Pico: Scheme for the Kids
-or-
The Power of Value Binding

Wolfgang De Meuter
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
wdmeuter@vub.ac.be

Sebastián González
Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
sgm@ingi.ucl.ac.be

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
tjdhondt@vub.ac.be

ABSTRACT

Our problem was to find a computer language to teach computer science to freshmen in exact sciences other than computer science. We were in search for a language that has the power of languages like Scheme, the readability of 'plain' mathematics, and the realism of contemporary imperative languages. To the best of our knowledge, no existing language is successful in uniting all these requirements. In the paper we show how a simple mathematical notation for function application combined with a original suite of parameter passing schemes is the key to the unification we were after. The outcome is called Pico, a simple powerful extensible language that is also easy to read.

1. INTRODUCTION

After years of teaching Pascal [12] in a computer science introductory course for freshmen in exact sciences other than computer science[1] we had to face the fact that the overall results were deplorable. Many students kept on struggling with the syntax, static typing rules, the positioning of semicolons, the difference between procedures and functions, the necessity to compile and so on. As a result the programming skills of most students remained regrettable and instead of being an inspiration for further exploration of computer science, the introductory course acted as a drag: students got demotivated, hated computer science and identified it with the technological fuss associated with Pascal compilers.

We therefore started a project to redesign the entire course. One of the problems we really wanted to get rid of was the laborious edit-compile-run cycle which we experienced to be a clog. Therefore we excluded languages with a "compilation culture" like Ada [10], C [7] and Java [4]. These languages were also way too complex as they are even richer than the Pascal we were about to abandon. We also wanted a language with automatic GC since manipulating pointers and managing memory was far beyond the skills of the intended audience. Pure functional programming was also ruled out: for many students, the aforementioned introductory course is the only one they will ever get during their university studies. One of the goals of the course was therefore to accustom them with the notion of a changing memory, a characteristic that is still inherent to mainstream computer science practice. All these restrictions led us to languages like Scheme [6] or Smalltalk [2]. But we were left no other choice than to admit that their simple regular syntax and semantics even take our computer science students two years to fully master.

That's why we started to dream about a simple language in which students were motivated into exploratory programming in a read-eval-print-loop as is the case with Scheme, but with a language that is:

- *easy to read.* The language had to look like calculus because that is about the only experience the intended audience had with formal languages.
- *as powerful and simple as Scheme.* We wanted a small number of powerful concepts instead of the baroque concept set of Pascal, Ada or Java.
- *extensible* in the same way Scheme and Smalltalk are. Instead of focussing on learning by heart a fixed control structures suite, we wanted that set to be easily extensible, much in the same way Smalltalk control structures are.

At first sight, this seemed impossible. The extensibility of a language in the spirit of Smalltalk or Scheme goes hand in hand with the simplicity and regularity of their syntax. However, the "easy to read" requirement seemed to be diametrically opposed to this. As we show, the exploration

[1] Physics, Chemistry, Mathematics, Biology, Biotechnology, Geography and Geology.

of an original suite of parameter passing techniques was the key to our quest. The outcome is called Pico.

1.1 The Acting Forces

In current day academic language design, syntax is no longer an exciting study field. But as explained in the introduction, in our case this was one of the actual endeavors of the project. The Pico experience taught us that it is not easy to come up with a syntax that is easy to read, very orthogonal and that allows the specification of a simple and regular extensible semantics.

When looking at existing languages, we can divide them into three flavors when it comes to their syntax:

An Irregular Keyword-based Syntax. The first kind of syntactic flavor is best known because so many popular languages have a syntactic system belonging to this flavor. The general idea is that the language is built around a number of keywords each with dedicated rules of how sub-expressions or sub-statements are to be centred around and grouped by those keywords. Examples of such languages are C, Pascal, Ada, Java, C++ and many more. These syntactic systems are irregular in the sense that just about every construct of the language has dedicated rules as to how instances of that construct are to be written down. Languages like these have the advantage of being easy to read. But they are hard to write. One really has to know their syntactic system thoroughly in order to construct programs correctly. As such, teaching the syntax in an encyclopedic, long-winded way is the only option. Furthermore it is not easy to equip such a language with an extensible syntax and semantics.

A Regular Syntax without Special forms. In languages of the second flavor, the syntactic rules are extremely simple in the sense that everything is expressed using one single syntax rule. Examples of such languages are pure functional languages (where everything is a function application) and Smalltalk or Self [11] (where everything is a message send). In pure functional languages it is possible to express everything as a function application since these languages usually support lazy evaluation allowing "keywords" to be functions whose arguments are not evaluated unless this is really necessary. However, as explained before, pure FP was ruled out. Let us then have a look at Smalltalk. This *is* an imperative language and since laziness doesn't combine too well with side effects, it *does* use eager evaluation, meaning that arguments are always evaluated before a message is sent. Therefore, in Smalltalk, arguments of "language messages" like `ifTrue:ifFalse:` must be manually wrapped in a lambda (called a block in Smalltalk) in order to delay them. The lambda is passed around and calling it then causes its body expression to be evaluated. Important for this paper is that the regularity of Smalltalk's syntax combined with its eager evaluation forces programmers to use its block system to clumsily prevent some expressions from being evaluated. Therefore, although languages in this second category are extremely easy to extend, programs written in them are often obscure, especially to novices.

A Regular Syntax with Special Forms. A third and final way of specifying syntax is a mixture of these keyword-based and regular syntax definition flavors. It is adopted by languages like Scheme and Prolog [9]. These languages are highly regular in the sense that all constructs are specified using exactly the same rules even though they semantically behave quite differently. In Scheme, almost every expression looks like a function call. Normal function calls follow eager evaluation, but for some "function names" (like `define`, `if`) a special -partially lazy- evaluation is defined. These "functions that bear a special status in the evaluator" are called *special forms* in Scheme. In Prolog we have the same situation where some predicate names behave differently from regular predicates. Just as is the case for languages in the second category, languages in this third category are easy to extend as is shown by the Scheme macro system. Furthermore, constructing programs in languages of this third category is quite simple. However, students have to meticulously "parse" programs in order to read them. We found that even sophomores in computer science that have been extensively exposed to Scheme [2] still find deciphering Scheme programs troublesome.

To the best of our knowledge an intersection of these three flavors that combines their advantages but shuns their disadvantages is empty. What we were actually after was a language with a syntax as easy to read as keyword-based languages, as easy to write as ordinary calculus, yet as orthogonal in syntax and semantics as the languages with a regular syntax. Furthermore, instead of hard coding all control structures with ad hoc rules to be taught in an encyclopedic tedious way, we wanted them to be specifiable and extensible in the language itself as in Smalltalk. But simpler.

As we will show in the rest of the paper, it seems that the key to solve our problem lies combining ordinary function application syntax from mathematics, with the invention of a suite of original parameter passing techniques. Parameter passing used to be a hot topic in the seventies when differences between call-by-value and call-by-reference were exploited in programming languages. However, the distinction between these parameter passing schemes was not really a result of language design considerations, but was rather driven from an implementational point of view: should a thing or a pointer to the thing be passed around. As far as we know, Ada was the only language (with its `IN`, `OUT` and `IN OUT` parameter annotation system) in which programmers were able to specify how parameters had to behave from a *conceptual* point of view. In our work we invented different conceptual parameter passing schemes which was the key to solve our language puzzle.

1.2 The Pico 3x3 Syntax System

The Pico syntax is two-tiered. The first layer consists of a suite of simple rules that allow for the construction of trivial programs much in the style of functional programming. The second layer adds a few bells and whistles which at first might seem like rough edges to the proposal. However upon closer inspection, it are exactly these that entailed the full power of Pico as an extensible language. In this section, we will explain the first layer of the syntax. The following sections explain the additional syntactical constructions. Before moving on to an in-depth explanation of the language, the following Pico code snippet, meant as a teaser, gives a

[2]We teach our CS freshmen SICP [1] plus an Algorithms & Data Structures course, and require them to do a 'big' two-semester programming project.

general flavor of the language. It is a naive implementation of the famous quick sort algorithm:

```
QuickSort(V, Low, High):
  { Left: Low;
    Right: High;
    Pivot: V[(Left + Right) // 2];
    until(Left > Right,
      { while(V[Left] < Pivot, Left:= Left+1);
        while(V[Right] > Pivot, Right:= Right-1);
        if(not(Left > Right),
          { Swap(V, Left, Right);
            Left:= Left+1; Right:= Right-1 },
          false) });
    if(Low < Right, QuickSort(V, Low, Right), false);
    if(High > Left, QuickSort(V, Left, High), false) }
```

The first layer of the Pico syntax and semantics is explained by means of the three by three matrix depicted in table 1. This two dimensional matrix emerges from taking all possible combinations of two design decision dimensions. First, a Pico expression is always evaluated in the context of an environment (called a dictionary in Pico terminology) and one dimension of understanding Pico consists of viewing Pico in terms of manipulations of this dictionary. Therefore, each row in table 1 gives syntax to add something to the dictionary (with `:`), to refer to something in the dictionary, and to update something in the dictionary (with `:=`). Second, all Pico values (aside from literal values like integer numbers, fractions, texts and void) are described by what we call *invocations*. These invocations constitute the horizontal dimension of the Pico language design space. Invocations are used to talk about values in an atomic way, to talk about tables, and to talk about functions. Tables are Pico terminology for what is usually known as arrays. The first kind of invocations are ordinary references `nam`. The second kind are of the form `nam[exp]` and the final ones look like $\texttt{nam(exp}_1\texttt{,...,exp}_n\texttt{)}$.[3].

Before doing so, let us emphasize the fact that in Pico, everything is considered to be a first class value: basic values, functions and tables all are 'on the same level': they can be passed around as arguments, can be returned from functions, can be used as the right hand side of an assignment and so on. Another important point to keep in mind is that Pico functions (in contrast to Scheme for example) always have a name. This decision was made because of the audience intened: indeed, mathematics does not feature something as anonymous functions (functions are always named f, g and h in mathematics and many students already consider names like `fac` or `fib` strange). Having said this, let us now run through the table:

- We start in the first column. A variable is installed in the current dictionary using `name: exp` where `exp` designates the initial value. `exp` can be anything since expressions *always* yield a value. Hence, the expression `v: (n: (t: "hello"))` will install three variables which are all initialized to `"hello"`. Referring to a variable simply happens by naming it. Finally, variable assignment[4] almost is the same as variable definition. The only difference is that the variable is already expected to be in the dictionary.

- Manipulating tables (i.e. arrays) is the topic of the second column. Table indexes run from 1 to the size of the table. Let us start with table definition which is the only expression type for which the semantics is not trivial from what one would normally expect. In an expression like $\texttt{t[exp}_1\texttt{]: exp}_2$, $\texttt{exp}_1$ is evaluated to yield an integer acting as the size of the new table being installed in the dictionary under the name `t`. The entries of that table will be filled with the result of evaluating $\texttt{exp}_2$. The unexpected behavior lies in the fact that $\texttt{exp}_2$ is evaluated again and again *for every* entry of the table. This allows for expressions like `t[n]: (i:=i+1)` to create a table with `n` numbers in ascending order (provided that a variable `i` exists somewhere). Referring to a value in a table happens with an expression of the form `nam[exp]` whose semantics is as expected. Finally, updating a table position happens with a $\texttt{nam[exp}_1\texttt{]:= exp}_2$ expression, of which the meaning is predictable as well.

- The final column in table 1 is about function manipulation. Function scoping is lexical as in Scheme. But in contrast to Scheme, Pico functions always carry a name. A function with name `nam`, parameters $\texttt{par}_1\texttt{, ... ,par}_n$ and body `exp` is defined by the expression $\texttt{nam(par}_1\texttt{, ... ,par}_n\texttt{): exp}$. The expression can be anything. Hence, the expression:

  ```
  f(x,y): g(t,u): x+y+t+u
  ```

  will define a function `f` with two parameters that will return another function of two parameters. An expression of the form $\texttt{nam(exp}_1\texttt{, ... ,exp}_n\texttt{)}$ will lookup `nam` and check it to be a function. After checking the number of arguments, the parameters will be bound from left to right to the arguments[5]. In the first layer of Pico we are describing here, eager evaluation is used such that all arguments will be evaluated. In the following sections we will see how to preclude arguments from being evaluated. Last, function assignment happens in the same way as function definition. The name is looked up and its associated value (whether it is a function or not) will be garbage collected. The name will be (re)associated with a new function.

  Notice that Pico allows operators to be used in infix notation. This is syntactical sugar, though. An operator application like `x+y` will be replaced by the parser by a regular function application `+(x,y)`. Operators are recognized by their name which consists of special symbols such as `+`, `*` and the like.

This ends our explanation of the first level of understanding Pico. Notice that the curly braces used in the QuickSort

[3] In this paper, we give an informal explanation of the Pico concepts. A formal specification in the form of a meta-circular definition can be downloaded from `http://pico.vub.ac.be`.

[4] We use `:=` for assignment as we wanted to reserve `=` for equality tests.

[5] Notice, that in contrast to Scheme, we *do* stress the order (left to right) in which the arguments are evaluated and bound.

Table 1: Pico basic syntax

kind of invocation:	variable invocations `nam`	table invocations `nam[exp1]`	application invocations `nam(exp1, ... ,expn)`
definition	`nam: exp`	`nam[exp1]: exp2`	`nam(exp1, ... ,expn): exp` (see also sections 1.3 and 2)
reference	`nam`	`nam[exp]`	`nam(exp1, ... ,expn)` (see also section 2)
assignment	`nam:= exp`	`nam[exp1]:= exp2`	`name(exp1, ... ,expn):= exp`

teaser are not covered by this syntax. Furthermore, with what we know so far, it is not possible to define or even *use* a conditional `if` expression. Indeed, using an `if` with our syntax would force us to write it down as `if(condition, then, else)` causing all(!) arguments to be evaluated. In order to mend this, the following sections add some bells and whistles to this basic framework, which at first might seem rough edges to the proposal. However, as we will show, they are the key to turning Pico into a practical extensible language.

1.3 Syntactic Sugar Due to `@`

As explained in the previous section, the Pico syntax is two tiered. The second layer consist of what we usually call "two footnotes to table 1". This section presents the first footnote which is about functions that take a variable number of arguments. Such functions are defined using the `@` notation. The following excerpt shows a function that can be called with any number of arguments. It returns the sum of the first three provided.

```
sum3@args: args[1]+args[2]+args[3]
```

Definition expressions of the form $\texttt{nam}_1\texttt{@nam}_2\texttt{: exp}$ will install a new function named $\texttt{nam}_1$ in the dictionary with body `exp`. The function can be called with any number of arguments which will be evaluated from left to right, and collected in a table of the appropriate size. This table is passed under the name $\texttt{nam}_2$. Inside the body of the function, one can thus use $\texttt{nam}_2$ as a normal Pico table as explained in the previous section. In the example, the call to `sum3` will always succeed, no matter how many arguments are passed. Inside `sum3` the arguments are accessible by manipulating the `args` table. Of course the body of `sum3` will fail whenever less than three arguments are passed to `sum3`.

The feature discussed here allows us to write our own `begin` function:

```
begin@args: args[size(args)]
```

This `begin` function can be called with any number of arguments. These will be bound, as always, from left to right. Of course, this is what one expects when writing an expression like $\texttt{begin(exp}_1\texttt{, ... ,exp}_n\texttt{)}$. Inside `begin`, the values of the expressions will reside in the `args` table. The body determines its size and returns its last entry (cf. the value of `begin` in Scheme).

Another function that comes handy when writing Pico programs, is the `tab` function:

```
tab@args: args
```

A call to `tab` like `tab(1, 2.0, 3, "hello world")` will evaluate the arguments, and pass them as a table; `tab` simply returns that table. Hence, `tab` allows for 'inline' table construction.

Syntactic Sugar Because `begin` and `tab` are used so often, the resulting number of parentheses starts to clutter up programs. Therefore the Pico parser allows some syntactic sugar:

- First, expressions can be grouped between curly braces and separated by semicolons as in $\{\ \texttt{exp}_1\texttt{; ... ;exp}_n\ \}$. However, this notation is merely syntactic sugar: the parser will replace this expression with a call to `begin`. Hence, internally the parse tree generated by the above expression will be the same as the parse tree of $\texttt{begin(exp}_1\texttt{; ... ;exp}_n\texttt{)}$. This means that redefining `begin` with the function assigment feature for example, will give Pico programs a new meaning, also when they use the curly braces notation.
- A second form of syntactic sugar that was added to the parser is the ability to construct tables 'inline' by enumerating their elements between brackets and separated by commas. Such expressions are syntactic sugar for the corresponding calls to `tab`. Hence, an expression `[[1,2], [3,4]]` is treated by the parser as `tab(tab(1,2), tab(3,4))`.

We found that introducing these two shorthands considerably improved the readability of Pico programs. Nevertheless, we managed to do so without really extending the suite of Pico concepts.

2. VALUE BINDING SEMANTICS

The first "footnote" to the Pico 3x3 syntax system has been presented in the previous section. We now proceed with the second annotation. When evaluating a function application, formal parameters are being bound to the actual arguments, one by one. During this process the lexical environment of the called function is progressively extended with new bindings yielding an extended environment in which the body of the function will be evaluated. This lexical scoping rule is exactly the same as in Scheme.

The thought experiments we carried out when designing Pico led to a generalized notion of parameter passing which we call *value binding*. Value binding is regarded as a general process that is not only applied to parameter passing but to variable definition in general. To bind a pair of values $x \leftarrow y$ means to associate them in a certain way, depending on the form of x. In that binding, x is called the *parameter* and y the *argument*. This nomenclature reflects the fact

that the ideas we will explain are in fact inspired on the study of parameter passing mechanisms, and they find their best application therein. Nevertheless, the generalization we developed constitutes a step forward in the orthogonality of the language semantics, as we will show next. Using the general notion of value binding, the definition `a: [1, 2, 3, 4]` invokes the exact same binding semantics in the interpreter as the function application `f([1, 2, 3, 4])`, supposing that `f(a): { f's body }` has been defined. Value binding is applied both during variable definition and parameter passing. Unlike other languages, there is only one concept for both mechanisms. In C or Java for instance, the definition `int a[] = { 1, 2, 3, 4 }` is valid, whereas `f({ 1, 2, 3, 4 })` with `f` defined as `void f(int a[]) { f's body }`, is not. In this case, the process of binding the variable `a` with its initial value is not the same as the binding of the parameter `a` with its argument. But in Pico, it is.

Apart from extra orthogonality simplifying the definition and implementation of the language, the notion of binding led us to consider arbitrary couples (x, y) of expressions, where x is not restricted to identifiers. Referring back to the last column of table 1 we notice that the formal parameters $\texttt{exp}_1, \ldots, \texttt{exp}_n$ of a function definition can, in fact, be arbitrary expressions! How a value binding semantics should be defined for each parameter type was an important part of our language design. Although there are many cases that do not make sense (e.g. it does not make sense to bind the parameter expression `3` to the argument expression `6`), we found some very interesting nontrivial cases which will be explained in this section and in section 3. The reason to split up the sections is to be consistent with other documentation material of Pico. The binding semantics explained in this section is part of the "standard Pico semantics" for a few years now. We understand it completely and it is a feature of every Pico implementation around (see `http://pico.vub.ac.be`). The value binding schemes of section 3 are more experimental and their repercussions on the pragmatics of the language are less well understood. We start the presentation with the most basic case, namely value binding for parameters that are ordinary variable references.

2.1 Value Binding for Reference Expressions: call-by-value

When a formal parameter of a function is an ordinary reference (i.e. an identifier), the Pico semantics prescribes that the passed argument will be evaluated and the dictionary is extended with one new association mapping the identifier to the evaluated value. This happens for instance when variables are defined:

```
a: 1; b: 2
```

or upon application of a function whose parameters are (ordinary) references:

```
f(a, b): { ... };  f(1, 2)
```

The function application causes the same binding processes to occur than the two variable definitions above. This semantics of binding a value to an identifier, often referred to as *call-by-value*, is the most common in programming languages.

2.2 Value Binding for Application Expressions: call-by-expression

Binding is more complex when parameter expressions are application invocations, i.e. constructions of the form $\texttt{nam}(\texttt{exp}_1, \ldots, \texttt{exp}_n)$. For a parameter like this, binding a given argument `exp` to it is achieved by creating a new function that corresponds to:

$\texttt{nam}(\texttt{exp}_1, \ldots, \texttt{exp}_n)$`: exp`

Hence, binding an expression to an application invocation parameter consists in constructing a new function with formal parameters $\texttt{exp}_1, \ldots, \texttt{exp}_n$ and body `exp`. The closure of this function is the current evaluation environment, which is "the environment of definition of the function", thereby following the lexical scoping rules. Let us exemplify this using the function definition:

```
g(f(a,b),x,y):  if(f(x,y) > 0, x, y)
```

`g` has three parameters, `f`, `x` and `y`, the first of which is an application expression, the type we are discussing here. The other two parameters are ordinary reference expressions discussed in section 2.1. Hence upon a function call `g(a+b, 1, 2)`, the value bindings `f(a,b): a+b`, `x: 1` and `y: 2` are performed, as always from left to right, and `g` can use them as needed. The point to highlight here is that the body and scope of `f` is dynamically associated upon each invocation of `g`. We call this type of parameter passing *call-by-expression*.

Although this might seem like an extremely obscure language feature at first, the value binding semantics explained here turns out to be natural for people without prior programming experience. Consider for example the bisection method to find the zero of a numerical function. Because computer scientists are so much used to work with higher order functions, they will say(!) that a numeric procedure for this method takes "a function f", boundaries a and b and an accuracy *epsilon*. High school students, however, will rather say that the bisection method can be used to find the zero of "a function f of x" between a and b with precision *epsilon*. The subtle difference between "f" and "f of x" is often a source of problems when teaching higher order functions: *we* as computer scientists rather work with f because we are used to lexical scoping and local parameters. Students with only an education in high school mathematics will prefer the second nomenclature. The value binding semantics explained in this section allows this to be written down in the most natural way:

```
zero(a, b, f(x), epsilon):
   c: (a+b)/2;
    if(abs(f(c)) < epsilon,
    c,
    if(f(a)*f(c) < 0,
      zero(a, c, f(x), epsilon),
      zero(c, b, f(x), epsilon)))
```

A call of `zero` such as `zero(-3, 3, x*x-6, 0.01)` will bind the reference invocations `a`, `b` and `epsilon` to `-3`, `3` and `0.01` respectively. The application invocation `f(x)` will be "value bound" to `x*x-6` which will internally create a function `f(x): x*x-6`.

2.3 Value Binding for Table Invocations

As the value binding semantics are orthogonal throughout the language, binding an argument `exp` to a parameter which is a table invocation expression `t[n]` results in a new table being created, where the passed argument is used as the initializer expression of the table's entries, i.e. `t[n]: exp`. Thus defining

```
f(t[n]): { f's body }
```

and then invoking `{ i: 0; n: 3; f(i:= i+1) }` results in the table `[1, 2, 3]` bound to the identifier `t` in the environment of execution of `f`'s body. This semantics for table invocation parameters is a homogeneous extension of the value binding explained in the previous section. We have to admit that we do not have a lot of experience with this feature currently.

2.4 Evaluation and Epilog

Measuring expressiveness and quality of programming languages is not an exact science. Nevertheless we have the feeling of having established a consistent language that is easy to learn and implement. This was confirmed by the results we obtained in the introductory computer science course we were talking about in section 1. While the course of the Pascal era majorly consisted of explaining Pascal based on simplistic examples like determining greatest common divisors, the size and simplicity of Pico actually allowed us to focus on programming! In the 30-hours course, we exposed the students (after explaining Pico itself) to four experiments designed to stimulate their appetite for further exploration of computer science in their respective fields: simulation of population growth (biology), triangularization of matrices (mathematics), simulation of forced oscillations (physics) and querying of a small database representing the periodic table of elements (chemistry). Again, all this was done with 18-year olds in 30 hours.

But apart from these results, Pico is also interesting from an academic point of view due to the value binding system explained in the previous section. As we will show now, the value binding mechanism for function application invocations allows for an easy extension of the language in the same way Smalltalk and Scheme can be extended. This point is made by the fact that Pico's boolean system and its complete suite of control structures were implemented in Pico itself. Pico's boolean system is actually an adaptation for imperative languages of the famous Church booleans. The idea thereof is to define booleans as functions that choose between two options given as arguments. Based on these definitions, one of the core control functions of Pico, the `if` decision function, can be introduced in the same way it was proposed by λ-calculus:

```
true(t,f): t
false(t,f): f
if(cond, t, f): cond(t, f)
```

This would work as expected, `if(true, 1, -1)` = `true(1, -1)` = 1. There is however a problem in this overly simplistic scheme: both branches passed as arguments to `if` will be evaluated when bound to the parameters `t` and `f` because it involves regular call-by-value. Thus code like:

```
if(true, display(1), display(2))
```

does not behave as expected. Call-by-expression was the key to making Church's boolean system applicable in Pico:

```
true(t(), f()): t()
false(t(), f()): f()
if(cond, t(), f()): cond(t(), f())
```

The `()`-parameters delay evaluation of the argument expressions because calling the function will not evaluate the argument but will instead create a function of zero parameters whose body will be the argument. This allows us to perform evaluation in a controlled way, by means of function application. Operations like `and`, `or` and `not` are also implemented in Pico this way.

Handy, but not essential of course, are some "commonly known" Algol-like control structures such as the `while` construct:

```
while(cond(), exp()):
  { loop(value, pred):
      pred(loop(exp(), cond()), value);
    loop(void, cond()) }
```

We leave it to the reader to figure out how this works because it is not essential for the rest of the paper. We merely wanted to prove our point of having an easily extensible language due to call-by-expression.

In all these examples, thunks (i.e. functions of zero arguments) are used to achieve what Smalltalk does with blocks. However, in Smalltalk "thunkification" is requested by the *user* of the control structure. We think that we have obtained a cleaner syntax and semantics with the same power. Furthermore it is not necessary to enrich the language with extra concepts like macros and quasiquoting as was needed to render Scheme's special form suite extensible.

3. VALUE BINDING: ROUND 2

In this section we describe some experiments we conducted in extending the standard Pico explained so far. In the value binding possibilities $x \leftarrow y$ we have presented until now, the left hand side had to be one of the types listed in the header row of table 1, i.e. x was either a variable, table or application invocation. In this section we present binding semantics for two other types of parameters we have been investigating. Experiments nonetheless suggest [3] that both of them enrich the possibilities of the language. Section 3.1 presents the case in which x is a table[6] and section 3.2 the case where x is a quoted expression.

3.1 Value Binding for Table Expressions: call-by-table

If the parameter in a binding operation is a table (as opposed to a table invocation, explained in section 2.3), then the argument must be a table as well, of the same size. The entries of the parameter table are considered a parameter each, and similarly each entry of the argument table is considered an argument. A 1-1 binding process is then applied over the two sets of parameters and arguments, from left to right. Let us consider some examples:

```
[a, b]: [1, 2]
[f(x), g(x)]: [x+2, x*2]
```

[6]Note that a "table" is different from a "table *invocation*". A table looks like `[exp,...,exp]`, a table invocation looks like `t[exp]`.

The first binding operation defines two variables `a` and `b` initialized to 1 and 2 respectively, and the second defines two functions `f(x): x+2` and `g(x): x*2`. Using this binding semantics in the formal parameters of a function works as expected:

```
f([a, b]): { f's body }; f([1, 2])
```

Upon invocation, `f`'s body will see the variables `a: 1` and `b: 2` defined in its execution environment.

One useful application we devise for this binding semantics is to "catch up" table return values from functions that return more than one result in the form of a table:

```
f(i): [i, i+1, i+2]; [x, y, z]: f(1)
```

The result value of `f`'s application – the table `[1, 2, 3]` – is bound to the receiving parameter `[x, y, z]`, implying that the bindings `x: 1`, `y: 2` and `z: 3` are performed, in that order, in the evaluation environment of the application.

Another use of table parameters consists in emulating the "head/tail" parameter matching mechanism available in Haskell [5]:

```
length [] = 0
length (head:tail) = 1 + length tail
```

In Pico, we can first define a list framework similar to that of most functional languages:

```
empty: [void, void]                  -- the empty list
prepend(elem, list): [elem, list]    -- like Scheme's cons
head@list: list[1]                   -- like Scheme's car
tail@list: list[2]                   -- like Scheme's cdr
```

Then, we can define functions with "head/tail pattern parameters", like in Haskell:

```
length([head,tail]):
  if(is_void(tail), 0, 1+length(tail))
```

Note that the table parameter `[head,tail]` of `length` acts as a pattern-matching construct: when the function is invoked passing a list as argument, the `head` and `tail` of the list will be bound accordingly.

We have shown the usefulness of table binding both to match arguments and to "catch up" results from functions. But what is more important is that (in our experimental version of Pico discussed here) table binding is, together with reference binding, the most used binding mechanism of the language. It is used whenever a fixed-arity function is applied, because the formal parameters of fixed-arity functions are represented as normal Pico tables of expressions. When the function is invoked, the table of arguments is bound to the table of parameters, thus invoking the semantics explained in this section. This semantics happens to be the same as that described in section 1.2 (the table sizes must coincide, and the binding is performed from left to right). For variable-arity functions, i.e. `nam@args`-style functions (recall section 1.3), normal reference binding is performed: the table of arguments is bound to the only parameter `args` declared by the function.

3.2 Value Binding for Quoted Expressions: call-by-quoting

Now we will describe a simple, although rather innovative parameter passing mechanism that was inspired by the quoting mechanism of Scheme. In an experimental Pico version, quoting an expression (by prefixing it with a quote `'`) is made possible.

When the parameter x of an association $x \leftarrow y$ is a quoted expression, the value binding is done by unquoting the parameter and value binding it to the quoted argument. For instance when binding `1` to the parameter `'x`, first the expression `'x` is unquoted, resulting in the ordinary reference `x`, then `1` is quoted and bound to `x`, i.e. the binding `x: '1` is performed. Another example:

`('y): i+1` – defines `y` as the quoted expression `'i+1`

The parentheses are needed to force the binding of the quoted reference `'y` to the expression `i+1`; if the parentheses were omitted, evaluating `'y: i+1` would just quote the whole expression `y: i+1`, without invoking any binding process at all.

This mechanism is also used to define functions with quoted parameters. We call the parameter passing technique that results from this value binding technique *call-by-quoting*. For example, the function `f('a): a` just quotes its argument and returns it.

Quoting equips Pico with a very simple reflection [8] mechanism as it is a way to turn Pico programs into Pico data structures manipulatable by other Pico programs: the parse tree of every expression in Pico is made available as a regular Pico table. Hence, the result of quoting an expression is nothing but a table representing the parse tree of the expression. Since the parse tree can be modified and evaluated using constructs from the language itself, a door to reflection is open.

A more complicated example, illustrating a very convenient combination of call-by-quoting and the reflective architecture, comes from an experiment we did to implement a simple object-oriented extension of Pico. In this system we made a representation of objects in Pico (basically objects = dictionaries) and implemented mechanisms like method lookup and method application as functions that operate on this representation. The example illustrating call-by-quoting involves the implementation of a message sending operator `<-` to be used like `obj<-msg`, where `obj` is the receiver object and `msg` is a regular function application invocation `nam(`$\texttt{exp}_1$`, ... ,`$\texttt{exp}_n$`)`. The implementation of this operator is as follows[7].

```
<-(receiver , ['name, args] ):
  apply(lookup(receiver, name), args, receiver)
```

The inner workings of method lookup and invocation is irrelevant for this paper. Let us concentrate on the formal parameters by looking at the evaluation of the expression `obj<-msg(1, 2)`, i.e. `<-(obj, msg(1, 2))`. This gives rise to the bindings

```
receiver: obj
```

[7]Recall from the end of section 1.2 that Pico allows infix notation for operators. Nevertheless they are nothing but functions.

```
['name, args]: msg(1, 2)
```

The first is an ordinary reference binding. As the second has a table on the left-hand side, it is a table binding, of the type described in section 3.1 where we explained that the argument should be a table having the same size as the parameter table. The reflective architecture of Pico implies that the expression `msg(1, 2)` (a function application invocation) has a table representation of the form `[msg, [1, 2]]`. Hence the second binding is in fact:

```
['name, args]: [msg, [1, 2]]
```

whose meaning is now clear. Table binding implies a 1-1 binding between the entries of two the tables, thus the bindings `'name: msg` and `args: [1,2]` are performed. In the end, the following variables are defined for the `<-` operator to use: `receiver: obj` (a Pico dictionary), `name: 'msg` (a reference invocation expression) and `args: [1,2]` (a table with the arguments). To conclude the example, note in the definition of the `<-` operator that parameter nesting has been used – the second parameter is a table expression which contains in its turn a quoted expression and a reference invocation expression.

A key benefit of call-by-quoting is that the quoting of arguments is requested from the function implementor's side rather than forcing the client to quote the arguments on each invocation (as opposed for instance to Scheme). What is more, as the previous example demonstrates, quoted parameters are sometimes indispensable. As in the case of call-by-expression parameters, call-by-quoting parameters are a way of avoiding special forms, since they delay the evaluation of arguments passed to functions. The way quoted parameters are used is pretty different, though. While the code contained in functions obtained by call-by-expression can be parametrized and comes along with an evaluation environment, quoted expressions constitute "raw" code alone, with no context at all. Functions are intended to be applied, whereas quoted code is intended – most of the time – to be manipulated reflectively. These constitute two radically different ways of using delayed arguments.

4. CONCLUSIONS

Our goal was to find a simple interpreted language that is as easy to read as keyword-based languages, as easy to write as basic calculus, and with the same power of highly regular languages like Scheme and Smalltalk. As we have shown, the basis for our work is a simple 3x3 syntactic system that allows us to define, update and refer variables, functions and tables. This system was turned into a powerful language by adding variable argument functions and by adding a suite of originally designed parameter passing mechanisms. The result is a language that has the full power of Scheme and Smalltalk, but which is much smaller and many times easier to implement. The (non reflective version of the) language was successfully used to introduce non CS students to topics as complex as data structures and higher order programming.

5. REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 2nd edition, 1996.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachussetts, 1983.

[3] Sebastián González. Design and Implementation of a Reflective Hybrid Functional / Prototype-Based Language Kernel. Master's thesis, Vrije Universiteit Brussel, in collaboration with École des Mines de Nantes, August 2002.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, Boston, Massachussetts, 2nd edition, 2000.

[5] Simon Peyton Jones, John Hughes, and (Eds.). Haskell 98: A Non-strict, Purely Functional Language, February 1999. Online: <http://www.haskell.org/onlinereport/> [visited: October 2002].

[6] Richard Kelsey, William Clinger, Jonathan Rees, and (Eds.). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, October 1998. Also in Higher-Order and Symbolic Computation, 11(3):7–105, 1998.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[8] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 147–155. ACM Press, December 1987.

[9] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Advanced programming techniques. MIT Press, Cambridge, Massachussetts, 2nd edition, 1994.

[10] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. International Standard ISO/IEC 8652:1995(E).

[11] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 227–242, New York, NY, December 1987. ACM Press.

[12] N. Wirth. The Programming Language Pascal. In *Acla Informatica*, volume I, pages 35–63, 1971.