Using Codswallop RPL

ian Butler

December 2023

Part I
Theory and Overview

1 What is RPL?

Codswallop RPL is based loosely on Hewlett-Packard's Reverse Polish Lisp, a language and operating system used in their 28 and 48 series calculators. RPL is a Forth-like language with Lisp characteristics such as robust support for lists and a bewildering quantity of delimiters, as well as strong typing and a number of data types. The Codswallop dialect is simplified in certain respects and improved in others; it retains the leisurely pace of execution of HP's implementation, particularly in its current form as written in Python.

Codswallop RPL is for amusement only. In this document, RPL refers specifically to this dialect and not HP's version, to which it may or may not apply.

1.1 Everything is an object

RPL is a collection of objects, first and foremost. They are not objects in the paradigm of having extendable, attached methods, but they are all capable of evaluating themselves, and they have some printable form. Stringing together objects is the fundamental way RPL programs are created.

Object Type	Example
Integer	#13
Float	13.37
String	“Hello, world.”
Quote	’FOO
Symbol	BAR
Comment	(This does nothing)
Built-in	+
Internal	<internal>
Handle	(I/O handle for foo.txt)
Tag	:foo: #13
Directory	[dir: :foo: #13]
List	{ #13 13.37 { ’FOO } }
Code	:: #2 #2 + DISP ;

1.1.1 Integers

Integers are represented as a number with a preceding pound sign. These can be freely mixed with floating-point numbers in math operations, and are typically used for counting and list indices. In a bare-metal implementation, their width would likely be equal to the word size of the machine; in the Python implementation, it’s whatever Python integers are.

1.1.2 Floats

Floating-point numbers are the default for any number the parser encounters. Currently they will cause an error if they’re used as a list index; it is possible a future implementation could try to coerce a float into an integer first.

1.1.3 Strings

Strings can contain anything other than a quotation mark, including newlines and whitespace and all manner of whatnottery. They cannot be properly represented if they contain a quotation mark, though enterprising individuals may certainly weasel such characters into a string to be used for display purposes. While not a composite object, some composite commands will work on strings, which can be used to slice and manipulate them.

1.1.4 Quote

An object beginning with an apostrophe is a quote. A quote will return its contents when evaluated, so objects such as symbols and code can be pushed to the stack to wait to be called upon. It’s used to reference an object in named storage or a code object without immediately evaluating it. Storing a symbol to itself is not allowed:

```
:: 'x 'x STO ;

You have died of dysentery.

in object 2 of :: 'x 'x STO ;

The complaint leveled against you by STO is as follows:
cDonalds Theorem does not apply to symbolic references
```

1.1.5 Symbol

An symbol is any name, typically starting with a letter or symbol. When encountered in a program, it is recalled from the named store and evaluated immediately.

1.1.6 Preprocessed object

The parser catches any unquoted symbol beginning with a grave (‘); it will attempt to recall such a symbol (sans grave). This is useful for including named constants in lists and code, and the parser will return an error if the symbol doesn't already exist.

1.1.7 Comment

A comment follows the same rules as a string, but delimited by parentheses instead of quotation marks. However, when evaluating itself, a comment disappears rather than landing on the stack. They are still objects, and take up space in lists and code.

1.1.8 Built-in

A built-in object is a command which behaves like a symbol. Built-ins may provide additional hints for their usage and arguments, and as regular objects in the named store, they can also be erased (at your peril.) They can dispatch differently according to the types of arguments expected.

1.1.9 Internal

Internals are the thinnest wrapper around a machine call (currently, a Python function call) which executes when evaluated. They are not readily accessible and are a good way to crash the interpreter.

1.1.10 Handle

A general I/O object generated when a file is opened, which can typically be read from or written to. Reads are subject to an internal limit, currently 256KB per single operation.

1.1.11 Tag

Tags are special mutable pairs, containing a simple symbol name (without subdirectories) and an arbitrary object. Tags can be used as references for argument passing, and for other purposes.

1.1.12 Directory

Directories are a special type of linked list which are the basis for named storage. Each directory entry contains a tag and a link to the next directory entry, if any. Generally, these entries aren't used directly, but a complete directory can be retrieved.

1.1.13 Lists

Lists are the fundamental plural data type in RPL. They can be of any size, and contain any object, including other lists. They can be fully manipulated: individual elements can be retrieved or replaced, objects can be appended to it, and subsets of the list can be extracted. Lists return themselves when evaluated. List indices begin at zero.

1.1.14 Code

Code is heavily based upon the list. Like lists, they can be fully manipulated and can contain any type of object. Instead of returning themselves when evaluated, they evaluate each of their objects in turn from start to finish. While there is no explicit flow control, there are built-ins and library objects which provide this functionality.

1.2 Objects live in several places

1.2.1 The stack

When an object is evaluated, it typically returns a copy of itself to the data stack. The stack is a special list to which any object can be pushed, and from which all functions take their arguments. It is shared universally and can grow to any size. For example, evaluating #2, and then evaluating #2 once more will cause the stack to contain both values:

```
{ #2 #2 }
```

With those two items on the stack, evaluating the built-in + will consume both items and return a result:

```
{ #4 }
```

The stack can be cleared, items duplicated, swapped around, stored, printed, and so on. It is the general scratch workspace for temporary storage of all manner of objects.

1.2.2 Named store

Any object which can be placed on the stack (that is, any object except the stack itself) can also be assigned a name, stored for later recall. This name may start with anything except a number or a delimiter, and may contain anything except a delimiter. For example, to store the integer 5 into a variable named x, one places the integer object on the stack, and the quoted symbol x, and executes the command STO:

```
:: #5 'x STO ;
```

Evaluating an object from the named store can be as simple as mentioning the object as an unquoted symbol.

1.2.3 Hierarchical named store

Each code object can also be assigned local variables. These names will cover any identical names further down the call stack, all the way to the global set of variables; they must be defined before the code is executed, and they disappear when evaluation of the code object is complete. More information on this behavior can be found in the section covering the LOCAL command.

1.2.4 Subdirectories

The basic building block of the named store is an object called a directory entry, a number of which form a linked list. Directory entries can also be created and stored by name, which can then have additional entries. These subdirectories can be stored to and recalled from with the familiar dot notation, for example:

```
:: MKDIR 'numbers STO #5 'numbers.five STO ;
```

will store an integer 5 to the name 'five', within a new subdirectory named 'numbers', where it can be recalled with 'numbers.five'. Additional things could be stored into the subdirectory similarly:

```
:: #6 'numbers.six STO ;
```

1.2.5 The ether

Code blocks currently being executed exist only in the ether (that is, on the call stack.) They are in memory, but are inaccessible except for the most current context, and cannot be modified; they disappear entirely along with their local variables when evaluation is complete. However, code can be freely modified before it is evaluated; you can see the section Building Code for more details.

Objects referenced by accessible lists, code, or directory entries (even entries no longer in the named store) also persist for as long as necessary.

1.3 First steps.

1.3.1 Starting the Python-based interpreter

The interpreter will run in Python 3. You can type

```
$ python3 rpl.py
```

to start the interpreter, which will bring you to the prompt:

```
Codswallop RPL
::
```

From here, you may type in any valid RPL code. Newlines are supported when typing directly into the interpreter by typing a backslash as the last character before pressing enter. The trailing semicolon is automatically appended. For example, when evaluating 2+2:

```
$ python3 rpl.py
Codswallop RPL
:: #2 #2 +
Stack: { #4 }
::
```

You may exit at any time by typing Ctrl-C or your operating system's end of file character (typically Ctrl-D or Ctrl-Z.) Typing Ctrl-C while code is executing will start a traceback and return to the REPL, if one was running. If your system has readline, you'll have familiar editing keys and a command history.

1.3.2 Common stack manipulation

It's very common to manipulate the stack in several ways. The most common are DUP and DROP; DUP will duplicate the first level of the stack, and DROP will remove the most recent item from it. You can also switch the two most recent items around with SWAP:

```
:: “Hello!” #2 DUP
Stack: { “Hello!” #2 #2 }
:: DROP
Stack: { “Hello!” #2 }
:: #3
Stack: { “Hello!” #2 #3 }
:: SWAP
Stack: { “Hello!” #3 #2 }
::
```

Finally, you can clear both the data and call stacks with CLR, which will leave the global named store untouched but will end any running program. This includes the code block CLR is in; anything after it will be gleefully ignored:

```
Stack: { "Hello!" #3 #2 }
:: CLR "Gentlemen!" #5 3.14
Stack: { }
::
```

1.3.3 Your first program

Of course your first program should be a friendly hello. You could type it and evaluate it immediately:

```
:: "Hello, world!" DISP
Hello, world!
Stack: { }
::
```

But of course, such savory appellations are best stored and enjoyed later, and again and again. So instead of typing a program to be evaluated, let us instead quote some code which will stay on the stack for a moment:

```
:: ':: "Hello, world!" DISP ;
Stack: { :: "Hello, world!" DISP ; }
::
```

And then we can put it in our named store under a friendly name, and just call it as we please:

```
:: 'hi! STO
Stack { }
:: hi!
Hello, world!
Stack { }
::
```

You can even store it to disk, so you can enjoy the greeting yet a second time:

```
:: 'hi! "hullo.rpl" >DSK
Stack: { }
:: ^C
$ python3 rpl.py
Codswallop RPL
:: "hullo.rpl" DSK> hi!
Hello, world!
Stack: { }
::
```

1.3.4 A little more stack play

Sometimes you may wish to recall an object from the named store without evaluating it, for example to bring up a code object without it running. RCL can be used for this purpose:

```
:: 'hi! RCL
Stack: { :: "Hello, world!" DISP ; }
::
```

Once you have a program on the stack, you can evaluate it explicitly using EVAL:

```
:: EVAL
Hello, world!
Stack: { }
::
```

This can be useful for many purposes, especially when code objects are stored in lists to be selected from and evaluated. EVAL will also recall and evaluate a quoted symbol, if it exists.

1.3.5 Dealing with errors

You will probably make a lot of mistakes. There are two main types of errors you are likely to encounter: the first is when entering text to be turned into objects. For example, if you mistyped your first program, you might end up with the following exchange:

```
:: "Hello, world!' DISP

Your words fail to become actions.
Stopped on line 1, position 4:
:: "Hello, world!' DISP ;
   ↑
In particular: Strings don't just start with quotes
Stack: { }
::
```

If the parser fails to make heads or tails of your input, it will make a solid effort to tell you exactly where it's failed. On the other hand, the interpreter itself will provide a comprehensive traceback of exactly where an error occurred, and why:

```
:: ':: x y + ; 'sum STO
Stack: { }
:: #3 'x STO sum DISP
```

```
You have died of dysentery.

in object 3 of :: #3 'x STO  sum  DISP ;
in object 1 of :: x  y  + ;

The complaint leveled against you by the symbol resolver is as follows:
We seek y but we cannot always find y
Stack: { #3 }
::
```

1.3.6 Exploring the named store

Other than this document, one way to look around in the interpreter to see what your options are is to use ?, which will display a list of the global symbols currently on offer. After you've found a name you're interested in, if it's a built-in, you can quote it and see what it has to say for itself with DOC:

```
:: 'STO DOC
Write an object into the named store, either to a
local variable if it exists, or to the main store.

STO is looking for 2 arguments.  For example:
Stack configuration 0:
  Line 2: Any
  Line 1: Symbol
```

1.4 Managing the flow

Because code is a list of objects evaluated in order, it's not instantly obvious how flow control might work. However, code is also an object, and can be stored on the stack, which broadens the possibilities significantly.

1.4.1 Function calls

1.4.2 If-then blocks

Consider this Python if-then statement about cakes:

```
lexcakes = 40
if lexcakes >= 40:
  print("That's terrible!")
  sadness = 1
```

Of course we can perform the assignment and the test for this statement easily in RPL:

```
:: #40 'lexcakes STO
Stack: { }
```

```
:: lexcakes #40 >=
Stack: { #1 }
```

Now that we've performed the test, instead of executing our sadness code outright, we can leave it on the stack in the form of a complete code block. By quoting it, it won't evaluate immediately:

```
:: ':: "That's terrible!" DISP #1 'sadness STO ;
Stack: { #1 :: "That's terrible!" DISP #1 'sadness STO ; }
```

With these two items in place, we can use the command IFT, which will consume both our test value and the code object, and evaluate the latter if the statement is true:

```
:: IFT
That's terrible!
Stack: { }
```

As with many things in RPL, the object to evaluated does not have to be of any particular type. In addition, a useful variant of this command is IFTE, if-then-else, which takes three arguments instead of two, and evaluates the second or third based upon the test value:

```
:: "That's " lexcakes #40 >=
Stack: { "That's " #1 }
:: "terrible!" "not so bad I suppose..." IFTE
Stack: { "That's " "terrible!" }
:: + DISP
That's terrible!
Stack: { }
```

1.4.3 CASE and KCASE

If you have a cascade of if-then-else blocks, a better way might be to use CASE or KCASE. The standard library contains these two versions of a case construct, which iterate instead of nesting. They both take a list in the following format:

```
{ { if-case then-case }
  { if-case then-case }
  ... }
```

CASE is exactly equivalent to a series of if-then statements; KCASE is similar, but accepts a constant which is memorized and pushed to the stack before each if-case is evaluated; it can also be recalled from either block as _*k*. The first if-case to return a nonzero result has its then-case evaluated, and no additional cases are matched. These functions support tail calls from then-case blocks. As demonstrated here, you can make an "else" case out of your last entry by making sure it always returns a nonzero result (or you can use ELSE for your final test case, which is a local variable which will do the same thing). Since the code here is in a list, it does not need to be quoted:

```
:: “That’s ” lexcakes
  { { :: 0 < ; “highly unlikely.” }
    { :: 3 <= ; “a completely reasonable number.” }
    { :: 39 < ; “a suspicious number of cakes.” }
    { :: DROP #1 ; “terrible!” } }
  KCASE + DISP ;
```

1.4.4 Looping

There is no atomic looping construct. The standard library does include an object called REP, which will evaluate a code object repeatedly until it returns zero. This can be used for any sort of “while” type loop. It can also be pressed into service as both an anonymous counter:

```
:: #1 ’:: DUP DISP #1 + DUP #3 <= ; REP
1
2
3
Stack: { #4 }
```

and a for-next style loop, using named storage:

```
:: #1 ’x STO
Stack: { }
:: ’:: x DISP x #1 + DUP ’x STO #3 <= ; REP
1
2
3
Stack: { }
:: ’x DISP
4
Stack: { }
```

1.4.5 Slowing it down

Because each code object is evaluated in order, and because the state of the stack is relevant to successive operations, it can be extremely helpful to single step through a program for debugging purposes. Single stepping can be started with the command SST, which will cause the interpreter to prompt after every command is executed, without clearing the call stack. SST can also be called from within a program to assist in debugging a problem section. Pressing enter repeatedly will cause successive objects to be evaluated, showing the stack state and the object that’s been evaluated each time.

```
:: #9 ’num STO “You have played ” num “ times” SST + DISP
Single step evaluated: SST
```

```
Call stack depth: 2
Stack: { }
Resume, break, shell, or enter to step:
Single step evaluated: +
Call stack depth: 2
Stack: { "You have played " "9 times" }
Resume, break, shell, or enter to step:
9 times
Single step evaluated: DISP
Call stack depth: 2
Stack: { "You have played " }
Resume, break, shell, or enter to step: r
Stack: { "You have played " }
::
```

An important feature of single stepping is being able to launch a shell at any point. This will evaluate REPL (if it exists), without clearing the call stack or local variables, so the whole machine state can be examined and modified. As soon as the REPL is dismissed (with ^D/^C or BAIL) the program continues. The break option ("b" or ^C) will stop execution with a traceback.

1.4.6 Light argument checking

Sometimes it's helpful, especially when debugging, to make sure a given block of code receives an appropriate number of arguments. By checking early in the function, you can isolate whether the problem is in the call or in the guts of your code. For this purpose, you can use REQUIRE instead of creating a built-in. For example, a function to add three things:

```
:: #3 REQUIRE + + ;
```

If your code fails at REQUIRE, you know it didn't receive the right number of arguments.

1.5 Working with lists

Lists are general-purpose containers into which any object or set of objects can be placed, and there are a few useful built-ins for dealing with them.

1.5.1 GET and PUT

You can retrieve a single item from a list, if it exists, or store a single item to a list (if the list is already an appropriate size) with GET and PUT:

```
:: { 1 2 3 "hello" } #3 GET
Stack: { "hello" }
:: { 1 2 3 4 }
```

```
Stack: { { 1.0 2.0 3.0 4.0 } }
:: “hello” #2 PUT
Stack: { { 1.0 2.0 “hello” 4.0 } }
::
```

You can also immediately evaluate an object out of a list rather than simply recalling it to the stack with GETE.

1.5.2 SUBS, LEFT, and RIGHT

To retrieve a subset of a list, place the list on the stack with starting and ending integer subscripts to receive your prize:

```
:: { 1 2 3 4 5 6 }
Stack: { { 1.0 2.0 3.0 4.0 5.0 6.0 } }
:: #2 #3 SUBS
Stack: { { 3.0 4.0 } }
::
```

All three functions work with strings as well as lists:

```
:: “left” #2 LEFT “right” #3 RIGHT +
Stack: “leght”
```

1.5.3 Adding and multiplying lists

You can append to a list with +, which will add a single object to a list:

```
:: { “butcher” “baker” }
Stack: { { “butcher” “baker” } }
:: “candlestick maker” +
Stack: { { “butcher” “baker” “candlestick maker” } }
::
```

And you can also multiply a list's contents by an integer:

```
:: { #1 #2 #3 }
Stack: { { #1 #2 #3 } }
:: #3 *
Stack: { { #1 #2 #3 #1 #2 #3 #1 #2 #3 } }
::
```

1.6 Descending into the deep

Since code is a type of data, some unexpected and interesting things can be done to it. It's also worth exploring how calls and returns are made internally, to see how optimizations can be made.

1.6.1 The mysterious call stack

The call stack is not viewable from within RPL, but the interpreter depends upon it. Each line of the call stack contains three items:

1. The code object being evaluated
2. An index pointing to the next object to evaluate
3. The first local directory entry

Except when single-step mode is active, the interpreter continuously evaluates objects from the most recent context, discards that context when it reaches the end of the code, and returns to the :: prompt when it has fully cleared the call stack. When a line is typed in at the :: prompt, it's parsed into a code object and pushed onto the call stack for evaluation in the same manner.

1.6.2 Introspection

It's possible to retrieve the currently running code, but it cannot be modified. This is sometimes useful for passing oneself as an argument.

```
:: ':: "Hello, world." DISP SELF ; EVAL
Hello, world.
Stack: { :: "Hello, world." DISP SELF ; }
::
```

SELF EVAL can be used to recurse; if it occurs at the end of a code block, it is effectively a jump back to the first object in the current code.

1.6.3 Tail call optimization

If you think about a program called minus that calls itself to count down to zero:

```
:: DUP ':: #1 - minus ; "Done!" IFTE ; 'minus STO
```

It would seem that the call stack would quickly be buried, as each :: #1 - minus ; block and each copy of minus would be appended as it recursed. However, each time a new code object is evaluated, it isn't blindly appended: if the current context is already at its last object, the new code replaces the old code on the current line. This optimization doesn't scrutinize too hard, though, so

```
:: DUP ':: #1 - minus ; "Done!" IFTE (Unoptimized!) ; 'minus STO
```

will quickly fill up the call stack, even though the only object left to evaluate is a comment. It is not strictly a recursion optimization; any call at the end of a code object is effectively a jump, albeit one which retains the current storage context (see below).

1.6.4 Named storage contexts

We've only looked so far at the global named store, but in fact, every time a new program is called, the first directory entry for the named store is attached to it, with local variables coming first. Whenever an object is stored or recalled, the interpreter starts by looking in the local store, and continues down through any further local stores until it reaches the global store. Local names can cover global names, or even local names from further-away contexts. STO will store to the nearest context that already has the name, and if it doesn't exist, it goes into the global store. Recalls will fail with an error if nobody has what they're looking for.

To make a local context, the objects to be stored are first placed upon the stack. A code object follows, and finally a list of names. The LOCAL command then creates a new context (or in the case of an optimized tail call, appends to the current context), assigning the objects remaining on the stack to the list of names by popping them one at a time. For example, to create a local context with a, b, and c containing the numbers 1, 2, and 3:

```
:: #1 #2 #3 ':: "It's easy as " a + b + c + DISP ; { c b a } LOCAL
It's easy as 123
Stack: { }
::
```

Local contexts are often useful for named argument passing. In addition to symbols, it's sometimes useful (for example, when local variables are to be used for counters or other purposes unrelated to arguments) to use tags instead of putting an object on the stack. Either method can be freely mixed in a local variable list:

```
:: #1 #3 ':: "It's easy as " a + b + c + DISP; { c :b:#2 a } LOCAL
It's easy as 123
Stack: { }
::
```

1.6.5 Dropping out of a context

A provision is made through the BAIL command to leave a context early. BAIL not only returns from the current context, but optimizes for returns much like calls are optimized. Therefore a BAIL at the end of an IFT or IFTE code object will not only leave its own context, but the one it's been called from (or more, if the IFT/IFTE is at the end of its own code block.)

```
:: #1 'BAIL IFT "Hello"
Stack: { }
:: #1 ':: BAIL ; IFT "Hello"
```

```
Stack: { }
:: :: #1 ':: BAIL ; IFT ; EVAL "Hello"
Stack: { }
:: #1 ':: BAIL (Unoptimized) ; IFT "Hello"
Stack: { "Hello" }
```

Similarly, BEVAL can be used to bail from the current context or contexts and begin a new one; if used in the middle of a code block, it is effectively a jump, similar to a tail call.

1.6.6 Building new code programmatically

Code is a special type of list, but it is a list, and you can build up blocks of code within a program before evaluating them. For example, if you wanted to roll a die of x sides, you might use a program like the following:

```
':: RND x * 1 + >INT ; 'dieroll STO
```

And if you wished to, for example, roll a die for even values of x from 6 to 10, you could produce a for-type loop:

```
:: #6 'x STO
  ':: RND x * 1 + >INT
    x #2 + DUP 'x STO #12 < ;
  REP ;
```

But what about two die rolls per loop? You could of course prepare an inner loop to count from 1 to 2, but you could also build your loop programatically (shown here as typed directly into the interpreter):

```
:: #6 'x STO
Stack: { }
:: ':: RND x * 1 + >INT ;
Stack: { :: RND x * 1 + >INT ; }
:: #2 *
Stack: { :: RND x * 1 + >INT RND x * 1 + >INT ; }
::
```

If you multiply code by an integer, it will produce a new program with an arbitrary number of copies of the original concatenated, just as with multiplying lists. While this is fine, it does not actually prepare our loop. But we can add the looping construct by using + to concatenate two programs:

```
:: ':: x #2 + DUP 'x' STO #12 < ; +
Stack: { :: RND x * 1 + >INT RND x * 1 + >INT x #2 + DUP 'x' STO #12 < ; }
```

And in this case use REP to execute our synthesized code.

1.7 Working with text files

It's possible to read, write, and append to text files with Cods. You can open a file with FOPEN, which returns an I/O handle object representing it:

```
:: "salaries.csv" "read" FOPEN
Stack: { (I/O handle for salaries.csv) }
:: 'sal STO
Stack: { }
```

From here you can use READ or READL to read in a fixed-length string or a whole line, respectively:

```
:: sal READL
Stack: { "Big Boss,12500000,pesetas,George Town,Cayman Islands" }
```

And you can use EOF to check if you're at the end of a file. Writing a string is similar to STO, with WRITE recoridng an entire line and WRITEN doing the same without a newline:

```
:: "salaries.csv" "append" FOPEN 'sal STO
Stack: { }
:: "Shift Supervisor,1250,cents,Saginaw,Michigan" sal WRITE
Stack: { }
:: sal FCLOSE
```

To open a file, read it in entirety and close it again in a relatively atomic operation, you can use READF, which accepts a filename and returns a list of lines within the file. And while not file-specific, to fit with our example, it's possible to split up a string by its delimiters to help with parsing:

```
:: "Shift Supervisor,1250,cents,Saginaw,Michigan" "," SPLIT
Stack: { { "Shift Supervisor" "1250" "cents" "Saginaw" "Michigan" } }
```

It's a good idea to close a file with FCLOSE when you're done with it. Files are safely closed implicitly even without using FCLOSE, but at a time and place of the machine's choosing.

1.8 A study of types

Cods can have any number of object types, and there's no guarantee the type number returned by TYPE will be exactly the same from session to session, though type number won't change during a session. The Types directory contains information about which types are available. Each type's number is stored in Types.[type]. For example, Types.Integer might contain #10. There is also a list, Types.n, to do the reverse; in this example, Types.n #10 GET will return "Integer". Prototypical objects for user types are in Types.Proto, when used.

1.8.1 Registering a new type

Any object can be used as a prototype for a user-defined type. When registered with >TYPE, the prototype will be turned into a new object type with its own name and type number. For example, to make a counter which contains an integer, you could register a new Counter type which contains one. The only additional information needed is an internal name for the data. "self" is a good choice:

```
:: :self: #1 'Counter >TYPE
```

Registering a new type will make its type number available as Types.Counter and its name available in Types.n, just as all other objects. The prototype object is stored in Types.Proto.Counter, with the default value. To make a new object of a user type, it is generally sufficient to make a copy of the prototype object. By default, a user type will appear as a tag:

```
:: (Make a new counter.) Types.Proto.Counter CP
Stack: { :self: #1 }
:: 'ourcount STO
Stack: { }
```

Even though our new user type here contains only an integer, it is now its own type with full rights and privileges, and thus built-ins will not treat it as an integer. In fact, at the moment, very few functions will interact with it unless they accept any type of object. What our new object needs is some methods.

1.8.2 Making a method

User type objects have access to METH, which will evaluate a block of method code in a special environment. The object is stored as a local variable with its internal name, and its data can be accessed directly and updated. Anything stored back to it will be retained. For example, to increment our counter by 5:

```
:: ourcount ':: self #5 + 'self STO ; METH
Stack: { }
:: (Both copies are different...) ourcount Types.Proto.Counter
Stack: { :self: #6 :self: #1 }
```

Of course, as shown this technique will permit a method to be evaluated against any user type. What we could do instead is make a built-in which accepts our type.

1.8.3 Making a built-in

Built-ins are defined by the number of required arguments, a hint string describing what the command will do, and its name. The name doesn't have to match what the built-in is stored as, but it is what will claim responsibility in case of errors. a There is also a list of ways to call referred to as the dispatch table. For the purposes of this example, we can start by creating a built-in with an empty dispatch table. Of course a full table can be included here just as readily:

```
:: {} #2 “Increment a counter by a positive integer.” ’inc >BIN
Stack: { inc }
:: ’inc STO
Stack: { }
```

As stored here, inc will always produce an error: if there isn’t one item on the stack, it will complain of too few arguments, and if there is an item on the stack, it will proclaim the argument type is incorrect. But additional dispatch lines can be added to this or any built-in after the fact, and so from here, we can make a fully type checked inc which will only work on Counter types.

1.8.4 Getting HOOK on METH

Every line of a built-in’s dispatch table consists of an object to evaluate if it matches, followed by zero or more argument types according to how many the built-in has been told to expect (in our case, two: our counter, and an increment value.) HOOK can be used to add lines to the beginning of an extant built-in’s dispatch table. The code to do our increment in this example will look something like:

```
::
  DUP #0 >
  ’:: SWAP ’:: self + ’self STO ; METH ;
  ’:: “Increment means ’goes up’, dearheart” DED ;
  IFTE ;
```

Maybe we thought long and hard about that routine, and carefully wrote it and left it on the stack. That’s all right; we can stash it into a local variable and make a dispatch table that way. While we’re at it, let’s patch up DISP to show a nice version of our counter instead of the clumsy looking tag. HOOK will recall any symbols in the dispatch table, so we can make a nice looking one that will reference all the numeric types and our fancy thing by name:

```
:: :: { { ourfancything Types.Counter Types.Integer } }\
..? ’inc RCL HOOK ; { ourfancything } LOCAL
Stack: { }
:: { { :: ’:: \
..? “This counter reads: “ self + DISP ; METH ; Types.Counter } } \
..? ’DISP RCL HOOK
Stack: { }
```

And from here, why not try them both? Let’s bring our counter up to a nice round number and print it:

```
:: ourcount #10 inc ourcount DISP
This counter reads: 16
Stack: { }
```

Neither HOOK nor METH attempt to save an adventurous programmer from his or her efforts to create infinite recursions (for example, if that DISP code attempted to call DISP with a Counter instead of a string, it would call itself forever). Such a condition will cause the running code to hang, but you can return to the REPL with ^C.

1.8.5 The documentation is automatic

It's worth noting that because DOC's output is generated upon request, the new hooks and even the new type name appear immediately:

```
:: 'DISP DOC
DISP is a symbol, following it...
What we have here is a builtin which calls itself DISP:

Print any object in human-readable form to the screen.

It takes 1 argument, and there are 2 ways to call it.

Stack configuration 0:
  Line 1: Counter
Stack configuration 1:
  Line 1: Any
```

1.8.6 Just two more things

If you need to pass the entire object along from within a method, you can quote and dereference it. From our example:

```
:: ourcounter ':: (Increment by one and return ourselves) self #1 \
..? + 'self STO 'self DEREF ; METH
Stack: { :self: #17 }
```

If your type contains a directory, it will have to be copied separately. You can include this in your constructor function:

```
::
  Types.Proto.Counter CP
  ':: self CP 'self STO 'self DEREF ; METH ; ;
```

This is because making a memory copy of a tag makes a new copy of the tag, pointing to the original object.

1.9 Faster execution with the source-source compiler

Normally when code is evaluated, all symbols are looked up and evaluated at runtime. This provides maximum flexibility for reusing code, as symbols like DISP or STO could have their behavior changed by storing those names as local variables. However, for improved speed, it's useful to reduce the number of symbol lookups and to reduce the amount of time each lookup takes.

For the latter, it's well to remember that the interpreter searches for local variables first; the closer the name is to the beginning, the faster it will be found. Conversely, global variables are always appended to the very end of the store, and thus will always be the slowest to find.

For the former, one way to reduce symbol lookups is to use STATIC or STATICN, a source-source compiler included with the language. They take any object and recursively delve into it, looking for symbols which currently resolve to a built-in or internal object. Where such symbols are found, they're replaced with a direct reference to the builtin or internal, thus saving a lookup each time that object is evaluated in the future. The compiler will return either the original object if no changes were made, or a new, compiled object.

In most situations, STATICN will suffice: it takes only one argument, which would typically be code but can be any type of object. If it's important to leave a symbol unresolved, STATIC additionally accepts a list of exceptions.

While entire directories can be compiled, there is less chance of excitement if the directory is first removed from the store:

```
:: Projectdir 'Projectdir RM STATICN 'Projectdir STO ;
```

The compiler recurses into tags, lists, code, quotes, and directories looking for symbols. Once a symbol is found, it recurses into that symbol if it exists in the named store, and continues to do so until finding a destination object. If the destination is a built-in or internal, the symbol is replaced. It will recurse to its depth limit, which is usually found in Static.default.depth; any deeper recursion is silently ignored.

It's possible to extend the compiler to recurse into user-defined types, but they are safely ignored by default.

Part II
Function reference

==, != Check two objects for equality or inequality. Equality is not restricted by type; congruent objects are matched, such as a symbol 'hello.there' and a function 'hello.there'. Directories, tags, code, or lists are equal only if they are the exact same object:

2.	Any
1.	Any

⇒

1.	Integer	Boolean result

<, <=, >, >= Compare two numbers:

2.	Number
1.	Number

⇒

1.	Integer	Boolean result

+ Arithmetic addition:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

$n.$	Integer
$n.$	Float

⇒

1.	Float

Or concatenate an object with a string:

$n.$	String
$n.$	Any

⇒

1.	String

Or add an item to a list:

2.	List
1.	Any

⇒

1.	List

Or merge two blocks of code:

2.	Code
1.	Code

⇒

1.	Code

- Arithmetic subtraction:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

$n.$	Integer
$n.$	Float

⇒

1.	Float

***** Arithmetic multiplication:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

$n.$	Integer
$n.$	Float

⇒

1.	Float

Or duplicate a string, list, or code:

2.	String, list, code
1.	Integer

⇒

1.	String, list, code * n

/ Arithmetic division:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

$n.$	Integer
$n.$	Float

⇒

1.	Float

^ Arithmetic power:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

$n.$	Integer
$n.$	Float

⇒

1.	Float

ABS Absolute value:

1.	Number

⇒

1.	|Number|

AND Logical AND, returning #1 if both numbers are nonzero:

2.	Number
1.	Number

⇒

1.	Integer

>**ASC** Return the character of a valid Unicode number:

1.	Integer	Character number

⇒

1.	String

ASC> Return the character number of the first character of a string:

1.	String

⇒

1.	Integer	Character number

BAIL Return immediately from a code block, to the nearest code block whose execution is not yet complete.

BEVAL BAIL and then EVAL. Stack requirements and results are identical to EVAL.

>**BIN** Prepare a built-in object. The dispatch table is built as follows:

```
{ { dispatch-object object-type (object-type...) }
  or
  { dispatch-symbol type-symbol (type-symbol...) }
  ... }
```

where the code to execute is the first object of each sub-list, and the remaining values are the integer types for the arguments on the stack (or #0 if any type is acceptable.) When called, a built-in tests the stack arguments and dispatches to the first matching line of this table. Any objects after the argument types are ignored, and can be used for comments. If the dispatch object is a symbol, it is recalled first. Symbols evaluating to valid integers are also permitted for object-type fields, so for example, the symbol Types.String is an acceptable object-type.

The name does not have to match the name the resulting object is stored into, nor does the object have to be stored to be evaluated; however, if an argument-checking error occurs, it will be attributed to the name given here, and it's the name shown when this object is displayed.

4.	List	Dispatch table		
3.	Integer	Argument count	⇒	1. Built-in
2.	String	Documentation		
1.	Quoted symbol	Name		

BRKSST By default, ^C will start a traceback. For debugging purposes, it can be useful to have it enter single step mode instead.

1.	Integer	#1	⇒ *single step on break*
1.	Integer	#0	⇒ *trace back on break*

CASE Effectively a nested IFTE, CASE evaluates test objects in order and then evaluates the first matching result object. It takes a list of the form:

```
{ { if-case then-case }
  { if-case then-case }
  ...
  { ELSE else-case } }
```

1.	List	CASE block	⇒

See also the slightly modified KCASE, which accepts and supplies a constant test value.

CLR Clear both the data and call stacks and return to the first REPL, if any.

CP Make a new copy of a mutable type, for example creating a duplicate user type, directory tree, tag, or a built-in. Tag contents are not copied, so if they contain a mutable type themselves (such as a directory) that will have to be copied separately:

1.	Source object	⇒	1.	Copy

DED Invoke the interpreter's error handler:

1.	String	Error message	⇒ *error*

DEDCONT Control the behavior of the interpreter's error handler.

1.	Integer	#1	⇒ *continue execution on error*
1.	Integer	#0	⇒ *trace back on error*

All normal runtime errors, including ^C breaks, are suppressed if continuation is enabled. Errors can be checked with ISDED, and parse errors will still be printed. Reasonable persons will leave this setting in its default state.

DEREF Return the tag (reference) for the closest name:

1.	Symbol	Name	⇒	1.	Tag

DIR Return a list of symbol names from a subdirectory. (see also: NAMES)

		$\Rightarrow$			
1.	Directory entry	$\Rightarrow$	1.	List	Names as unquoted symbols

DISP Display the printable form of any object:

1.	Any	Object to display	$\Rightarrow$

DISPN Display the printable form of any object, but suppress a newline:

1.	Any	Object to display	$\Rightarrow$

DOC Display information about a built-in, such as hints and information about valid stack frames. A symbol is also acceptable:

1.	Symbol, Built-in	Object of interest	$\Rightarrow$

DROP Pop the first line from the stack and discard:

1.	Any	Source object	$\Rightarrow$

>DSK Recall a symbol and record to disk, such that a subsequent DSK> will store the object back to that name:

2.	Symbol	Object name	$\Rightarrow$
1.	String	Destination filename	

Or record any other object type to disk:

2.	Any	Source object	$\Rightarrow$
1.	String	Destination filename	

Note also:

- Only the printable form of an object is recorded, so direct memory references are lost.
- Built-ins are backquoted (e.g., + becomes ‘+) to preserve some of the effects of STATIC.
- Strings containing quotes are not currently stored correctly.
- Margin and indentation settings are pulled from ANSI.default.
- If the file exists, it is appended to rather than overwritten.

DSK> Read a file from disk and pass it to the parser. This is equivalent to typing directly into the REPL, except that line breaks are allowed and any amount of data up to the interpreter's limit (currently 256KiB) can be read.

1.	String	Source filename	$\Rightarrow$

DUP Duplicate the first line of the stack:

			$\Rightarrow$	2.	Any	Source object
1.	Any	Source object		1.	Any	Source object

DUP2 Duplicate the first two lines of the stack:

2.	Any	Source object 2
1.	Any	Source object 1

⇒

4.	Any	Source object 2
3.	Any	Source object 1
2.	Any	Source object 2
1.	Any	Source object 1

EOF Determine if a file opened for reading is at the end of file. EOF will always return #0 for files opened for write or append.

1.	I/O handle	

⇒

1.	Integer	Boolean result

EPOCH Return current UNIX time (seconds since January 1, 1970, 00:00 UTC):

⇒

1.	Integer	Current epoch time

EVAL Evaluate an object:

1.	Code, built-in, internal	Source object

⇒ execute code

1.	Symbol	Source object

⇒ recall symbol and evaluate

1.	Quote	Source object

⇒

1.	Any	Unquoted object

1.	Comment	Source object

⇒

1.	Any other	Source object

⇒

1.	Any	Source object

EXISTS Determine whether a symbol exists in the named store:

1.	Symbol	Name

⇒

1.	Integer	Boolean result

>FLOAT Convert an object to a float:

1.	Integer, float, string

⇒

1.	Float

FCLOSE Explicitly close a file:

1.	I/O handle

⇒

FOPEN Open a text file. The mode may be "read", "write", or "append":

2.	String	Filename
1.	String	Mode

⇒

1.	I/O handle

FOREACH For each object in code or a list on line 2, supply that object on the stack and evaluate against the object on line 1. Objects within the list can be updated using the local symbol 'update'. Be aware that if the evaluator calls LOCAL, it should not do so as a tail call:

2.	List, code	Source list
1.	Any	Evaluator

⇒

1.	Destination list

GET Fetch an object from code or a list and return it to the stack:

2.	List, code	Source object	⇒	1.	Any	Fetched object
1.	Integer	Index (from zero)				

Or fetch a single character from a string:

2.	String	Source object	⇒	1.	String	Fetched character
1.	Integer	Index (from zero)				

GETE Fetch an object from code or a list and evaluate it:

2.	List, code	Source object	⇒
1.	Integer	Index (from zero)	

HAS Check to see if a list contains something like an item, using the same rules as ==:

2.	List		⇒	1.	Integer	Boolean result
1.	Any	Item to find				

HOOK Modify a built-in object by adding new lines to its dispatch table. The table structure is identical to that used by >BIN, and is prepended to the table already in use. The built-in is modified in place rather than issuing a new object; see CP if you wish to save a copy of the original object.

2.	List	Dispatch table
1.	Built-in	Object to modify

IFT Conditionally evaluate an object. If line 2 is true (typically if it's a nonzero number), evaluate line 1, otherwise can it.

2.	Any	Condition	⇒
1.	Any	Object to evaluate if true	

IFTE Conditionally evaluate an object. If line 3 is true (typically if it's a nonzero number), evaluate line 2, otherwise evaluate line 1.

3.	Any	Condition	
2.	Any	Object to evaluate if true	⇒
1.	Any	Object to evaluate if false	

INT Convert an object to an integer:

1.	Number, string	⇒	1.	Integer

IP Return the integer portion of a number:

1.	Number	⇒	1.	Float

ISDED Return information about the interpreter's error state. This is only meaningful in concert with DEDCONT and UNDED. If an untrapped error has occurred, the error state is true:

⇒

1.	Integer	Boolean error state (#1)

If no error has occurred, no string is returned:

⇒

1.	Integer	Boolean error state (#0)

KCASE Constant CASE. For each test case, a constant is supplied; it's also available within the test case as the local variable _k:

2.	Any	Constant object
1.	List	CASE block (see CASE)

⇒

LEN Return the length of an object:

1.	String, list, code	Source object

⇒

1.	Integer	Length

Or return the depth of a symbol, that is, how many individual names it contains:

1.	Symbol, function	Source object

⇒

1.	Integer	Depth

LEFT Return entries from the beginning of a string or composite object (see also RIGHT, SUBS):

2.	String, list, code	Source object
1.	Integer	Number of entries

⇒

1.	String, list, code	Subset

LOCAL Execute code with local variables:

n.	Any	Local variable n
3.	Any	Local variable 1
2.	Code	Local environment
1.	List	Local variable names

⇒ execute line 2

Each name is stored in order, so the list takes the form:

```
{ variable-1 ... variable-n }
```

Tags can also be included anywhere in the list, and will store the name and associated value without pulling from the stack. Be aware: tail call optimization can cause local variables to unexpectedly linger, depending upon where LOCAL is called in relation to the end of a code block. To force a new context for a LOCAL at the end of a code block, placing a comment after LOCAL will suffice.

>**LST** Create a list from items on the stack:

n.	Any	First object
2.	Any	Last object
1.	Integer	Number of entries

⇒

1.	List

METH Evaluate a block of method code in a special local context, where the contents of a user type are accessible as a local variable:

2.	User type	Source object
1.	Code	Local environment

⇒

MKDIR Create a new, empty directory:

⇒

1.	Directory

MOD Arithmetic modulo:

2.	Integer
1.	Integer

⇒

1.	Integer

or if one is a float:

n.	Integer
n.	Float

⇒

1.	Float

NAMES Return a list of currently stored symbol names, including local variables:

⇒

1.	List	Available names as unquoted symbols

NEG Negate a number:

1.	Integer, float

⇒

1.	0 - integer, float

NOT Logical NOT, returning #1 if number is zero and #0 if it's nonzero:

1.	Number

⇒

1.	Integer

>**OBJ** Pass a string to the parser:

1.	String	Source text

⇒

n.	Any	First parsed object
1.	Any	Last parsed object

OBJ> Break apart a composite object and return its contents to the stack:

1.	Code, list

⇒

n.	Any	First object
2.	Any	Last object
1.	Integer	Object count

Or return the innards of a tag:

1.	Tag

⇒

2.	Any	Stored object
1.	Symbol	Tag name

Or return the innards of a built-in:

1.	Built-in

⇒

4.	List	Dispatch table
3.	Integer	Argument count
2.	String	Hint text
1.	Symbol	Name

ODD Return #1 if a number is odd:

1.	Number

⇒

1.	Integer

OR Logical OR, returning #1 if either number is nonzero:

2.	Number
1.	Number

⇒

1.	Integer

POP Pop the last object off a list:

1.	List, code

⇒

2.	List minus popped object
1.	Object

PROMPT Read a line from the console:

1.	String	Prompt string

⇒

1.	String	User-entered text

PUT Store an object to code or a list:

3.	List, code	Destination
2.	Any	Object
1.	Integer	Index (from zero)

⇒

1.	List, code

QUOTE Quote an object:

1.	Any

⇒

1.	Quote

RCL Recall an object to the stack from the named store:

1.	Symbol	Object name

⇒

1.	Any	Object

Or recall an object from a tag:

1.	Tag

⇒

1.	Any	Object

READ Read some number of bytes from a file. If the number is less than 1, read til the end of file or the interpreter's limit.

2.	I/O handle	File
1.	Integer	Number of characters to read

⇒

1.	String

READF Read an entire text file and return a list of its constituent lines (each line must be less than the interpreter's limit):

1.	String	Filename

⇒

1.	List

READL Read one line from a file, up to the interpreter's limit:

1.	I/O handle

⇒

1.	String

REP Repeatedly evaluate code until it returns zero:

1.	Code

⇒

REQUIRE Check to make sure enough objects are on the stack:

1.	Integer	Required stack size

⇒

RIGHT Return entries from the end of a string or composite object (see also LEFT, SUBS):

2.	String, list, code	Source object
1.	Integer	Number of entries

⇒

1.	String, list, code	Subset

RM Remove a symbol from the named store:

1.	Symbol	Object name

⇒

RND Return a random number between 0 and 1:

⇒

1.	Float	Random number

ROLL Rotate n lines of the stack upward. For example, 1 2 3 #3 ROLL:

4.	Any	*(1)*
3.	Any	*(2)*
2.	Any	*(3)*
1.	Integer	Stack lines *(#3)*

⇒

3.	Any	*(2)*
2.	Any	*(3)*
1.	Any	*(1)*

ROLLD Rotate n lines of the stack downward. For example, 1 2 3 #3 ROLLD:

4.	Any	*(1)*
3.	Any	*(2)*
2.	Any	*(3)*
1.	Integer	Stack lines *(#3)*

⇒

3.	Any	*(3)*
2.	Any	*(1)*
1.	Any	*(2)*

SELF Return the currently-running code object:

⇒

1.	Code

SPLIT Split a source string into zero or more substrings around a given single character delimiter, such as a comma:

2.	String	Source string
1.	String	Delimiter

⇒

1.	List	Substrings

SST Begin single stepping debug mode.

SSTOFF End single stepping debug mode.

STACK Return the entire contents of the stack:

⇒

1.	List	Stack contents

STATIC Recursively rummage through an object and scrutinize all symbols found therein. If a symbol currently resolves to a built-in or internal, it is replaced with a direct reference, which prevents local names from pre-empting it but greatly speeds execution. Any symbols in the exclusion list remain untouched:

2.	Any	Source object	⇒	1.	Compiled object
1.	List	Symbol exclusion list			

STATICN Like STATIC, but with a null exclusion list:

1.	Any	Source object	⇒	1.	Compiled object

STO Store an object into the named store:

2.	Any	Object	⇒
1.	Quoted symbol	Name	

Or store an object into a tag:

2.	Any	Object	⇒
1.	Tag		

>STR Return the string representation of an object. This is the corollary to >OBJ, and like >DSK, is limited in that things like directory entries and built-ins revert to symbols when parsed again:

1.	Any	Object	⇒	1.	String	Printable representation

SUBS Return a subset of a string or composite object (see also RIGHT, LEFT):

3.	String, list, code	Source object				
2.	Integer	Lower subscript	⇒	1.	String, list, code	Subset
1.	Integer	Higher subscript				

SWAP Exchange two lines of the stack:

2.	Any	Source object 2	⇒	2.	Any	Source object 1
1.	Any	Source object 1		1.	Any	Source object 2

>SYM Convert a string into a symbol:

1.	String	⇒	1.	Symbol

>TAG Create a new tag from an object any a symbol. The symbol cannot contain a period:

Or return the innards of a tag:

2.	Any	Source object	⇒	1.	Tag
1.	Symbol	Tag name			

TRACE Set the number of running code blocks to leave on the call stack in case of an error. Normally this is set to 1 in an interactive session, so the furthest a traceback can unwind is to return to the REPL. If a program is loaded from the commandline, it's normally set to 0 to trace back all the way and exit.

1.	Integer	Traceback stopping point	⇒

TYPE Return the type number of an object:

1.	Any	Object	⇒	1.	Integer	Type number

>TYPE Register a new data type. The prototypical source object and internal name are combined into a tag, which is then modified with a new type number and type name. A type name must be a valid, simple symbol not already in use. After registering a new type, the prototype is available as Types.Proto.Typename, and new copies can be created with CP. Methods can then act upon the new object (see METH):

2.	Tag	Internal name and data	⇒
1.	Symbol	Type name	

UNDED Clear the interpreter's error state. This is only meaningful in concert with DEDCONT.

VAL Return the number represented by a string, and zero if the string can't be parsed:

1.	String	⇒	1.	Float	Parsed value

WRITE Write a line to a file:

2.	String	Text to write	⇒
1.	I/O handle	File	

WRITEN Write a line to a file, but do not insert a newline:

2.	String	Text to write	⇒
1.	I/O handle	File	