# Pico - a tiny functional language

## Language Description

### Functions are Just Underspecified Expressions

One of the main ideas in Pico is that the only difference between a function and a constant expression is that a function leaves some arguments unspecified. What do I mean by that? Well consider the expression:

1 < 2

This value is the boolean constant true, no matter what, because every argument has been specified. Now consider:

a < b

This value can be true or false, since we can choose any a and b. Then a function is simply an expression, the only difference being that it might not be constant.

Expressions can be conditional, like a ternary operator in C:

unsigned fact(unsigned i) { return i <= 1 ? i : i \* fact(i-1); }

We are essentially saying that the statement i <= 1 ? i : i \* fact(i-1) itself is a function, because we don't know what i is. But it can be made into a constant expression by specifying any i, in C by calling fact(/\*some number\*/).

We therefore don't need to separate functions and expressions, because from a programmatic point of view, all that matters is whether there are any unresolved symbols in an expression, and then it becomes a function.

fact = { if i:num < 2 then 1 else i:num \* fact(i-1) }

First things first: The curly braces introduce a new lexical scope, so that even if there is some i defined in an earlier context, this i will override it (otherwise, we'd have an error). The :num indicates i is a number.

Now even though we didn't explicitly say it, because there is an unresolved symbol in fact (namely i), we know that fact is a function which takes one variable; i.e. it's a function which requires one variable to become a constant expression. We can also see that the way to make this function into an expression is to satisfy its unresolved symbol:

a = fact(10)a = 3628800 <-- equivalent

### Extended Currying via Partial Specification

What if we only provide some of an expression's arguments? Consider:

foo = { a:num - b:num \* c:num }foo' = foo(2)

We can see that foo is a function with three unresolved symbols. What then, is foo'? Functional programmers will know this as currying: a is no longer unresolved, so foo has two unresolved symbols; therefore foo' is a function on two variables. Then the statement bar = foo'(3,4) resolves to 2 - 3 \* 4 = -10. But what about b and c? If we set b equal to 2 instead of a, then we still have two unresolved symbols. But now calling this function on (3,4) would resolve to 3 - 2 \* 4 = -5. We can see that on a function with initially n symbols unresolved, we can potentially construct 2^n - 1 child functions from it (not 2^n since providing all n arguments results in a constant expression). This is an extended idea of currying. Pico provides a syntax for this:

foo = {a:num \* b:num - c:num}foo1 = foo(2) <-- foo with a=2foo1(3,4) <-- 2foo2 = foo(,2) <-- foo with b=2foo2(3,4) <-- 2foo3 = foo(,,2) <-- foo with c=2foo3(3,4) <-- 10foo4 = foo(1,,2) <-- foo with a=1, c=2foo4(5) <-- 3

To make things clearer, we could use a dictionary-style syntax:

bar = {(a:num \* b:num + c:num ^ 2) ^ d:num }bar1 = bar(a=5, c=4)bar1(6, 2) <-- 2116bar(a=5, c=4)(b=6, d=2) <-- 2116

Of course, inside of an expression there might be other expressions, or assignments. Then we could have something like this tail-recursive factorial function:

fact = { fact' = { if n':num < 2 then acc:num else fact'(n'-1, acc\*n') }, fact'(n:num, 1)}fact(10) <-- 3628800

Here fact' is internal to fact (and would not be visible to other functions, because of the curly braces). fact has one unresolved symbol, n, while fact' seems to have two, n' and acc. But actually fact' is constant for any input of n, so we can consider its symbols to be resolved. But if n' is not provided, then the story changes. So we could write the same thing as

fact = {fact' = {if n:num < 2 then acc:num else fact'(n-1, acc\*n)}, fact'(,1)}fact(10) <-- 3628800

This works the same way, because fact(,1) provides one symbol to fact' (calling it with acc = 1), leaving one symbol (n) unresolved. Then fact is still a function of one variable. It might be easier to think of the 10 passed into fact being effectively handed to fact'.

### Anonymous recursion

The function fact' is only relevant inside of fact, and is only used once, so we really shouldn't need to name it. However, since it recurses, we need to be able to call it from inside itself, suggesting it needs a name. Pico solves this by introducing anonymous recursion. We can use the symbol $ to indicate "call this function again with these arguments". This meanse make the above definition of fact even more concise:

fact = {{if n:num < 2 then acc:num else $(n-1, acc\*n)}(,1)}

Note that the double curly braces are required; otherwise the $ would call fact itself with two arguments, which would be an error.

### More Partial Application Examples

Let's look at some more complicated examples of partial application, with a first look at function composition.

foo = {bar = i:num, bar}foo' = foo(1) <-- what happens here?

This one is obvious: foo' passes 1 into foo, which assigns bar to 1 and returns bar, so foo' equals 1. How about this?

foo = {bar = baz(i:num), bar}foo' = foo(1) <-- what happens here?

Here, foo' = bar(1) = baz(1). Let's make it a little more complex:

foo = { bar = { buzz = baz(i:num), herp = derp(j:num), buzz + herp}, bar}foo' = foo(1) <-- how about this?

The easy way to look at it is to find the first unresolved variable and replace it with 1, and see what we have. foo has no unresolved variables (returning the function bar), but bar has two: i and j. Resolving i resolves buzz, leaving bar as baz(1) + derp(j:num). Since foo resolves simply to bar, foo' = bar(1) = {baz(1) + derp(j:num)}, and foo' is a function of one variable.

### New Functions through Function Combination

In the example above, bar resolved to buzz + herp, but buzz and herp were each unresolved expressions themselves, so bar is a function created by combining two functions. Just as we can provide arguments to create new functions, we can add expressions to a function or combine it with other functions and create new functions.

foo = {a:num + b:num} <-- foo is a function of two variablesbar = {c:num - d:num} <-- bar is a function of two variablesbaz = foo \* bar <-- baz is a function of four variablesbaz(3,4) <-- equivalent to foo(3,4) \* barbaz(3,4,5,6) <-- equivalent to foo(3,4) \* bar(5,6) = (3 + 4) \* (5 - 6) = -1

Note that because functions have separate namespaces, combining two functions can result in namespace collisions. In these cases, if we want to use dictionary syntax to call the function, we have to indicate which function's variables we're supplying. This is similar to SQL.

foo = {a:num \* b:num}bar = {a:num / b:num} <-- bar can't see foo's a and b, so they are unrelated variablesqux = foo^bar <-- this is NOT the same as (a \* b) ^ (a \* b). It's the same as (foo.a \* foo.b) ^ (bar.a \* bar.b).lux = qux(foo.a = 4, bar.b = 5) <-- lux gives two args to qux, so lux is a function of 2 variablesbux = lux(,10) <-- second unbound variable is now bar.a, foo.b still unbounddux = bux(3) <-- no unbound variables, now we can resolve: <-- bux(3) = lux(1,10) = qux(4,3,10,5) = foo(4,3) / bar(10,5) <-- = (4 \* 3) ^ (10 / 5) = 12^2 = 144

Going further off of SQL inspiration, we could create aliases for some of these functions:

foo = {a:num \* b:num}bar = {a:num / b:num}qux = (foo f)^(bar b)lux = qux(f.a = 12, f.b = 3, b.a = 4, b.b = 8) <-- = (12 \* 3)^(1/2) = 6

### Typed vs. Dynamic and Unbound Variable Specification

Pico is a strongly typed language. Dynamic languages have their benefits, but there are many advantages to typing. One advantage in the case of Pico is simply that it better indicates which variables are unbound:

fact = {if n:num < 2 then acc:num else fact(n-1, acc\*n)} <-- function of two variablesfact1 = fact(,1) <-- partial application leaves fact1 a function of 1 variablefact2 = fact(acc=1) <-- different syntax, same resultfact3 = {fact(n:num, 1)} <-- note that this n is distinct from fact's n, so fact3 still has one unbound variable, and we need {} to indicate lexical scopefact4 = {fact(n = n:num, acc = 1)} <-- once again specifier:num means n is a new, unbound variablen = 10fact5 = fact(n,1) <-- n is defined, and this isn't a new lexical scope, so fact4 is the constant expression 3628800fact6 = fact3(n) <-- constant expression 3628800

We might extend the language to allow the writer (optionally) to indicate which unbound variables there are:

comp[i:num, j:num] = (if i < j then -1 else if i > j then 0 else 1)

The [] syntax would indicate the start of a new lexical scope, omitting the need for {} on the right side.

### Data Structures

How we handle data structures in Pico is still largely up in the air. However, let's imagine that we have polymorphic types, and a Vector type.

Let's look at a little more significant code, writing binary search over a vector.

bsearchr[start, finish, target:num, v:vector[num]] = { mid = elem(v, (start + finish)/2), if target == mid then True, else if start == finish then False, else if target > mid then $(mid, finish, target, v), else $(start, mid, target, v))}<-- here we'll inline the variable declarationsbsearch = bsearchr(0, len(v:vector[num]), target:num, v)<-- note that bsearch only takes 2 variables (the vector and the target)<-- alternative definition of bsearch:bsearch[v:vector[num], target:num] = { { mid = elem(v, (start:num + finish:num)/2), if target == mid then True, else if start == finish then False, else if target > mid then $(mid, finish), else $(start, mid)) }(0, len(v))}<-- Now we can use bsearch for some stuff.contains5 = bsearch(,5) <-- specifying the targetcontains5({1,2,3,4,5,6,7,8,9,10}) <-- trueinTen = bsearch({1,2,3,4,5,6,7,8,9,10}) <-- specifying the vectorinTen(6) <-- trueinTen(11) <-- false<-- note that the below is a meaningless function; it's just there <-- to illustrate combining two functions with an OR.usableVector = contains5 || vcontains <-- note, takes two variablesusableVector({1,2,3,4}, 5) <-- trueusableVector({1,2,3}, 12) <-- false<-- one interesting thing is that we could make the OR operator short-circuit, so that the following resolves to True even though it has unbound variables.usableVector({1,2,3,4,5}) <-- contains 5, so resolves to true.usableVector({1,2,3,4}) <-- is a function equivalent to vcontains<-- another example of thisf = (if i:num != i then j:num, else 1) <-- given any argument, f will always resolve to 1f(123) <-- 1

## Current Status

The first parser for Pico was written using Lex and Yacc, with an AST representation in C++. I later switched to Haskell, but for those interested, the C++ code can be viewed in the cpp directory -- although it might not reflect the latest syntax or evaluation schemes in Pico.

The current codebase for Pico can be found in the haskell directory. The parser is written using the Parsec library. There is also an evaluator written in Haskell which is able to correctly evaluate arithmetic expressions, variable assignment, and some degree of functions (for example, it can correctly compute factorials and fibonacci numbers). It can also correctly handle partial function application in the Haskell style (i.e. providing arguments from left to right), but not yet in the arbitrary style (where a new function can be created by specifying any subset of a function's variables). It supports treating functions as values (i.e. adding two functions to create a new function).

As of right now, only inline variable definitions are supported, and there are no data types besides primitives and functions.

## Running pico

As of right now, the best way to evaluate Pico code is using the eval' function from Eval.hs while in GHCi:

> cd haskell> ghciPrelude> :load Eval.hs Parser.hs AST.hs\*PicoEval> eval' "fact = {if n:num < 2 then 1 else n \* fact(n-1)}, fact(10)"((3628800.0, empty symtable),([<table>fact=>({if (n:NumT < 2.0) then 1.0 else (n \* fact((n - 1.0)))}, empty symtable)</table>],[]))\*PicoEval>

eval' takes a String of pico code and returns a value and a context, where the context is a tuple of a symbol table and a list of arguments. In short, the 3628800.0 is the value of note here.

## Future Directions

There is much work left to be done on Pico. The first priority is the correct evaluation of partial function application; next, I will add syntactic support for specifying function parameters out-of-line; next, I plan to add tuples and a list data type; after that, user-definable algebraic data types; finally, IO. IO in pico will not be handled purely, but there will be a modifier on functions which use IO so that they can only be called by other IO functions to encourage purity whenever possible. Further into the future, we may introduce concurrency primitives to Pico, most likely implementing a message-passing system.

My ultimate hope is that Pico become a simple, yet robust, functional, statically typed scripting language. There are several scripting languages which are functional; however, none of them enforce functional purity, immutable variables, or static typing, all of which Pico does (outside of IO). Combining the simplicity of a scripting language with the bug-reducing features above, I believe, will result in a very powerful language.

## Pico Grammar

Pico is designed to be a very small language with minimal syntax. Currently the entire grammar is here (this will be expanded slightly to include algebraic types, pattern matching and possibly interfaces)

pico -> expression '.' | pico expression '.'expression -> term | var '=' expression ',' expression | "if" expression "then" expression ',' "else" expressionterm -> literal | '(' expression ')' | invocation | term op term | unary\_op term | term ( '(' term (',' term)\* ')' )\*literal -> identifier type? | number | string\_literal | char\_literaltype -> ':' (identifier | '(' identifier (',' identifier)\* ')' )op -> '+' | '-' | '\*' | '/' | '%' | '>' | '<' | '&' | '|' | ">=" | "<=" | "==" | "!=" | "&&" | "||"unary\_op -> '-' | '!' | '~'identifier -> [a-zA-Z][a-zA-Z\_]\*number -> [0-9]\*(\.[0-9]+)?string -> \"(\\.|[^\\"])\*\"char -> \'(\\.|[^\\'])\*\'