Branch: **master** Find file Copy path

mal / process / **guide.md**

\m/ **mANIApHOBIC** Correct a trivial typographical error in the guide

8c0163f on Mar 28

30 contributors

and others

Raw Blame History

1695 lines (1405 sloc) 75.2 KB

The Make-A-Lisp Process

So you want to write a Lisp interpreter? Welcome!

The goal of the Make-A-Lisp project is to make it easy to write your own Lisp interpreter without sacrificing those many "Aha!" moments that come from ascending the McCarthy mountain. When you reach the peak of this particular mountain, you will have an interpreter for the mal Lisp language that is powerful enough to be self-hosting, meaning it will be able to run a mal interpreter written in mal itself.

So jump right in (er ... start the climb)!

Pick a language

You might already have a language in mind that you want to use. Technically speaking, mal can be implemented in any sufficiently complete programming language (i.e. Turing complete), however, there are a few language features that can make the task MUCH easier. Here are some of them in rough order of importance:

- A sequential compound data structure (e.g. arrays, lists, vectors, etc)
- An associative compound data structure (e.g. a dictionary, hash-map, associative array, etc)
- Function references (first class functions, function pointers, etc)
- Real exception handling (try/catch, raise, throw, etc)
- Variable argument functions (variadic, var args, splats, apply, etc)
- Function closures
- PCRE regular expressions

In addition, the following will make your task especially easy:

- Dynamic typing / boxed types (specifically, the ability to store different data types in the sequential and associative structures and the language keeps track of the type for you)

- Compound data types support arbitrary runtime "hidden" data (metadata, metatables, dynamic fields attributes)

Here are some examples of languages that have all of the above features: JavaScript, Ruby, Python, Lua, R, Clojure.

Michael Fogus has some great blog posts on interesting but less well known languages and many of the languages on his lists do not yet have any mal implementations:

- http://blog.fogus.me/2011/08/14/perlis-languages/
- http://blog.fogus.me/2011/10/18/programming-language-development-the-past-5-years/

Many of the most popular languages already have Mal implementations. However, this should not discourage you from creating your own implementation in a language that already has one. However, if you go this route, I suggest you avoid referring to the existing implementations (i.e. "cheating") to maximize your learning experience instead of just borrowing mine. On the other hand, if your goal is to add new implementations to mal as efficiently as possible, then you SHOULD find the most similar target language implementation and refer to it frequently.

If you want a list of programming languages with an approximate measure of popularity try the RedMonk Programming Language Rankings or the GitHut 2.0 Project.

Getting started

- Install your chosen language interpreter/compiler, language package manager and build tools (if applicable)
- Fork the mal repository on github and then clone your forked repository:

```
git clone git@github.com:YOUR_NAME/mal.git
cd mal
```

- Make a new directory for your implementation. For example, if your language is called "quux":

```
mkdir quux
```

- Modify the top level Makefile to allow the tests to be run against your implementation. For example, if your language is named "quux" and uses "qx" as the file extension, then make the following 3 modifications to Makefile:

```
IMPLS = ... quux ...
...
quux_STEP_TO_PROG = mylang/$($(1)).qx
```

- Add a "run" script to your implementation directory that listens to the "STEP" environment variable for the implementation step to run and defaults to "stepA_mal". Make sure the run script has the executable file permission set (or else the test runner might fail with a permission denied error message). The following are examples of "run" scripts for a compiled language and an interpreted language (where the interpreter is named "quux"):

```
#!/bin/bash
exec $(dirname $0)/${STEP:-stepA_mal} "${@}"
```

```
#!/bin/bash
exec quux $(dirname $0)/${STEP:-stepA_mal}.qx "${@}"
```

This allows you to run tests against your implementation like this:

```
make "test^quux^stepX"
```

If your implementation language is a compiled language, then you should also add a Makefile at the top level of your implementation directory. This Makefile will define how to build the files pointed to by the quux_STEP_TO_PROG macro. The top-level Makefile will attempt to build those targets before running tests. If it is a scripting language/uncompiled, then no Makefile is necessary because quux_STEP_TO_PROG will point to a source file that already exists and does not need to be compiled/built.

General hints

Stackoverflow and Google are your best friends. Modern polyglot developers do not memorize dozens of programming languages. Instead, they learn the peculiar terminology used with each language and then use this to search for their answers.

Here are some other resources where multiple languages are compared/described:

- http://learnxinyminutes.com/
- http://hyperpolyglot.org/
- http://rosettacode.org/
- http://rigaux.org/language-study/syntax-across-languages/

Do not let yourself be bogged down by specific problems. While the make-a-lisp process is structured as a series of steps, the reality is that building a lisp interpreter is more like a branching tree. If you get stuck on tail call optimization, or hash-maps, move on to other things. You will often have a stroke of inspiration for a problem as you work through other functionality. I have tried to structure this guide and the tests to make clear which things can be deferred until later.

An aside on deferrable/optional bits: when you run the tests for a given step, the last tests are often marked with an "optional" header. This indicates that these are tests for functionality that is not critical to finish a basic mal implementation. Many of the steps in this process guide have a "Deferrable" section, however, it is not quite the same meaning. Those sections include the functionality that is marked as optional in the tests, but they also include functionality that becomes mandatory at a later step. In other words, this is a "make your own Lisp adventure".

Use test driven development. Each step of the make-a-lisp process has a bunch of tests associated with it and there is an easy script to run all the tests for a specific step in the process. Pick a failing test, fix it, repeat until all the tests for that step pass.

Reference Code

The `process` directory contains abbreviated pseudocode and architecture diagrams for each step of the make-a-lisp process. Use a textual diff/comparison tool to compare the previous pseudocode step with the one you are working on. The architecture diagram images have changes from the previous step highlighted in red. There is also a concise cheatsheet that summarizes the key changes at each step.

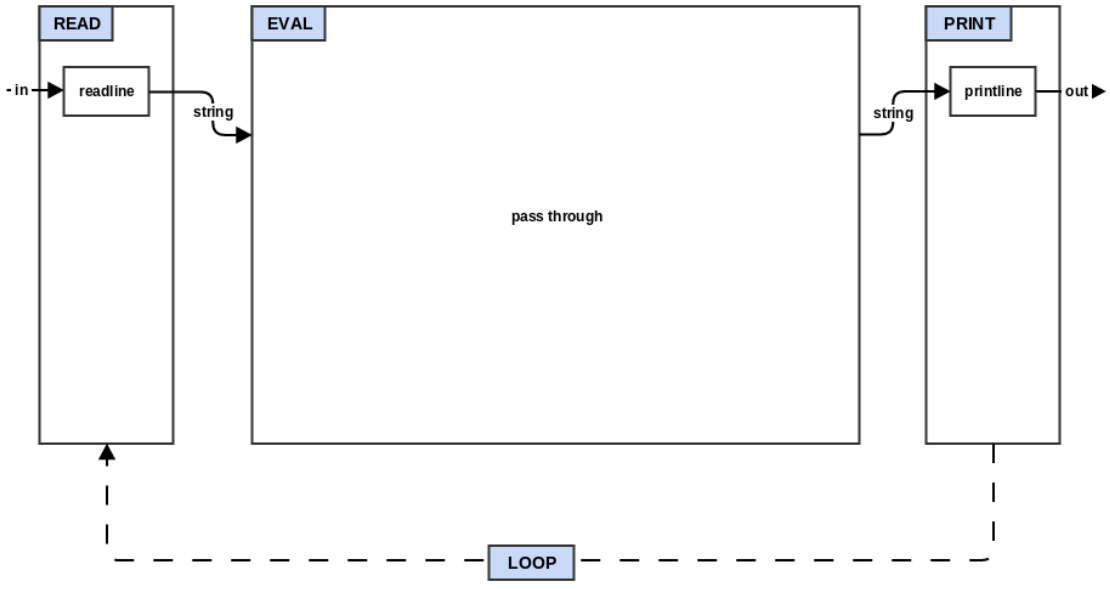
If you get completely stuck and are feeling like giving up, then you should "cheat" by referring to the same step or functionality in a existing implementation language. You are here to learn, not to take a test, so do not feel bad about it. Okay, you should feel a little bit bad about it.

The Make-A-Lisp Process

Feel free to follow the guide as literally or as loosely as you like. You are here to learn; wandering off the beaten path may be the way you learn best. However, each step builds on the previous steps, so if you are new to Lisp or new to your implementation language then you may want to stick more closely to the guide your first time through to avoid frustration at later steps.

In the steps that follow the name of the target language is "quux" and the file extension for that language is "qx".

Step 0: The REPL



This step is basically just creating a skeleton of your interpreter.

- Create a `step0_repl.qx` file in `quux/`.
- Add the 4 trivial functions `READ`, `EVAL`, `PRINT`, and `rep` (read-eval-print). `READ`, `EVAL`, and `PRINT` are basically just stubs that return their first parameter (a string if your target language is a statically typed) and `rep` calls them in order passing the return to the input of the next.

- Add a main loop that repeatedly prints a prompt (needs to be "user> " for later tests to pass), gets a line of input from the user, calls `rep` with that line of input, and then prints out the result from `rep`. It should also exit when you send it an EOF (often Ctrl-D).
- If you are using a compiled (ahead-of-time rather than just-in-time) language, then create a Makefile (or appropriate project definition file) in your directory.

It is time to run your first tests. This will check that your program does input and output in a way that can be captured by the test harness. Go to the top level and run the following:

```
make "test^quux^step0"
```

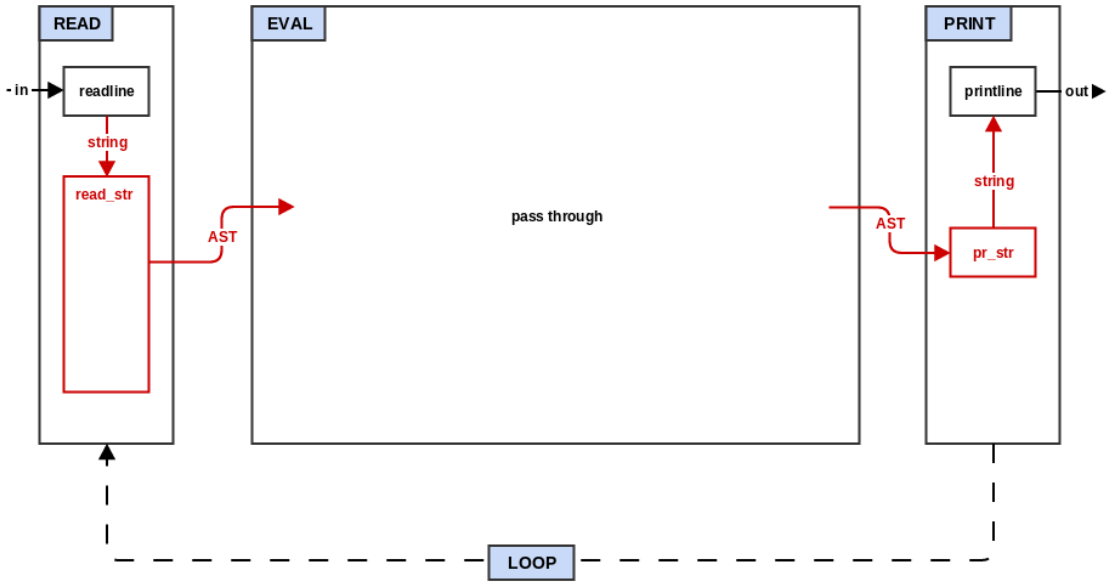
Add and then commit your new `step0_repl.qx` and `Makefile` to git.

Congratulations! You have just completed the first step of the make-a-lisp process.

Optional:

- Add full line editing and command history support to your interpreter REPL. Many languages have a library/module that provide line editing support. Another option if your language supports it is to use an FFI (foreign function interface) to load and call directly into GNU readline, editline, or linenoise library. Add line editing interface code to `readline.qx`

Step 1: Read and Print



In this step, your interpreter will "read" the string from the user and parse it into an internal tree data structure (an abstract syntax tree) and then take that data structure and "print" it back to a string.

In non-lisp languages, this step (called "lexing and parsing") can be one of the most complicated parts of the compiler/interpreter. In Lisp, the data structure that you want in memory is basically represented directly in the code that the programmer writes (homoiconicity).

For example, if the string is "(+ 2 (* 3 4))" then the read function will process this into a tree structure that looks like this:

```
          List
         / |  \
        /  |   \
       /   |    \
Sym:+  Int:2  List
              / |  \
             /  |   \
            /   |    \
       Sym:*  Int:3  Int:4
```

Each left paren and its matching right paren (lisp "sexpr") becomes a node in the tree and everything else becomes a leaf in the tree.

If you can find code for an implementation of a JSON encoder/decoder in your target language then you can probably just borrow and modify that and be 75% of the way done with this step.

The rest of this section is going to assume that you are not starting from an existing JSON encoder/decoder, but that you do have access to a Perl compatible regular expressions (PCRE) module/library. You can certainly implement the reader using simple string operations, but it is more involved. The `make`, `ps` (postscript) and Haskell implementations have examples of a reader/parser without using regular expression support.

- Copy `step0_repl.qx` to `step1_read_print.qx`.
- Add a `reader.qx` file to hold functions related to the reader.
- If the target language has objects types (OOP), then the next step is to create a simple stateful Reader object in `reader.qx`. This object will store the tokens and a position. The Reader object will have two methods: `next` and `peek`. `next` returns the token at the current position and increments the position. `peek` just returns the token at the current position.
- Add a function `read_str` in `reader.qx`. This function will call `tokenize` and then create a new Reader object instance with the tokens. Then it will call `read_form` with the Reader instance.
- Add a function `tokenize` in `reader.qx`. This function will take a single string and return an array/list of all the tokens (strings) in it. The following regular expression (PCRE) will match all mal tokens.

```
[\s,]*(~@|[\[\]{}()'`~^@]|"(?:\\.|[^\\"])*"?|;.*|[^\s\[\]{}
('"`,;)]*)
```

- For each match captured within the parenthesis starting at char 6 of the regular expression a new token will be created.
    - `[\s,]*`: Matches any number of whitespaces or commas. This is not captured so it will be ignored and not tokenized.
    - `~@`: Captures the special two-characters `~@` (tokenized).

    - `[\[\]{}()'`~^@]` : Captures any special single character, one of `[]{}()'`~^@` (tokenized).
    - `"(?:\\.|[^\\"])*"?` : Starts capturing at a double-quote and stops at the next double-quote unless it was preceded by a backslash in which case it includes it until the next double-quote (tokenized). It will also match unbalanced strings (no ending double-quote) which should be reported as an error.
    - `;.*` : Captures any sequence of characters starting with `;` (tokenized).
    - `[^\s\[\]{}('"`,;)]*` : Captures a sequence of zero or more non special characters (e.g. symbols, numbers, "true", "false", and "nil") and is sort of the inverse of the one above that captures special characters (tokenized).

- Add the function `read_form` to `reader.qx`. This function will peek at the first token in the Reader object and switch on the first character of that token. If the character is a left paren then `read_list` is called with the Reader object. Otherwise, `read_atom` is called with the Reader Object. The return value from `read_form` is a mal data type. If your target language is statically typed then you will need some way for `read_form` to return a variant or subclass type. For example, if your language is object oriented, then you can define a top level MalType (in `types.qx`) that all your mal data types inherit from. The MalList type (which also inherits from MalType) will contain a list/array of other MalTypes. If your language is dynamically typed then you can likely just return a plain list/array of other mal types.

- Add the function `read_list` to `reader.qx`. This function will repeatedly call `read_form` with the Reader object until it encounters a ')' token (if it reach EOF before reading a ')' then that is an error). It accumulates the results into a List type. If your language does not have a sequential data type that can hold mal type values you may need to implement one (in `types.qx`). Note that `read_list` repeatedly calls `read_form` rather than `read_atom`. This mutually recursive definition between `read_list` and `read_form` is what allows lists to contain lists.

- Add the function `read_atom` to `reader.qx`. This function will look at the contents of the token and return the appropriate scalar (simple/single) data type value. Initially, you can just implement numbers (integers) and symbols. This will allow you to proceed through the next couple of steps before you will need to implement the other fundamental mal types: nil, true, false, and string. The remaining scalar mal type, keyword does not need to be implemented until step A (but can be implemented at any point between this step and that). BTW, symbols types are just an object that contains a single string name value (some languages have symbol types already).

- Add a file `printer.qx`. This file will contain a single function `pr_str` which does the opposite of `read_str`: take a mal data structure and return a string representation of it. But `pr_str` is much simpler and is basically just a switch statement on the type of the input object:

    - symbol: return the string name of the symbol
    - number: return the number as a string
    - list: iterate through each element of the list calling `pr_str` on it, then join the results with a space separator, and surround the final result with parens

- Change the `READ` function in `step1_read_print.qx` to call `reader.read_str` and the `PRINT` function to call `printer.pr_str`. `EVAL` continues to simply return its input but the type is now a mal data type.

You now have enough hooked up to begin testing your code. You can manually try some simple inputs:

- `123 -> 123`
- `123  -> 123`
- `abc -> abc`
- `abc  -> abc`
- `(123 456) -> (123 456)`
- `( 123 456 789 )  -> (123 456 789)`
- `( + 2 (* 3 4) )  -> (+ 2 (* 3 4))`

To verify that your code is doing more than just eliminating extra spaces (and not failing), you can instrument your `reader.qx` functions.

Once you have gotten past those simple manual tests, it is time to run the full suite of step 1 tests. Go to the top level and run the following:

```
make "test^quux^step1"
```

Fix any test failures related to symbols, numbers and lists.

Depending on the functionality of your target language, it is likely that you have now just completed one of the most difficult steps. It is down hill from here. The remaining steps will probably be easier and each step will give progressively more bang for the buck.

Deferrable:

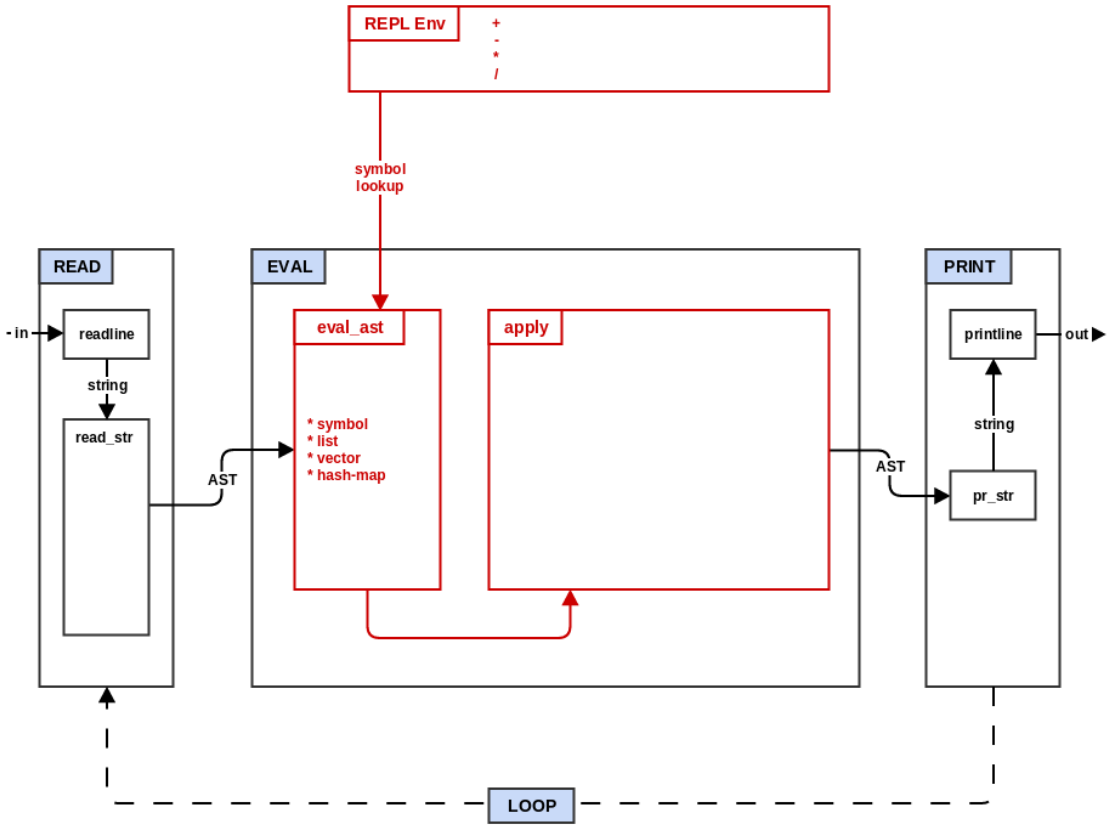
- Add support for the other basic data type to your reader and printer functions: string, nil, true, and false. Nil, true, and false become mandatory at step 4, strings at step 6. When a string is read, the following transformations are applied: a backslash followed by a doublequote is translated into a plain doublequote character, a backslash followed by "n" is translated into a newline, and a backslash followed by another backslash is translated into a single backslash. To properly print a string (for step 4 string functions), the `pr_str` function needs another parameter called `print_readably`. When `print_readably` is true, doublequotes, newlines, and backslashes are translated into their printed representations (the reverse of the reader). The `PRINT` function in the main program should call `pr_str` with print_readably set to true.
- Add error checking to your reader functions to make sure parens are properly matched. Catch and print these errors in your main loop. If your language does not have try/catch style bubble up exception handling, then you will need to add explicit error handling to your code to catch and pass on errors without crashing.
- Add support for reader macros which are forms that are transformed into other forms during the read phase. Refer to `tests/step1_read_print.mal` for the form that these macros should take (they are just simple transformations of the token stream).
- Add support for the other mal types: keyword, vector, hash-map.
    - keyword: a keyword is a token that begins with a colon. A keyword can just be stored as a string with special unicode prefix like 0x29E (or char 0xff/127 if the target language does not have good unicode support) and the printer translates strings with that prefix back to the keyword representation. This makes it easy to use keywords as hash map keys in

most languages. You can also store keywords as a unique data type, but you will need to make sure they can be used as hash map keys (which may involve doing a similar prefixed translation anyways).

    - vector: a vector can be implemented with same underlying type as a list as long as there is some mechanism to keep track of the difference. You can use the same reader function for both lists and vectors by adding parameters for the starting and ending tokens.
    - hash-map: a hash-map is an associative data structure that maps strings to other mal values. If you implement keywords as prefixed strings, then you only need a native associative data structure which supports string keys. Clojure allows any value to be a hash map key, but the base functionality in mal is to support strings and keyword keys. Because of the representation of hash-maps as an alternating sequence of keys and values, you can probably use the same reader function for hash-maps as lists and vectors with parameters to indicate the starting and ending tokens. The odd tokens are then used for keys with the corresponding even tokens as the values.

- Add comment support to your reader. The tokenizer should ignore tokens that start with ";". Your `read_str` function will need to properly handle when the tokenizer returns no values. The simplest way to do this is to return `nil` mal value. A cleaner option (that does not print `nil` at the prompt is to throw a special exception that causes the main loop to simply continue at the beginning of the loop without calling `rep`.

Step 2: Eval



In step 1 your mal interpreter was basically just a way to validate input and eliminate extraneous white space. In this step you will turn your interpreter into a simple number calculator by adding functionality to the evaluator (EVAL).

Compare the pseudocode for step 1 and step 2 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step1_read_print.txt ../process/step2_eval.txt
```

- Copy `step1_read_print.qx` to `step2_eval.qx` .
- Define a simple initial REPL environment. This environment is an associative structure that maps symbols (or symbol names) to numeric functions. For example, in python this would look something like this:

```
repl_env = {'+': lambda a,b: a+b,
            '-': lambda a,b: a-b,
            '*': lambda a,b: a*b,
            '/': lambda a,b: int(a/b)}
```

- Modify the `rep` function to pass the REPL environment as the second parameter for the `EVAL` call.
- Create a new function `eval_ast` which takes `ast` (mal data type) and an associative structure (the environment from above). `eval_ast` switches on the type of `ast` as follows:
    - symbol: lookup the symbol in the environment structure and return the value or raise an error if no value is found
    - list: return a new list that is the result of calling `EVAL` on each of the members of the list
    - otherwise just return the original `ast` value
- Modify `EVAL` to check if the first parameter `ast` is a list.
    - `ast` is not a list: then return the result of calling `eval_ast` on it.
    - `ast` is a empty list: return ast unchanged.
    - `ast` is a list: call `eval_ast` to get a new evaluated list. Take the first item of the evaluated list and call it as function using the rest of the evaluated list as its arguments.

If your target language does not have full variable length argument support (e.g. variadic, vararg, splats, apply) then you will need to pass the full list of arguments as a single parameter and split apart the individual values inside of every mal function. This is annoying, but workable.

The process of taking a list and invoking or executing it to return something new is known in Lisp as the "apply" phase.

Try some simple expressions:

- `(+ 2 3)` -> `5`
- `(+ 2 (* 3 4))` -> `14`

The most likely challenge you will encounter is how to properly call a function references using an arguments list.

Now go to the top level, run the step 2 tests and fix the errors.

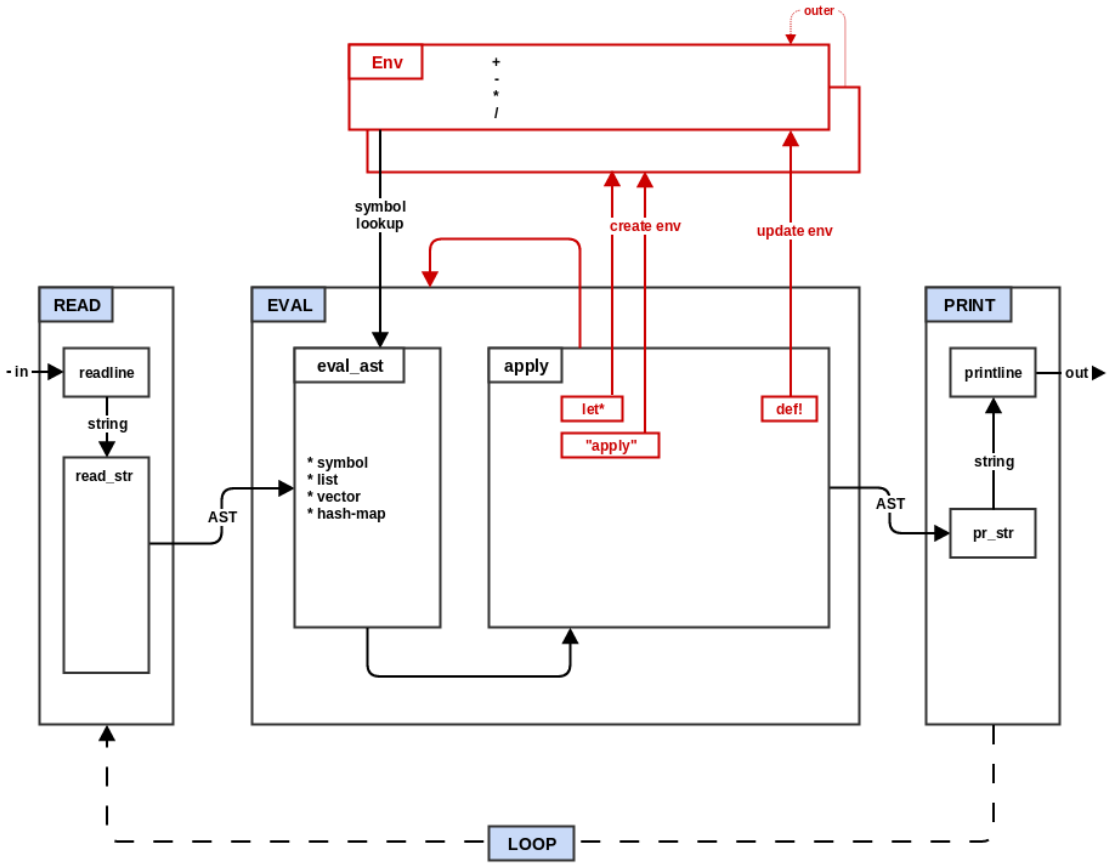
```
make "test^quux^step2"
```

You now have a simple prefix notation calculator!

Deferrable:

- `eval_ast` should evaluate elements of vectors and hash-maps. Add the following cases in `eval_ast`:
  - If `ast` is a vector: return a new vector that is the result of calling `EVAL` on each of the members of the vector.
  - If `ast` is a hash-map: return a new hash-map which consists of key-value pairs where the key is a key from the hash-map and the value is the result of calling `EVAL` on the corresponding value.

Step 3: Environments



In step 2 you were already introduced to REPL environment (`repl_env`) where the basic numeric functions were stored and looked up. In this step you will add the ability to create new environments (`let*`) and modify existing environments (`def!`).

A Lisp environment is an associative data structure that maps symbols (the keys) to values. But Lisp environments have an additional important function: they can refer to another environment (the outer environment). During environment lookups, if the current environment does not have the symbol, the lookup continues in the outer environment, and continues this way until the symbol is either found, or the outer environment is `nil` (the outermost environment in the chain).

Compare the pseudocode for step 2 and step 3 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step2_eval.txt ../process/step3_env.txt
```

- Copy `step2_eval.qx` to `step3_env.qx`.
- Create `env.qx` to hold the environment definition.
- Define an `Env` object that is instantiated with a single `outer` parameter and starts with an empty associative data structure property `data`.
- Define three methods for the Env object:
    - set: takes a symbol key and a mal value and adds to the `data` structure
    - find: takes a symbol key and if the current environment contains that key then return the environment. If no key is found and outer is not `nil` then call find (recurse) on the outer environment.
    - get: takes a symbol key and uses the `find` method to locate the environment with the key, then returns the matching value. If no key is found up the outer chain, then throws/raises a "not found" error.
- Update `step3_env.qx` to use the new `Env` type to create the repl_env (with a `nil` outer value) and use the `set` method to add the numeric functions.
- Modify `eval_ast` to call the `get` method on the `env` parameter.
- Modify the apply section of `EVAL` to switch on the first element of the list:
    - symbol "def!": call the set method of the current environment (second parameter of `EVAL` called `env`) using the unevaluated first parameter (second list element) as the symbol key and the evaluated second parameter as the value.
    - symbol "let*": create a new environment using the current environment as the outer value and then use the first parameter as a list of new bindings in

the "let*" environment. Take the second element of the binding list, call `EVAL` using the new "let*" environment as the evaluation environment, then call `set` on the "let*" environment using the first binding list element as the key and the evaluated second element as the value. This is repeated for each odd/even pair in the binding list. Note in particular, the bindings earlier in the list can be referred to by later bindings. Finally, the second parameter (third element) of the original `let*` form is evaluated using the new "let*" environment and the result is returned as the result of the `let*` (the new let environment is discarded upon completion).

- otherwise: call `eval_ast` on the list and apply the first element to the rest as before.

`def!` and `let*` are Lisp "specials" (or "special atoms") which means that they are language level features and more specifically that the rest of the list elements (arguments) may be evaluated differently (or not at all) unlike the default apply case where all elements of the list are evaluated before the first element is invoked. Lists which contain a "special" as the first element are known as "special forms". They are special because they follow special evaluation rules.

Try some simple environment tests:

- `(def! a 6) -> 6`
- `a -> 6`
- `(def! b (+ a 2)) -> 8`
- `(+ a b) -> 14`
- `(let* (c 2) c) -> 2`

Now go to the top level, run the step 3 tests and fix the errors.

```
make "test^quux^step3"
```

You mal implementation is still basically just a numeric calculator with save/restore capability. But you have set the foundation for step 4 where it will begin to feel like a real programming language.

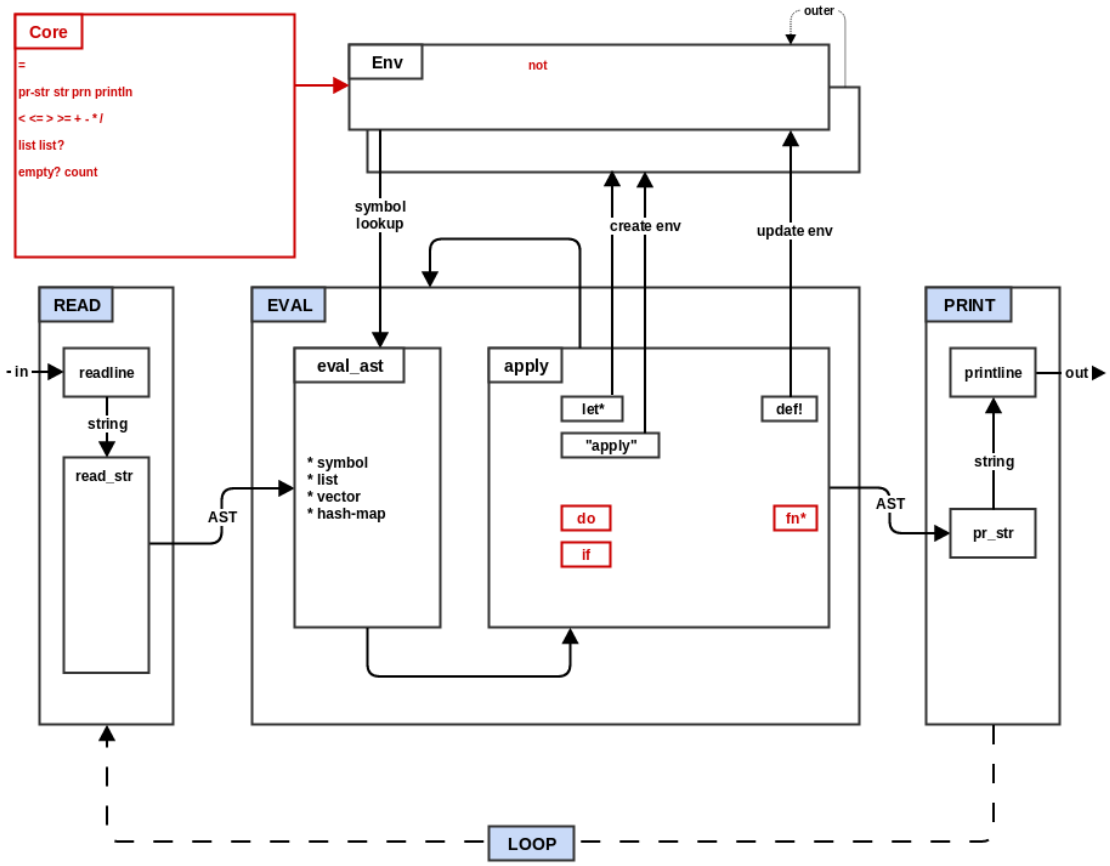
An aside on mutation and typing:

The "!" suffix on symbols is used to indicate that this symbol refers to a function that mutates something else. In this case, the `def!` symbol indicates a special form that will mutate the current environment. Many (maybe even most) of runtime problems that are encountered in software engineering are a result of mutation. By clearly marking code where mutation may occur, you can more easily track down the likely cause of runtime problems when they do occur.

Another cause of runtime errors is type errors, where a value of one type is unexpectedly treated by the program as a different and incompatible type. Statically typed languages try to make the programmer solve all type problems before the program is allowed to run. Most Lisp variants tend to be dynamically typed (types of values are checked when they are actually used at runtime).

As an aside-aside: The great debate between static and dynamic typing can be understood by following the money. Advocates of strict static typing use words like "correctness" and "safety" and thus get government and academic funding. Advocates of dynamic typing use words like "agile" and "time-to-market" and thus get venture capital and commercial funding.

Step 4: If Fn Do



In step 3 you added environments and the special forms for manipulating environments. In this step you will add 3 new special forms (`if` , `fn*` and `do`) and add several more core functions to the default REPL environment. Our new architecture will look like this:

The `fn*` special form is how new user-defined functions are created. In some Lisps, this special form is named "lambda".

Compare the pseudocode for step 3 and step 4 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step3_env.txt ../process/step4_if_fn_do.txt
```

- Copy `step3_env.qx` to `step4_if_fn_do.qx` .
- If you have not implemented reader and printer support (and data types) for `nil` , `true` and `false` , you will need to do so for this step.
- Update the constructor/initializer for environments to take two new arguments: `binds` and `exprs` . Bind (`set`) each element (symbol) of the binds list to the respective element of the `exprs` list.
- Add support to `printer.qx` to print functions values. A string literal like "#<function>" is sufficient.
- Add the following special forms to `EVAL` :
    - `do` : Evaluate all the elements of the list using `eval_ast` and return the final evaluated element.
    - `if` : Evaluate the first parameter (second element). If the result (condition) is anything other than `nil` or `false` , then evaluate the second parameter (third element of the list) and return the result. Otherwise, evaluate the third parameter (fourth element) and return the result. If condition is false and there is no third parameter, then just return `nil` .
    - `fn*` : Return a new function closure. The body of that closure does the following:
        - Create a new environment using `env` (closed over from outer scope) as the `outer` parameter, the first parameter (second list element of `ast` from the outer scope) as the `binds` parameter, and the parameters to the closure as the `exprs` parameter.
        - Call `EVAL` on the second parameter (third list element of `ast` from outer scope), using the new environment. Use the result as the return

value of the closure.

If your target language does not support closures, then you will need to implement `fn*` using some sort of structure or object that stores the values being closed over: the first and second elements of the `ast` list (function parameter list and function body) and the current environment `env`. In this case, your native functions will need to be wrapped in the same way. You will probably also need a method/function that invokes your function object/structure for the default case of the apply section of `EVAL`.

Try out the basic functionality you have implemented:

- `(fn* (a) a) -> #<function>`
- `( (fn* (a) a) 7) -> 7`
- `( (fn* (a) (+ a 1)) 10) -> 11`
- `( (fn* (a b) (+ a b)) 2 3) -> 5`
- Add a new file `core.qx` and define an associative data structure `ns` (namespace) that maps symbols to functions. Move the numeric function definitions into this structure.
- Modify `step4_if_fn_do.qx` to iterate through the `core.ns` structure and add (`set`) each symbol/function mapping to the REPL environment (`repl_env`).
- Add the following functions to `core.ns`:
    - `prn`: call `pr_str` on the first parameter with `print_readably` set to true, prints the result to the screen and then return `nil`. Note that the full version of `prn` is a deferrable below.
    - `list`: take the parameters and return them as a list.
    - `list?`: return true if the first parameter is a list, false otherwise.
    - `empty?`: treat the first parameter as a list and return true if the list is empty and false if it contains any elements.
    - `count`: treat the first parameter as a list and return the number of elements that it contains.
    - `=`: compare the first two parameters and return true if they are the same type and contain the same value. In the case of equal length lists, each element of the list should be compared for equality and if they are the same return true, otherwise false.

    - `<`, `<=`, `>`, and `>=`: treat the first two parameters as numbers and do the corresponding numeric comparison, returning either true or false.

Now go to the top level, run the step 4 tests. There are a lot of tests in step 4 but all of the non-optional tests that do not involve strings should be able to pass now.

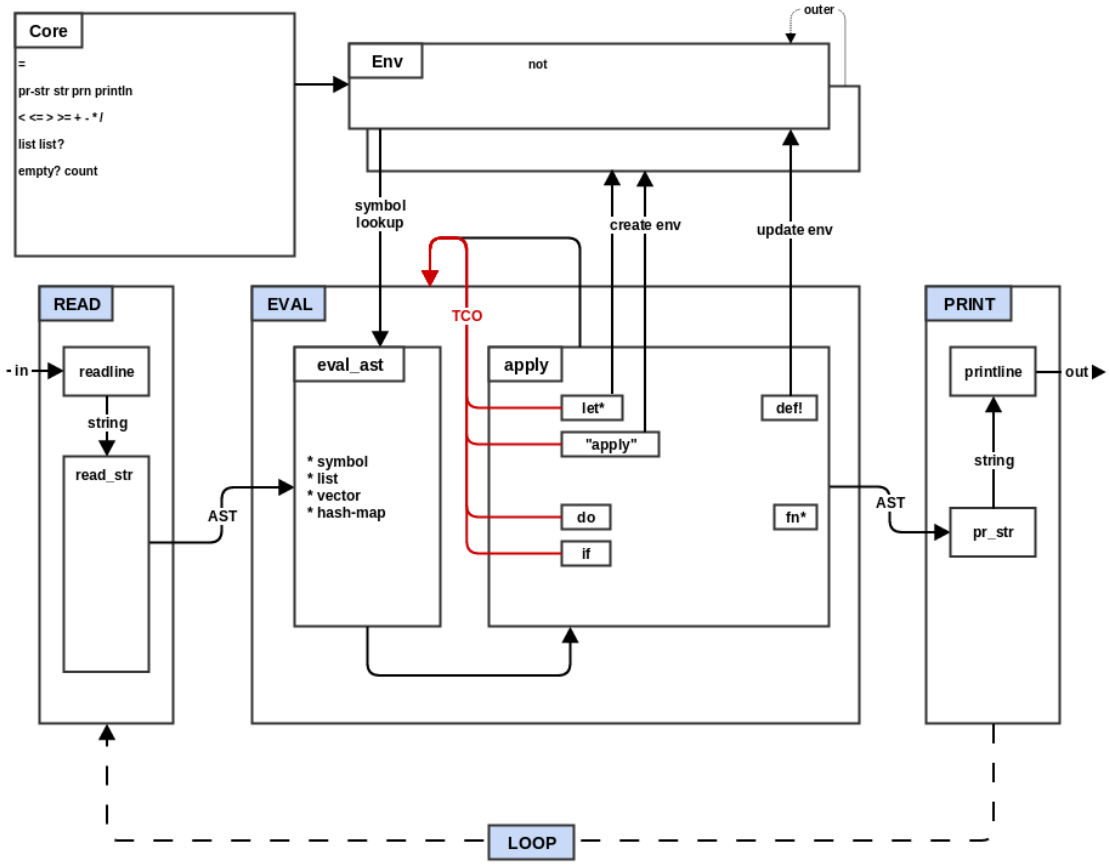
```
make "test^quux^step4"
```

Your mal implementation is already beginning to look like a real language. You have flow control, conditionals, user-defined functions with lexical scope, side-effects (if you implement the string functions), etc. However, our little interpreter has not quite reached Lisp-ness yet. The next several steps will take your implementation from a neat toy to a full featured language.

Deferrable:

- Implement Clojure-style variadic function parameters. Modify the constructor/initializer for environments, so that if a "&" symbol is encountered in the `binds` list, the next symbol in the `binds` list after the "&" is bound to the rest of the `exprs` list that has not been bound yet.
- Define a `not` function using mal itself. In `step4_if_fn_do.qx` call the `rep` function with this string: "(def! not (fn* (a) (if a false true)))".
- Implement the strings functions in `core.qx`. To implement these functions, you will need to implement the string support in the reader and printer (deferrable section of step 1). Each of the string functions takes multiple mal values, prints them (`pr_str`) and joins them together into a new string.
    - `pr-str`: calls `pr_str` on each argument with `print_readably` set to true, joins the results with " " and returns the new string.
    - `str`: calls `pr_str` on each argument with `print_readably` set to false, concatenates the results together ("" separator), and returns the new string.
    - `prn`: calls `pr_str` on each argument with `print_readably` set to true, joins the results with " ", prints the string to the screen and then returns `nil`.
    - `println`: calls `pr_str` on each argument with `print_readably` set to false, joins the results with " ", prints the string to the screen and then returns `nil`.

Step 5: Tail call optimization



In step 4 you added special forms `do`, `if` and `fn*` and you defined some core functions. In this step you will add a Lisp feature called tail call optimization (TCO). Also called "tail recursion" or sometimes just "tail calls".

Several of the special forms that you have defined in `EVAL` end up calling back into `EVAL`. For those forms that call `EVAL` as the last thing that they do before returning (tail call) you will just loop back to the beginning of eval rather than calling it again. The advantage of this approach is that it avoids adding more frames to the call stack. This is especially important in Lisp languages because they tend to prefer using recursion instead of iteration for control structures. (Though some Lisps, such as Common Lisp, have iteration.) However, with tail call optimization, recursion can be made as stack efficient as iteration.

Compare the pseudocode for step 4 and step 5 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step4_if_fn_do.txt ../process/step5_tco.txt
```

- Copy `step4_if_fn_do.qx` to `step5_tco.qx`.

- Add a loop (e.g. while true) around all code in `EVAL`.
- Modify each of the following form cases to add tail call recursion support:
  - `let*`: remove the final `EVAL` call on the second `ast` argument (third list element). Set `env` (i.e. the local variable passed in as second parameter of `EVAL`) to the new let environment. Set `ast` (i.e. the local variable passed in as first parameter of `EVAL`) to be the second `ast` argument. Continue at the beginning of the loop (no return).
  - `do`: change the `eval_ast` call to evaluate all the parameters except for the last (2nd list element up to but not including last). Set `ast` to the last element of `ast`. Continue at the beginning of the loop (`env` stays unchanged).
  - `if`: the condition continues to be evaluated, however, rather than evaluating the true or false branch, `ast` is set to the unevaluated value of the chosen branch. Continue at the beginning of the loop (`env` is unchanged).
- The return value from the `fn*` special form will now become an object/structure with attributes that allow the default invoke case of `EVAL` to do TCO on mal functions. Those attributes are:
  - `ast`: the second `ast` argument (third list element) representing the body of the function.
  - `params`: the first `ast` argument (second list element) representing the parameter names of the function.
  - `env`: the current value of the `env` parameter of `EVAL`.
  - `fn`: the original function value (i.e. what was return by `fn*` in step 4). Note that this is deferrable until step 9 when it is required for the `map` and `apply` core functions). You will also need it in step 6 if you choose to not to defer atoms/ `swap!` from that step.
- The default "apply"/invoke case of `EVAL` must now be changed to account for the new object/structure returned by the `fn*` form. Continue to call `eval_ast` on `ast`. The first element of the result of `eval_ast` is `f` and the remaining elements are in `args`. Switch on the type of `f`:
  - regular function (not one defined by `fn*`): apply/invoke it as before (in step 4).
  - a `fn*` value: set `ast` to the `ast` attribute of `f`. Generate a new environment using the `env` and `params` attributes of `f` as the `outer`

and `binds` arguments and `args` as the `exprs` argument. Set `env` to the new environment. Continue at the beginning of the loop.

Run some manual tests from previous steps to make sure you have not broken anything by adding TCO.

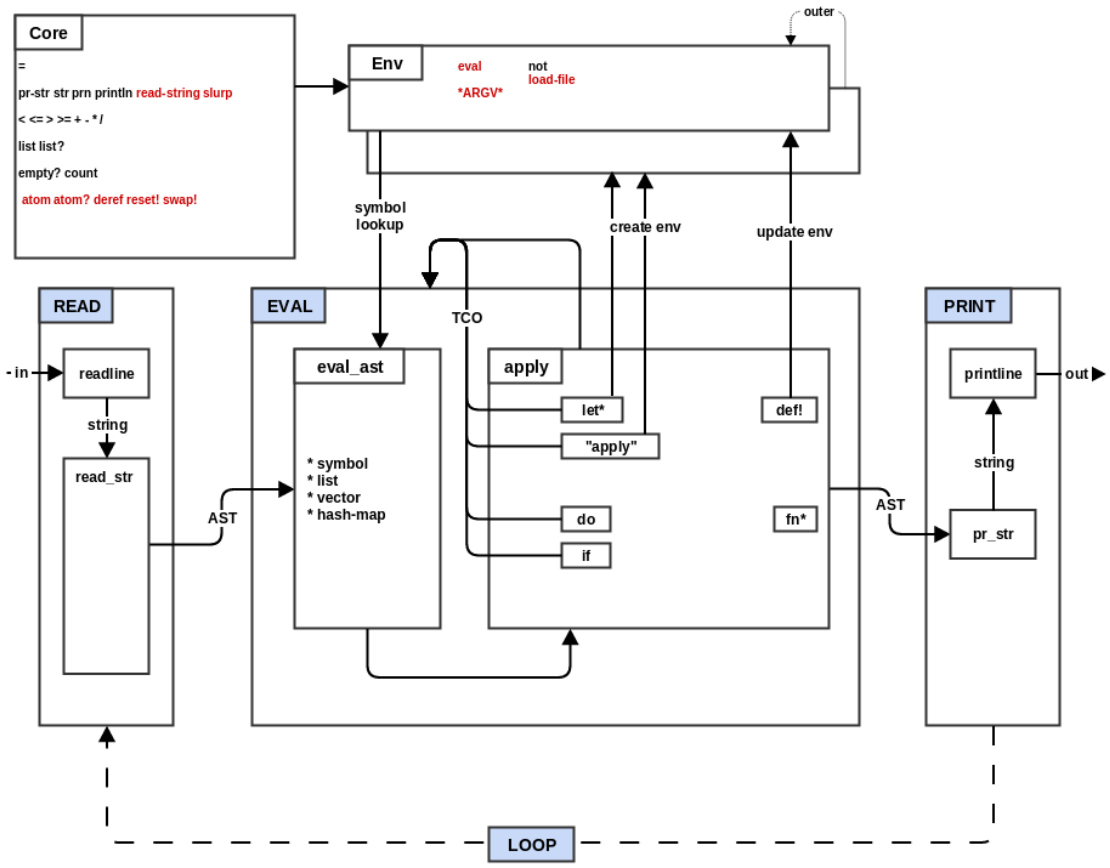
Now go to the top level, run the step 5 tests.

```
make "test^quux^step5"
```

Look at the step 5 test file `tests/step5_tco.mal`. The `sum-to` function cannot be tail call optimized because it does something after the recursive call (`sum-to` calls itself and then does the addition). Lispers say that the `sum-to` is not in tail position. The `sum2` function however, calls itself from tail position. In other words, the recursive call to `sum2` is the last action that `sum2` does. Calling `sum-to` with a large value will cause a stack overflow exception in most target languages (some have super-special tricks they use to avoid stack overflows).

Congratulations, your mal implementation already has a feature (TCO) that most mainstream languages lack.

Step 6: Files, Mutation, and Evil



In step 5 you added tail call optimization. In this step you will add some string and file operations and give your implementation a touch of evil ... er, eval. And as long as your language supports function closures, this step will be quite simple. However, to complete this step, you must implement string type support, so if you have been holding off on that you will need to go back and do so.

Compare the pseudocode for step 5 and step 6 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step5_tco.txt ../process/step6_file.txt
```

- Copy `step5_tco.qx` to `step6_file.qx`.
- Add two new string functions to the core namespaces:
  - `read-string`: this function just exposes the `read_str` function from the reader. If your mal string type is not the same as your target language (e.g. statically typed language) then your `read-string` function will need to unbox (extract) the raw string from the mal string type in order to call `read_str`.

    - `slurp` : this function takes a file name (string) and returns the contents of the file as a string. Once again, if your mal string type wraps a raw target language string, then you will need to unmarshall (extract) the string parameter to get the raw file name string and marshall (wrap) the result back to a mal string type.

- In your main program, add a new symbol "eval" to your REPL environment. The value of this new entry is a function that takes a single argument `ast`. The closure calls the your `EVAL` function using the `ast` as the first argument and the REPL environment (closed over from outside) as the second argument. The result of the `EVAL` call is returned. This simple but powerful addition allows your program to treat mal data as a mal program. For example, you can now to this:

```
(def! mal-prog (list + 1 2))
(eval mal-prog)
```

- Define a `load-file` function using mal itself. In your main program call the `rep` function with this string: "(def! load-file (fn* (f) (eval (read-string (str "(do " (slurp f) "\nnil)")))))".

Try out `load-file`:

- `(load-file "../tests/incA.mal") -> 9`
- `(inc4 3) -> 7`

The `load-file` function does the following:

- Call `slurp` to read in a file by name. Surround the contents with "(do ...)" so that the whole file will be treated as a single program AST (abstract syntax tree). Add a new line in case the files ends with a comment. The `nil` ensures a short and predictable result, instead of what happens to be the last function defined in the loaded file.
- Call `read-string` on the string returned from `slurp`. This uses the reader to read/convert the file contents into mal data/AST.
- Call `eval` (the one in the REPL environment) on the AST returned from `read-string` to "run" it.

Besides adding file and eval support, we'll add support for the atom data type in this step. An atom is the Mal way to represent *state*; it is heavily inspired by Clojure's atoms. An atom holds a reference to a single Mal value of any type; it supports reading that Mal value and *modifying* the reference to point to another Mal value. Note that this is the only Mal data type that is mutable (but the Mal values it refers to are still immutable; immutability is explained in greater detail in step 7). You'll need to add 5 functions to the core namespace to support atoms:

- `atom` : Takes a Mal value and returns a new atom which points to that Mal value.
- `atom?` : Takes an argument and returns `true` if the argument is an atom.
- `deref` : Takes an atom argument and returns the Mal value referenced by this atom.
- `reset!` : Takes an atom and a Mal value; the atom is modified to refer to the given Mal value. The Mal value is returned.
- `swap!` : Takes an atom, a function, and zero or more function arguments. The atom's value is modified to the result of applying the function with the atom's value as the first argument and the optionally given function arguments as the rest of the arguments. The new atom's value is returned. (Side note: Mal is single-threaded, but in concurrent languages like Clojure, `swap!` promises atomic update: `(swap! myatom (fn* [x] (+ 1 x)))` will always increase the `myatom` counter by one and will not suffer from missing updates when the atom is updated from multiple threads.)

Optionally, you can add a reader macro `@` which will serve as a short form for `deref`, so that `@a` is equivalent to `(deref a)`. In order to do that, modify the conditional in reader `read_form` function and add a case which deals with the `@` token: if the token is `@` (at sign) then return a new list that contains the symbol `deref` and the result of reading the next form (`read_form`).

Now go to the top level, run the step 6 tests. The optional tests will need support from the reader for comments, vectors, hash-maps and the `@` reader macro:

```
make "test^quux^step6"
```

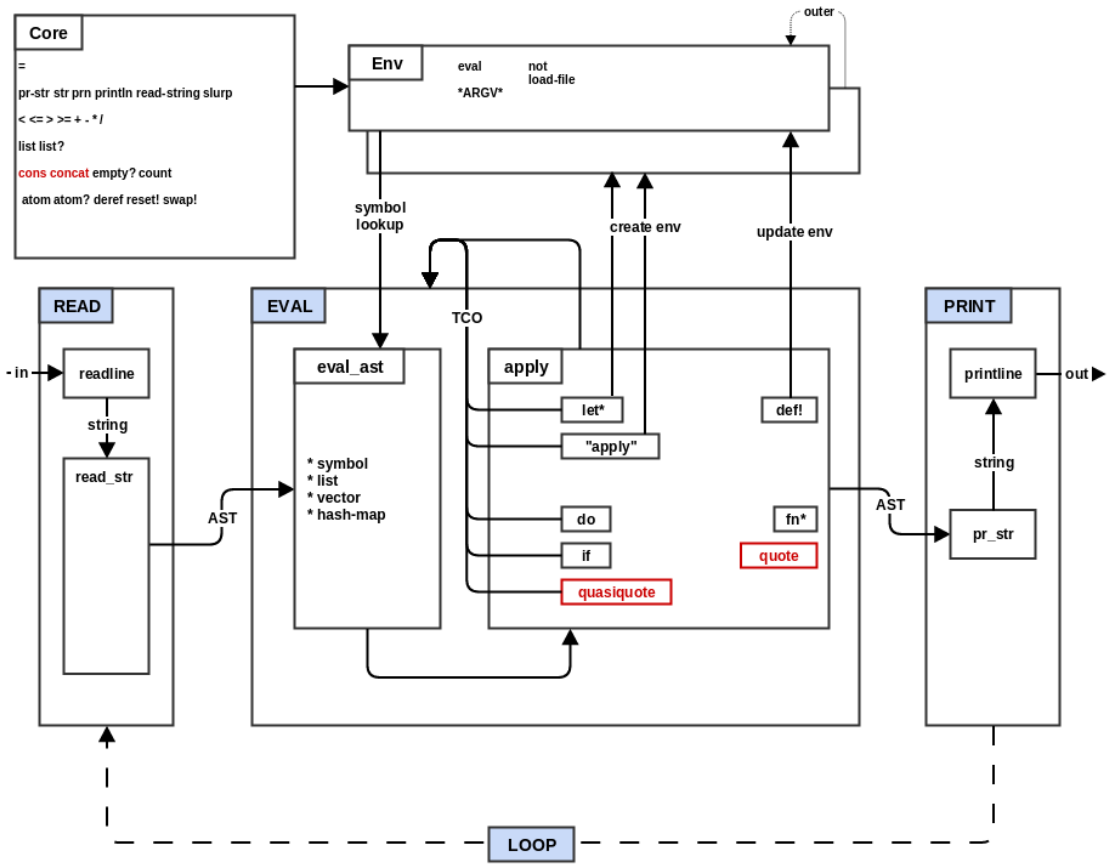
Congratulations, you now have a full-fledged scripting language that can run other mal programs. The `slurp` function loads a file as a string, the `read-string` function calls the mal reader to turn that string into data, and the `eval` function takes data and evaluates it as a normal mal program. However, it is important to note that the `eval` function is not just for running external programs. Because mal programs are regular mal data structures, you can dynamically generate or manipulate those data structures before calling `eval` on them. This isomorphism (same shape) between data and programs is known as "homoiconicity". Lisp languages are homoiconic and this property distinguishes them from most other programming languages.

You mal implementation is quite powerful already but the set of functions that are available (from `core.qx`) is fairly limited. The bulk of the functions you will add are described in step 9 and step A, but you will begin to flesh them out over the next few steps to support quoting (step 7) and macros (step 8).

Deferrable:

- Add the ability to run another mal program from the command line. Prior to the REPL loop, check if your mal implementation is called with command line arguments. If so, treat the first argument as a filename and use `rep` to call `load-file` on that filename, and finally exit/terminate execution.
- Add the rest of the command line arguments to your REPL environment so that programs that are run with `load-file` have access to their calling environment. Add a new "*ARGV*" (symbol) entry to your REPL environment. The value of this entry should be the rest of the command line arguments as a mal list value.

Step 7: Quoting



In step 7 you will add the special forms `quote` and `quasiquote` and add supporting core functions `cons` and `concat`. The two quote forms add a powerful abstraction for manipulating mal code itself (meta-programming).

The `quote` special form indicates to the evaluator (`EVAL`) that the parameter should not be evaluated (yet). At first glance, this might not seem particularly useful but an example of what this enables is the ability for a mal program to refer to a symbol itself rather than the value that it evaluates to. Likewise with lists. For example, consider the following:

- `(prn abc)`: this will lookup the symbol `abc` in the current evaluation environment and print it. This will result in error if `abc` is not defined.
- `(prn (quote abc))`: this will print "abc" (prints the symbol itself). This will work regardless of whether `abc` is defined in the current environment.
- `(prn (1 2 3))`: this will result in an error because `1` is not a function and cannot be applied to the arguments `(2 3)`.
- `(prn (quote (1 2 3)))`: this will print "(1 2 3)".
- `(def! l (quote (1 2 3)))`: list quoting allows us to define lists directly in the code (list literal). Another way of doing this is with the list function: `(def! l (list 1 2 3))`.

The second special quoting form is `quasiquote`. This allows a quoted list to have internal elements of the list that are temporarily unquoted (normal evaluation). There are two special forms that only mean something within a quasiquoted list: `unquote` and `splice-unquote`. These are perhaps best explained with some examples:

- `(def! lst (quote (2 3))) -> (2 3)`
- `(quasiquote (1 (unquote lst))) -> (1 (2 3))`
- `(quasiquote (1 (splice-unquote lst))) -> (1 2 3)`

The `unquote` form turns evaluation back on for its argument and the result of evaluation is put in place into the quasiquoted list. The `splice-unquote` also turns evaluation back on for its argument, but the evaluated value must be a list which is then "spliced" into the quasiquoted list. The true power of the quasiquote form will be manifest when it is used together with macros (in the next step).

Compare the pseudocode for step 6 and step 7 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step6_file.txt ../process/step7_quote.txt
```

- Copy `step6_file.qx` to `step7_quote.qx`.
- Before implementing the quoting forms, you will need to implement some supporting functions in the core namespace:
    - `cons`: this function takes a list as its second parameter and returns a new list that has the first argument prepended to it.
    - `concat`: this functions takes 0 or more lists as parameters and returns a new list that is a concatenation of all the list parameters.

An aside on immutability: note that neither cons or concat mutate their original list arguments. Any references to them (i.e. other lists that they may be "contained" in) will still refer to the original unchanged value. Mal, like Clojure, is a language which uses immutable data structures. I encourage you to read about the power and importance of immutability as implemented in Clojure (from which Mal borrows most of its syntax and feature-set).

- Add the `quote` special form. This form just returns its argument (the second list element of `ast`).

- Add the `quasiquote` special form. First implement a helper function `is_pair` that returns true if the parameter is a non-empty list. Then define a `quasiquote` function. This is called from `EVAL` with the first `ast` argument (second list element) and then `ast` is set to the result and execution continues at the top of the loop (TCO). The `quasiquote` function takes a parameter `ast` and has the following conditional:

    i. if `is_pair` of `ast` is false: return a new list containing: a symbol named "quote" and `ast`.
    ii. else if the first element of `ast` is a symbol named "unquote": return the second element of `ast`.
    iii. if `is_pair` of the first element of `ast` is true and the first element of first element of `ast` (`ast[0][0]`) is a symbol named "splice-unquote": return a new list containing: a symbol named "concat", the second element of first element of `ast` (`ast[0][1]`), and the result of calling `quasiquote` with the second through last element of `ast`.
    iv. otherwise: return a new list containing: a symbol named "cons", the result of calling `quasiquote` on first element of `ast` (`ast[0]`), and the result of calling `quasiquote` with the second through last element of `ast`.

Now go to the top level, run the step 7 tests:

```
make "test^quux^step7"
```

Quoting is one of the more mundane functions available in mal, but do not let that discourage you. Your mal implementation is almost complete, and quoting sets the stage for the next very exciting step: macros.

Deferrable

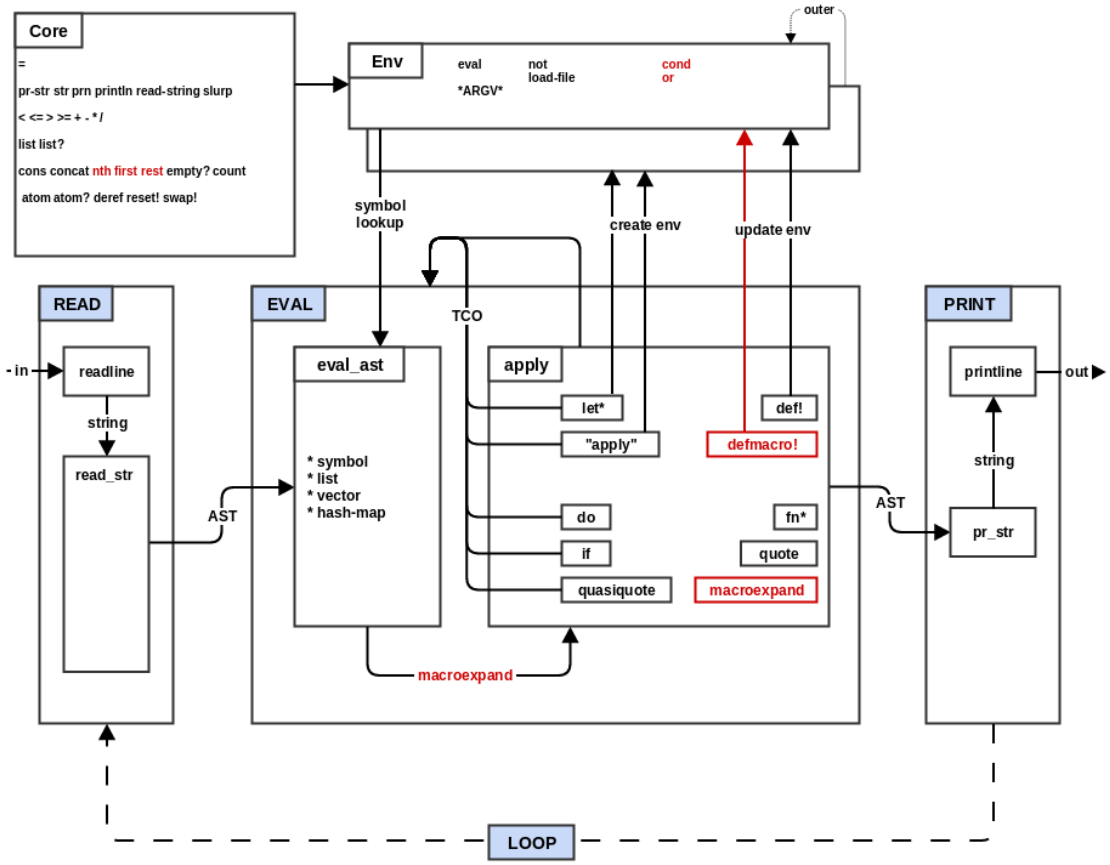
- The full names for the quoting forms are fairly verbose. Most Lisp languages have a short-hand syntax and Mal is no exception. These short-hand syntaxes are known as reader macros because they allow us to manipulate mal code during the reader phase. Macros that run during the eval phase are just called "macros" and are described in the next section. Expand the conditional with reader `read_form` function to add the following four cases:

    - token is "'" (single quote): return a new list that contains the symbol "quote" and the result of reading the next form (`read_form`).
    - token is "`" (back-tick): return a new list that contains the symbol "quasiquote" and the result of reading the next form (`read_form`).

    - token is "~" (tilde): return a new list that contains the symbol "unquote" and the result of reading the next form (`read_form`).
    - token is "~@" (tilde + at sign): return a new list that contains the symbol "splice-unquote" and the result of reading the next form (`read_form`).

- Add support for quoting of vectors. The `is_pair` function should return true if the argument is a non-empty list or vector. `cons` should also accept a vector as the second argument. The return value is a list regardless. `concat` should support concatenation of lists, vectors, or a mix or both. The result is always a list.

Step 8: Macros



Your mal implementation is now ready for one of the most lispy and exciting of all programming concepts: macros. In the previous step, quoting enabled some simple manipulation data structures and therefore manipulation of mal code (because the `eval` function from step 6 turns mal data into code). In this step you will be able to mark mal functions as macros which can manipulate mal code before it is evaluated. In other words, macros are user-defined special forms. Or to look at it another way, macros allow mal programs to redefine the mal language itself.

Compare the pseudocode for step 7 and step 8 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step7_quote.txt ../process/step8_macros.txt
```

- Copy `step7_quote.qx` to `step8_macros.qx`.

You might think that the infinite power of macros would require some sort of complex mechanism, but the implementation is actually fairly simple.

- Add a new attribute `is_macro` to mal function types. This should default to false.
- Add a new special form `defmacro!`. This is very similar to the `def!` form, but before the evaluated value (mal function) is set in the environment, the `is_macro` attribute should be set to true.
- Add a `is_macro_call` function: This function takes arguments `ast` and `env`. It returns true if `ast` is a list that contains a symbol as the first element and that symbol refers to a function in the `env` environment and that function has the `is_macro` attribute set to true. Otherwise, it returns false.
- Add a `macroexpand` function: This function takes arguments `ast` and `env`. It calls `is_macro_call` with `ast` and `env` and loops while that condition is true. Inside the loop, the first element of the `ast` list (a symbol), is looked up in the environment to get the macro function. This macro function is then called/applied with the rest of the `ast` elements (2nd through the last) as arguments. The return value of the macro call becomes the new value of `ast`. When the loop completes because `ast` no longer represents a macro call, the current value of `ast` is returned.
- In the evaluator (`EVAL`) before the special forms switch (apply section), perform macro expansion by calling the `macroexpand` function with the current value of `ast` and `env`. Set `ast` to the result of that call. If the new value of `ast` is no longer a list after macro expansion, then return the result of calling `eval_ast` on it, otherwise continue with the rest of the apply section (special forms switch).

- Add a new special form condition for `macroexpand`. Call the `macroexpand` function using the first `ast` argument (second list element) and `env`. Return the result. This special form allows a mal program to do explicit macro expansion without applying the result (which can be useful for debugging macro expansion).

Now go to the top level, run the step 8 tests:

```
make "test^quux^step8"
```

There is a reasonably good chance that the macro tests will not pass the first time. Although the implementation of macros is fairly simple, debugging runtime bugs with macros can be fairly tricky. If you do run into subtle problems that are difficult to solve, let me recommend a couple of approaches:

- Use the macroexpand special form to eliminate one of the layers of indirection (to expand but skip evaluate). This will often reveal the source of the issue.
- Add a debug print statement to the top of your main `eval` function (inside the TCO loop) to print the current value of `ast` (hint use `pr_str` to get easier to debug output). Pull up the step8 implementation from another language and uncomment its `eval` function (yes, I give you permission to violate the rule this once). Run the two side-by-side. The first difference is likely to point to the bug.

Congratulations! You now have a Lisp interpreter with a super power that most non-Lisp languages can only dream of (I have it on good authority that languages dream when you are not using them). If you are not already familiar with Lisp macros, I suggest the following exercise: write a recursive macro that handles postfixed mal code (with the function as the last parameter instead of the first). Or not. I have not actually done so myself, but I have heard it is an interesting exercise.

In the next step you will add try/catch style exception handling to your implementation in addition to some new core functions. After step9 you will be very close to having a fully self-hosting mal implementation. Let us continue!

Deferrable

- Add the following new core functions which are frequently used in macro functions:
    - `nth`: this function takes a list (or vector) and a number (index) as arguments, returns the element of the list at the given index. If the index is out of range, this function raises an exception.

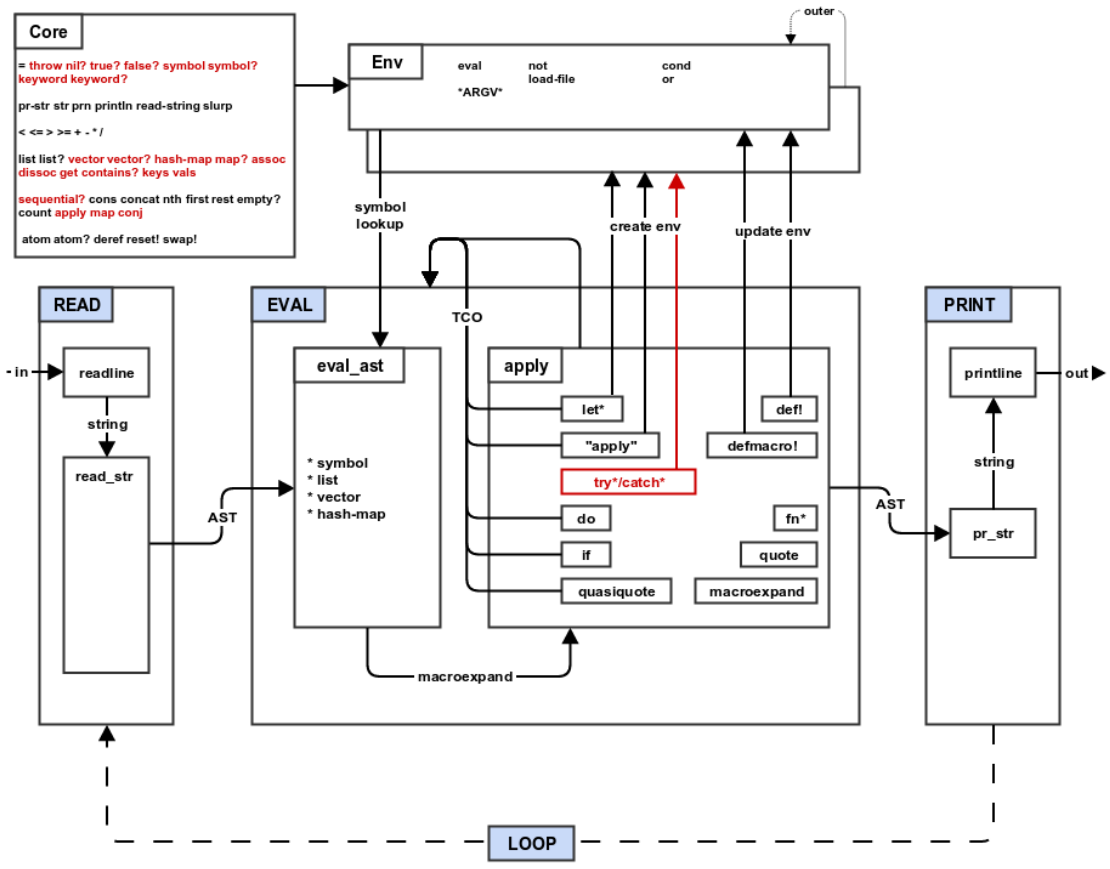
    - `first` : this function takes a list (or vector) as its argument and return the first element. If the list (or vector) is empty or is `nil` then `nil` is returned.
    - `rest` : this function takes a list (or vector) as its argument and returns a new list containing all the elements except the first. If the list (or vector) is empty or is `nil` then `()` (empty list) is returned.

- In the main program, call the `rep` function with the following string argument to define a new control structure.

```
"(defmacro! cond (fn* (& xs) (if (> (count xs) 0) (list 'if (first
xs) (if (> (count xs) 1) (nth xs 1) (throw \"odd number of forms to
cond\")) (cons 'cond (rest (rest xs)))))))"


* Note that `cond` calls the `throw` function when `cond` is
  called with an odd number of args. The `throw` function is
  implemented in the next step, but it will still serve it's
  purpose here by causing an undefined symbol error.
```

Step 9: Try



In this step you will implement the final mal special form for error/exception handling: `try*/catch*`. You will also add several core functions to your implementation. In particular, you will enhance the functional programming pedigree of your implementation by adding the `apply` and `map` core functions.

Compare the pseudocode for step 8 and step 9 to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step8_macros.txt ../process/step9_try.txt
```

- Copy `step8_macros.qx` to `step9_try.qx`.
- Add the `try*/catch*` special form to the EVAL function. The try catch form looks like this: `(try* A (catch* B C))`. The form `A` is evaluated, if it throws an exception, then form `C` is evaluated with a new environment that binds the symbol `B` to the value of the exception that was thrown.
    - If your target language has built-in try/catch style exception handling then you are already 90% of the way done. Add a (native language) try/catch block that evaluates `A` within the try block and catches all exceptions. If an exception is caught, then translate it to a mal type/value. For native exceptions this is either the message string or a mal hash-map that contains the message string and other attributes of the exception. When a regular mal type/value is used as an exception, you will probably need to store it within a native exception type in order to be able to convey/transport it using the native try/catch mechanism. Then you will extract the mal type/value from the native exception. Create a new mal environment that binds `B` to the value of the exception. Finally, evaluate `C` using that new environment.
    - If your target language does not have built-in try/catch style exception handling then you have some extra work to do. One of the most straightforward approaches is to create a a global error variable that stores the thrown mal type/value. The complication is that there are a bunch of places where you must check to see if the global error state is set and return without proceeding. The rule of thumb is that this check should happen at the top of your EVAL function and also right after any call to EVAL (and after any function call that might happen to call EVAL further down the chain). Yes, it is ugly, but you were warned in the section on picking a language.
- Add the `throw` core function.

    - If your language supports try/catch style exception handling, then this function takes a mal type/value and throws/raises it as an exception. In order to do this, you may need to create a custom exception object that wraps a mal value/type.
    - If your language does not support try/catch style exception handling, then set the global error state to the mal type/value.

- Add the `apply` and `map` core functions. In step 5, if you did not add the original function (`fn`) to the structure returned from `fn*`, the you will need to do so now.

    - `apply`: takes at least two arguments. The first argument is a function and the last argument is list (or vector). The arguments between the function and the last argument (if there are any) are concatenated with the final argument to create the arguments that are used to call the function. The apply function allows a function to be called with arguments that are contained in a list (or vector). In other words, `(apply F A B [C D])` is equivalent to `(F A B C D)`.
    - `map`: takes a function and a list (or vector) and evaluates the function against every element of the list (or vector) one at a time and returns the results as a list.

- Add some type predicates core functions. In Lisp, predicates are functions that return true/false (or true value/nil) and typically end in "?" or "p".

    - `nil?`: takes a single argument and returns true (mal true value) if the argument is nil (mal nil value).
    - `true?`: takes a single argument and returns true (mal true value) if the argument is a true value (mal true value).
    - `false?`: takes a single argument and returns true (mal true value) if the argument is a false value (mal false value).
    - `symbol?`: takes a single argument and returns true (mal true value) if the argument is a symbol (mal symbol value).

Now go to the top level, run the step 9 tests:

```
make "test^quux^step9"
```

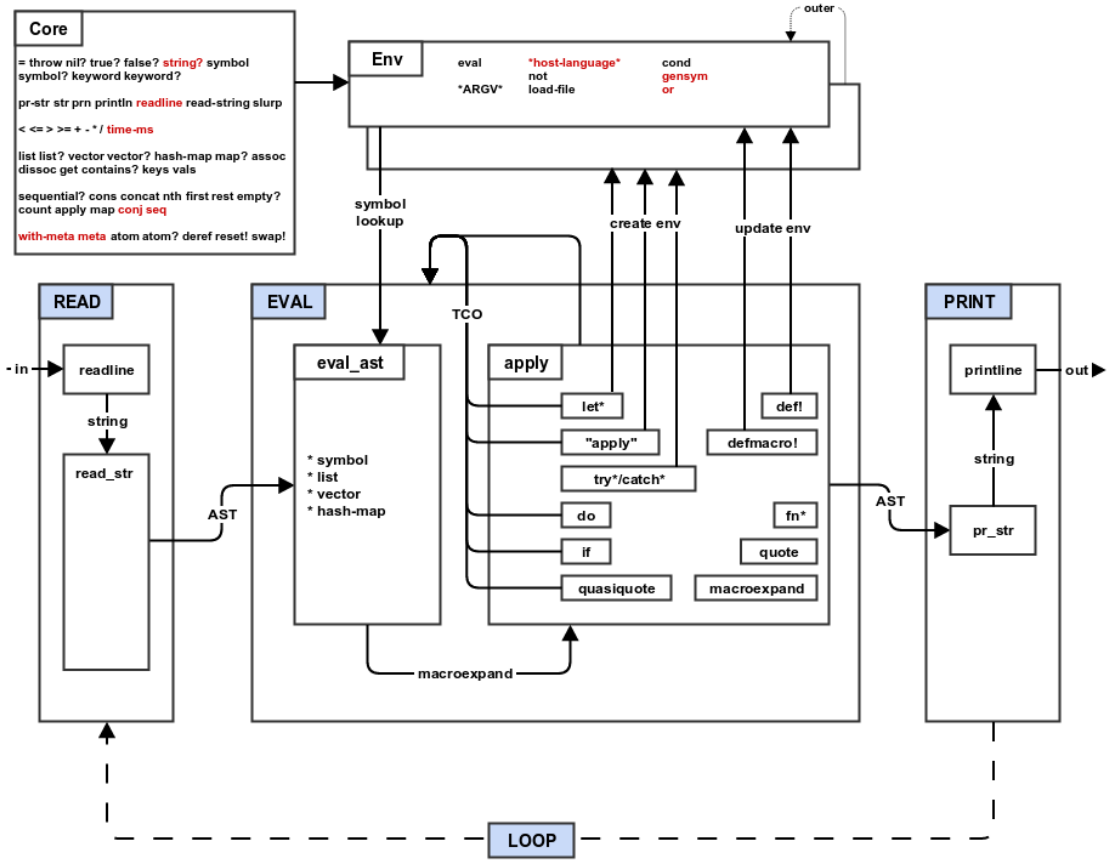
Your mal implementation is now essentially a fully featured Lisp interpreter. But if you stop now you will miss one of the most satisfying and enlightening aspects of creating a mal implementation: self-hosting.

Deferrable

- Add the following new core functions:
  - `symbol` : takes a string and returns a new symbol with the string as its name.
  - `keyword` : takes a string and returns a keyword with the same name (usually just be prepending the special keyword unicode symbol). This function should also detect if the argument is already a keyword and just return it.
  - `keyword?` : takes a single argument and returns true (mal true value) if the argument is a keyword, otherwise returns false (mal false value).
  - `vector` : takes a variable number of arguments and returns a vector containing those arguments.
  - `vector?` : takes a single argument and returns true (mal true value) if the argument is a vector, otherwise returns false (mal false value).
  - `sequential?` : takes a single argument and returns true (mal true value) if it is a list or a vector, otherwise returns false (mal false value).
  - `hash-map` : takes a variable but even number of arguments and returns a new mal hash-map value with keys from the odd arguments and values from the even arguments respectively. This is basically the functional form of the `{}` reader literal syntax.
  - `map?` : takes a single argument and returns true (mal true value) if the argument is a hash-map, otherwise returns false (mal false value).
  - `assoc` : takes a hash-map as the first argument and the remaining arguments are odd/even key/value pairs to "associate" (merge) into the hash-map. Note that the original hash-map is unchanged (remember, mal values are immutable), and a new hash-map containing the old hash-maps key/values plus the merged key/value arguments is returned.
  - `dissoc` : takes a hash-map and a list of keys to remove from the hash-map. Again, note that the original hash-map is unchanged and a new hash-map with the keys removed is returned. Key arguments that do not exist in the hash-map are ignored.
  - `get` : takes a hash-map and a key and returns the value of looking up that key in the hash-map. If the key is not found in the hash-map then nil is returned.
  - `contains?` : takes a hash-map and a key and returns true (mal true value) if the key exists in the hash-map and false (mal false value) otherwise.
  - `keys` : takes a hash-map and returns a list (mal list value) of all the keys in the hash-map.

- `vals` : takes a hash-map and returns a list (mal list value) of all the values in the hash-map.

Step A: Metadata, Self-hosting and Interop



You have reached the final step of your mal implementation. This step is kind of a catchall for things that did not fit into other steps. But most importantly, the changes you make in this step will unlock the magical power known as "self-hosting". You might have noticed that one of the languages that mal is implemented in is "mal". Any mal implementation that is complete enough can run the mal implementation of mal. You might need to pull out your hammock and ponder this for a while if you have never built a compiler or interpreter before. Look at the step source files for the mal implementation of mal (it is not cheating now that you have reached step A).

If you deferred the implementation of keywords, vectors and hash-maps, now is the time to go back and implement them if you want your implementation to self-host.

Compare the pseudocode for step 9 and step A to get a basic idea of the changes that will be made during this step:

```
diff -urp ../process/step9_try.txt ../process/stepA_mal.txt
```

- Copy `step9_try.qx` to `stepA_mal.qx`.
- Add the `readline` core function. This functions takes a string that is used to prompt the user for input. The line of text entered by the user is returned as a string. If the user sends an end-of-file (usually Ctrl-D), then nil is returned.
- Add a new "*host-language*" (symbol) entry to your REPL environment. The value of this entry should be a mal string containing the name of the current implementation.
- When the REPL starts up (as opposed to when it is called with a script and/or arguments), call the `rep` function with this string to print a startup header: "(println (str "Mal [" *host-language* "]"))".
- Ensure that the REPL environment contains definitions for `time-ms`, `meta`, `with-meta`, `fn?` `string?`, `number?`, `seq`, and `conj`. It doesn't really matter what they do at this stage: they just need to be defined. Making them functions that raise a "not implemented" exception would be fine.

Now go to the top level, run the step A tests:

```
make "test^quux^stepA"
```

Once you have passed all the non-optional step A tests, it is time to try self-hosting. Run your step A implementation as normal, but use the file argument mode you added in step 6 to run a each of the step from the mal implementation:

```
./stepA_mal.qx ../mal/step1_read_print.mal
./stepA_mal.qx ../mal/step2_eval.mal
...
./stepA_mal.qx ../mal/step9_try.mal
./stepA_mal.qx ../mal/stepA_mal.mal
```

There is a very good chance that you will encounter an error at some point while trying to run the mal in mal implementation steps above. Debugging failures that happen while self-hosting is MUCH more difficult and mind bending. One of the best approaches I have personally found is to add prn statements to the mal implementation step (not your own implementation of mal) that is causing problems.

Another approach I have frequently used is to pull out the code from the mal implementation that is causing the problem and simplify it step by step until you have a simple piece of mal code that still reproduces the problem. Once the reproducer is simple enough you will probably know where in your own implementation that problem is likely to be. Please add your simple reproducer as a test case so that future implementers will fix similar issues in their code before they get to self-hosting when it is much more difficult to track down and fix.

Once you can manually run all the self-hosted steps, it is time to run all the tests in self-hosted mode:

```
make MAL_IMPL=quux "test^mal"
```

When you run into problems (which you almost certainly will), use the same process described above to debug them.

Congratulations!!! When all the tests pass, you should pause for a moment and consider what you have accomplished. You have implemented a Lisp interpreter that is powerful and complete enough to run a large mal program which is itself an implementation of the mal language. You might even be asking if you can continue the "inception" by using your implementation to run a mal implementation which itself runs the mal implementation.

Optional additions

- Add meta-data support to composite data types (lists, vectors and hash-maps), and to functions (native or not), by adding a new metadata attribute that refers to another mal value/type (nil by default). Add the following metadata related core functions (and remove any stub versions):
    - `meta` : this takes a single mal function argument and returns the value of the metadata attribute.
    - `with-meta` : this function takes two arguments. The first argument is a mal function and the second argument is another mal value/type to set as metadata. A copy of the mal function is returned that has its `meta` attribute set to the second argument. Note that it is important that the environment and macro attribute of mal function are retained when it is copied.
    - Add a reader-macro that expands the token "^" to return a new list that contains the symbol "with-meta" and the result of reading the next next form (2nd argument) (`read_form`) and the next form (1st argument) in

that order (metadata comes first with the ^ macro and the function second).

    - If you implemented as `defmacro!` to mutate an existing function without copying it, you can now use the function copying mechanism used for metadata to make functions immutable even in the defmacro! case...

- Add the following new core functions (and remove any stub versions):

    - `time-ms` : takes no arguments and returns the number of milliseconds since epoch (00:00:00 UTC January 1, 1970), or, if not possible, since another point in time (`time-ms` is usually used relatively to measure time durations). After `time-ms` is implemented, you can run the performance micro-benchmarks by running `make perf^quux` .
    - `conj` : takes a collection and one or more elements as arguments and returns a new collection which includes the original collection and the new elements. If the collection is a list, a new list is returned with the elements inserted at the start of the given list in opposite order; if the collection is a vector, a new vector is returned with the elements added to the end of the given vector.
    - `string?` : returns true if the parameter is a string.
    - `number?` : returns true if the parameter is a number.
    - `fn?` : returns true if the parameter is a function (internal or user-defined).
    - `macro?` : returns true if the parameter is a macro.
    - `seq` : takes a list, vector, string, or nil. If an empty list, empty vector, or empty string ("") is passed in then nil is returned. Otherwise, a list is returned unchanged, a vector is converted into a list, and a string is converted to a list that containing the original string split into single character strings.

- For interop with the target language, add this core function:

    - `quux-eval` : takes a string, evaluates it in the target language, and returns the result converted to the relevant Mal type. You may also add other interop functions as you see fit; Clojure, for example, has a function called `.` which allows calling Java methods. If the target language is a static language, consider using FFI or some language-specific reflection mechanism, if available. The tests for `quux-eval` and any other interop function should be added in `quux/tests/stepA_mal.mal` (see the tests for `lua-eval` as an example).

Next Steps

- Join the #mal IRC channel. It's fairly quiet but there are bursts of interesting conversation related to mal, Lisps, esoteric programming languages, etc.
- If you have created an implementation for a new target language (or a unique and interesting variant of an existing implementation), consider sending a pull request to add it into the main mal repository. The FAQ describes general requirements for getting an implementation merged into the main repository.
- Take your interpreter implementation and have it emit source code in the target language rather than immediately evaluating it. In other words, create a compiler.
- Pick a new target language and implement mal in it. Pick a language that is very different from any that you already know.
- Use your mal implementation to implement a real world project. Many of these will force you to address interop. Some ideas:
  - Web server (with mal as CGI language for extra points)
  - An IRC/Slack chat bot
  - An editor (GUI or curses) with mal as a scripting/extension language.
  - An AI player for a game like Chess or Go.
- Implement a feature in your mal implementation that is not covered by this guide. Some ideas:
  - Namespaces
  - Multi-threading support
  - Errors with line numbers and/or stack traces.
  - Lazy sequences
  - Clojure-style protocols
  - Full call/cc (call-with-current-continuation) support
  - Explicit TCO (i.e. `recur`) with tail-position error checking