
   +---- LINK  <-- L <-- Apply
```

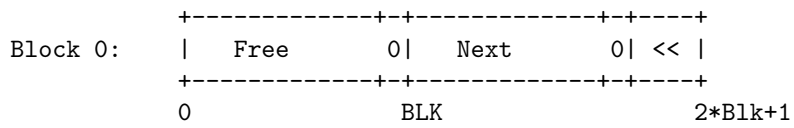
```
Catch frame:
                  ^
         X        |
         Y        |
         Z        |
         L        |
   <III> [env]    |
   <II>  fin      |
   <I>   tag      |
         LINK ----+  <-- Catch


I/O frame:
                  ^
   <III> put/get  |
   <II>  pid      |
   <I>   fd       |
         LINK ----+  <-- inFrames, outFrames, errFrames, ctlFrames


Coroutine frame:
                  ^
         X        |
         Y        |
         Z        |
         L        |
   <III> [env]    |
   <II>  seg      |
   <I>   lim      |
         LINK ----+  <-- co7


Stack segment:
   <-I>  tag      # Tag
   <-II> stk      # Stack pointer --+
         [env]    # Environment     |
         Stack ...                  |
         X                          |
         Y                          |
         Z                          |
         L  <-----------------------+
```

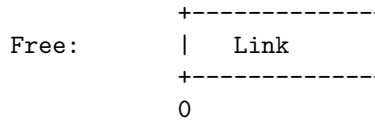
46.4 Memory

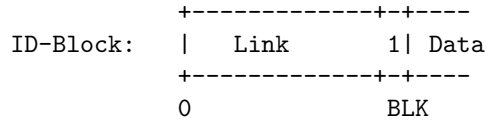
```
inFile:
   -->   fd       # File descriptor
   <I>   ix       # Read index
   <II>  cnt      # Buffer count
   <III> next     # Next character
   <IV>  line     # Line number
   <V>   src      # Source line number
   <VI>  name     # Filename
   <VII> buf      # Buffer [BUFSIZ]

outFile:
   -->   fd       # File descriptor
   <I>   ix       # Read index
   <II>  tty      # TTY flag
   <III> buf      # Buffer [BUFSIZ]


child:
   -->   pid      # Process ID
   <I>   hear     # Pipe read end
   <II>  tell     # Pipe write end
   <III> ofs      # Buffer offset
   <IV>  cnt      # Buffer count
   <V>   buf      # Buffer pointer
```

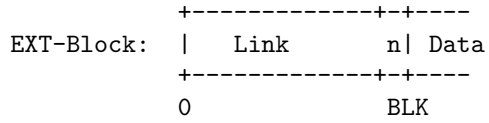


46.5 Database File










```
dbFile:  # Size VIII (64 bytes)
   -->   fd       # File descriptor
   <I>   db       # File number
   <II>  sh       # Block shift
   <III> siz      # Block size (64 << sh)
   <IV>  flgs     # Flags: Lock(0), Dirty(1)
   <V>   marks    # Mark vector size
   <VI>  mark     # Mark bit vector
   <VII> fluse    # Free list use count
```

46.6 Assumptions

- 8 bit per byte
- 64 bit per word
- Stack grows downwards, aligned to 64 bit
- Memory access legal also at 4-byte boundaries

Part VIII

Ersatz PicoLisp

47

README Ersatz-PicoLisp

Alexander Burger

abu@software-lab.de

Summary. This is the README file from the Ersatz PicoLisp distribution.

47.1 Ersatz PicoLisp

Ersatz PicoLisp is a version of PicoLisp completely written in Java. It requires a 1.6 Java Runtime Environment.

It should be the last resort when there is no other way to run a "real" PicoLisp. Also, it may be used to bootstrap the 64-bit version, which requires a running PicoLisp to build from the sources.

Unfortunately, ErsatzLisp lacks everything which makes up "true" PicoLisp: Speed, small memory footprint, and simple internal structures.

Performance is rather poor. It is 5 to 10 times slower, allocates a huge amount of memory at startup (600 MB vs. 3 MB), and needs 2.5 to 4 times the space for runtime Lisp data. But efficiency was not a major goal. Instead, performance was often sacrificed in favor of simpler or more modular structures.

There is no support for

- raw console input ('key') and line editing
- child processes ('fork')
- interprocess communication ('tell', 'hear', 'ipc', 'udp' etc.)
- databases (external symbols)
- signal handling

47.1.1 Invocation

Ersatz PicoLisp can be started - analog to 'pil' - as

```
$ ersatz/pil
```

This includes slighly simplfied versions of the standard libraries as loaded by the "real" 'pil' (without database, but with Pilog and XML support).

To start it in debug mode, use

```
$ ersatz/pil +
```

On non-Unix systems, you might start 'java' directly, e.g.:

```
java -DPID=42 -cp .;tmp;picolisp.jar PicoLisp lib.l
```

Instead of '42' some other number may be passed. It is used to simulate a "process ID", so it should be different for every running instance of Ersatz PicoLisp.

47.1.2 Building the JAR file

The actual source files are

```
sys.src  # The system
fun.src  # Function definitions
```

The PicoLisp script "mkJar" will read them, generate the Java source file "PicoLisp.java", compile that with 'javac', and pack the result into a JAR (Java Archive) file. "mkJar" expects to be run in the "ersatz/" directory, e.g.:

```
$ (cd ersatz; ./mkJar)
```

48

Ersatz PicoLisp Java Reflection API

Alexander Burger

abu@software-lab.de

Summary. This article introduces the *Ersatz PicoLisp Java Reflection API* and its most important functions.

48.1 Introduction

As Ersatz Lisp (available in the "ersatz/" directory in the Picolisp release, or as a separate tarball) is completely written in Java, it comes with a set of dedicated functions to interface to the Java Reflection API.

48.2 Important functions

48.2.1 The `java` function

The central function is `java`. It comes in four forms:

```
(java 'cls 'T ['any ..]) -> obj
```

is a constructor call, returning a Java object. `cls` is the name of a Java class. The optional `any` arguments are passed to the Java constructor.

```
(java 'cls 'msg ['any ..]) -> obj
```

calls a static method `msg` for a class `cls`.

```
(java 'obj 'msg ['any ..]) -> obj
```

calls a dynamic method `msg` for an object `obj`. The optional any arguments to the above three forms can be

- T or NIL, then the type passed to Java is boolean
- A number, then it must fit in 32 bits and will be passed as int
- A normal symbol, then it will be passed as String
- A Java object, as returned by java or one of the xxx: conversion functions (see below).
- A list, then all elements must be of the same type and will be passed as Array.

```
(java 'obj ['cnt]) -> any
```

converts a Java object to the corresponding Lisp type. Supported types, and their conversion results are:

- Boolean objects are converted to T or NIL
- Byte, Character, Integer, Long or BigInteger objects are converted to numbers
- Double objects are converted to fixnums, using the scale cnt
- String objects are converted to transient symbols
- Arrays of byte, char, int or long are converted to lists of numbers
- Arrays of double are converted lists of fixnums, using the scale cnt

Example:

```
: (setq Sb (java "java.lang.StringBuilder" T "abc"))
-> $StringBuilder
: (java Sb "append" (char: 44))
-> $StringBuilder
: (java Sb "append" 123)
-> $StringBuilder
: (java Sb "toString")
-> $String
: (java @)
-> "abc,123"
```

```
: (setq S (java "java.lang.String" T "abcde"))
-> $String
: (java (java S "getBytes"))
-> (97 98 99 100 101)
```

```
: (java "java.lang.String" T (mapcar byte: (100 101 102)))
-> $String
: (java @)
-> "def"
```

48.2.2 The `public` function

To access public fields in Java objects or classes, public can be used:

```
(public 'obj 'any ['any ..]) -> obj
```

Returns the value of a public field any in object obj.

```
(public 'cls 'any ['any ..]) -> obj
```

Returns the value of a public field `any` in class `cls`. In both forms, the optional `any` arguments will in turn access corresponding fields in the object retrieved so far.

Example:

```
: (public "java.lang.System" "err")
-> $PrintStream
: (java @ "println" "Hello world")
Hello world
-> NIL
```

48.2.3 The `interface` function

To create an interface object:

```
(interface 'cls|lst 'sym 'fun ..) -> obj
```

Creates an interface, i.e. a set of methods for a class `cls` (or a list of classes `lst`).

Example:

```
#!ersatz/pil
(let
   (Frame (java "javax.swing.JFrame" T "Bye-Frame")
      Button (java "javax.swing.JButton" T "OK") )
   (java Frame "add" "South" Button)
   (java Button "addActionListener"
      (interface "java.awt.event.ActionListener"  # When button is clicked,
         'actionPerformed '((Ev) (bye)) ) )       # Exit PicoLisp
   (java Frame "setSize" 100 60)
   (java Frame "setVisible" T) )
```

48.2.4 Type conversion functions

Finally, we have a set of type conversion functions, to produce Java objects from Lisp data:

```
(byte: `num|sym) -> obj
(char: `num|sym) -> obj
(int: `num) -> obj
(long: `num) -> obj
(double: `str `cnt) -> obj
(double: `num `cnt) -> obj
(big: `num) -> obj
```

For a more elaborated example, take a look at the article Swing REPL written in Ersatz PicoLisp.

A

GNU Free Documentation License

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if

there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License. I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make

the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.