Fn Programming Language Manual (pre-0.1)

Jack Pugmire

November 20, 2021

Contents

1 Command Line Interface

All functionality is accessed via that `fn` command line program.

1.1 Invocation

The fn command line tool has the following interface:

```
fn [options] sources ...
```

When invoked without any arguments, a repl is started in namespace fn/user.

The main options are:

`-r` Start a REPL after evaluating all sources.

`--repl-ns <namespace>` Use the specified namespace for the repl, (as opposed to the default `fn/user`).

`-h` Show help and exit.

There are also two more options that are unlikely to be useful for a typical user:

`-d` Print disassembled bytecode after compiling each expression.

`-l` Print LLIR (low-level intermediate representation) before compiling each expression.

The sources are files and strings to evaluate. The types of source arguments are:

`<filename>` Evaluate and load an entire file. The package and namespace are determined automatically (see below).

`--eval <string>` Evaluate a string in the default namespace, fn/user.

`--eval-in <namespace> <string>` Evaluate a string in the given namespace.

`--eval-file <namespace> <string>` Evaluate an entire file in the given namespace. Note that this overrides the file's package declaration and treats it as just a sequence of expressions.

When the filename option above is used, package and namespace are determined based on the file's name and package declaration. The namespace name is set to the stem of the filename, and the package is set to the one specified in the file's package declaration. If the file contains no package declaration, it defaults to using `fn/user` as the package (e.g. a `foo.fn` without a package declaration will be in namespace `fn/user/foo`).

1.2 REPL

In addition to running code, additional functionality can be accessed by typing keywords (i.e. symbols whose names begin with ":"). The available REPL keywords are:

`:quit` quit the interpreter

`:ns <namespace>` Switch to the given namespace.

`:load <filename>` Evaluate a file in the current namespace.

`:reimport <namespace>` Redo a namespace import as if it was for the first time. Be warned that references to the old namespace are not deleted, so the two instances of the namespace can get out of sync and cause errors.

2 Language Guide

2.1 Overview

Fn is a dynamically typed, garbage-collected, general-purpose programming language with Lisp-like syntax. A couple of its defining characteristics are:

- The core language has relatively few operators, with special care taken to ensure they have consistent and concise syntax.
- Fn is designed for a "mostly functional" programming style, where immutability and referential transparency are favored, but not required.
- The native macro system allows definition of new syntax (and even entire domain-specific languages) using regular Fn code and data structures.

The syntax looks like this:

```
(def sqrt-precision 0.001)
(def approx-sqrt (x)
  (letfn iterate (guess)
    (if (< (abs (- x (* guess guess)) sqrt-precision))
        guess
        (iterate (/ (+ guess (/ x guess) 2)))))
  (if (>= x 0)
      (iterate (/ x 2))
      (error "Cannot approximate square root of a negative number.")))
```

2.2 Variables and Mutation

2.2.1 Local Variables

Local variables can be created using one of the special operators `let`, `letfn`, or `with`.[1] They all bind variables in the same way, but with different syntax for programmer convenience. Function parameters are also treated as local variables within the function body.

Before proceeding, we note that the full story about local variables involves variable capture semantics, which are covered in a later section. Variable capture doesn't affect any of the concepts discussed in the rest of this section.

`let` is the most elementary way to create a local variable. It defines one or more new variables in the current lexical environment.

```
;; let binds variables to the given values
(let x 'symbol)
;; multiple definitions can be made in a single let
(let a 16
     b (reverse "string")
     ;; value expressions can refer to variables from earlier in the same let
     c (+ a (length b))
```

`with` is similar to `let`, but rather than creating definitions in the containing environment, it creates a new lexical environment.

[1] In Fn, the `with` operator provides the functionality of what most Lisp-like languages call `let`, while Fn's `let` is quite different, as it acts on the surrounding environment.

```
;; this creates two variables
(with (a 3
       b 4)
  ;; the body can contain multiple expressions
  (println "hello")
  (+ a b))
;; returns 7
;; the variables a, b do not exist outside of the with body
```

`letfn` has a streamlined syntax for creating functions, but otherwise behaves like `let`. See the documentation below for details.

All local variables can have their value changed with `set!`. The exclamation point is because mutation is not to be taken lightly. The syntax for `set!` is like this:

```
(set! var-name new-value)
;; for example
(let var 'hi)
(println var) ;; prints 'hi
(set! var 'lo)
(println var) ;; prints 'lo
```

Note that attempting to `set!` a global variable will result in an error.

2.2.2 Global Variables

Global variables in Fn are created using `def` or `defn`. E.g.

```
(def my-global 'special-constant)
(def my-other-global (+ 21 69))
```

`defn` behaves exactly like `def`, but has special syntax streamlined for defining functions.

Global variables are immutable, i.e. they cannot be changed by using `set!`. However, by assigning global variables to mutable datatypes or by exploiting variable capture (discussed in a later section), mutable state can still be associated to a global variable. This is intended behavior, however, it is not recommended that you abuse it.

2.3 Data Types

Fn provides the following builtin data types (type names in Fn are `Capitalized-Like-This`):

`Nil` The special constant `nil`, used to indicate no value.

`Bool` The special boolean constants `true` and `false`.

`Num` Floating-point numbers. (These are almost IEEE 64-bit floats, but we truncate the significand by four bits to fit type information).

`Symbol` Internalized strings. These are essentially strings with a faster equality test, at the expensive of slower access to the characters of the string. They are used extensively by the macro system.

`String` (Immutable) sequences of bytes. Usually these are UTF-8 encoded character streams.

`List` (Immutable) singley-linked lists.

`Table` Mutable key-value stores.

Of these, only lists and tables logically contain other values. (Substrings can be extracted from strings, but this actually creates a new string object and just copies in data from the other string). So, we call `List` and `Table` the two **compound data types**, and call the rest of them **simple data types**.

2.3.1 Simple Data Types

Here is what the syntax looks like for the simple data types:

```
;; numbers are pretty much what you'd expect
2
-6
3.14159
2.0e-6 ;; we have scientific notation
0xFf ;; hexadecimal, even!

;; strings are enclosed within matched double quotes
"string"
"Fn uses escape codes from C, e.g. \\ \"\n"
""
```

```
;; symbols are prefixed by a single quote.
'sym1
'sym2
;; symbols can contain whitespace and syntax characters, provided they are
;; escaped with a backslash
'sym\ with\ \"escapes\"
;; be careful about the quote operator. If the quoted expression is a number,
;; it will result in a number instead of a symbol. You can get around this
;; with escapes:
'0xb8  ;; this is a number
'\0xb8 ;; this is a symbol

;; booleans and nil are called by name
true
false
nil
```

See also subsection 2.3.4 for more on symbols and the quote operator.

2.3.2 Lists

Lists in Fn are what you'd expect for a functional programming language.
They're created using square brackets or by using the `List` function.

```
[] ; empty list
['a 'b] ; list of two symbols
[1 'a "str"] ; lists may contain objects of arbitrary type

;; List is identical to square bracket syntax
[1 2 3]
(List 1 2 3)
```

Lists can be manipulated with builtin functions:

```
(def list1 [["str" 2] 'a 'b])
(def list2 [0 2 4 6 8 10])

;; head and tail access the head and tail of the list
(head list1) ;=> ["str" 2]
(head list2) ;=> 0
```

```
(tail list1) ;=> ['a 'b]

(tail [])    ;=> []
(head [])    ;=> error (empty list has no head)

;; nth allows random access:
(nth list1 2) ;=> 'b
(nth list2 1) ;=> 2

;; length gives the length of a list
(length []) ;=> 0
(length list1) ;=> 3
(length list2) ;=> 6

;; cons prepends elements
(cons 2 []) ;=> [2]
(cons nil list1) ;=> [nil ["str" 2] 'a 'b]

;; concat concatenates two or more lists
(concat [1 2 3] [4 5 6]) ;=> [1 2 3 4 5 6]
(concat [37] ['foo] ["bar"]) ;=> [37 'foo "bar"]
(concat list2 list1) ;=> [0 2 4 6 8 10 ["str" 2] 'a 'b]

;; reverse reverse the direction of a list
(reverse list2) => [10 8 6 4 2 0]
```

2.3.3 Tables

Tables are key-value stores. Any type of object may be used as a key or a value, (note, however, that it takes longer to hash more complicated data structures since we have to descend on their fields)[2].

Tables are built using braces `{}` or the equivalent `Table` function. This must be passed an even number of arguments.

```
{} ;=> empty table
{'key1 4 'key 6} ;=> table with two kv-pairs
```

[2]Two keys are equal if `(= k1 k2)` is true (using the builtin equality function). For simple data types the meaning of equality is obvious. Lists and tables are compared componentwise. That is, two lists are equal if and only if all their respective entries are equal. Two tables are equal if their key sets are equal (disregarding order), and for each key the corresponding values in each table are equal.

```
(Table 'key1 4 'key 6) ;=> table with two kv-pairs
```

Table elements may be accessed using the builtin function `get`. When the key is a constant symbol, dot syntax (or the equivalent `dot` special operator) can be used instead. This is how this looks:

```
(def tab1 {'name "Mr. Table"
           'occupation "Holds data"
           'child {'name "Table Jr."
                   'occupation "Holds less data"}})
(def tab2 {0 'zero 1 'one 2 'two 3 'three 4 'four})

;; these all return "Mr. Table"
(get tab1 'name)
tab1.name
(dot tab1 name) ; equivalent syntax to the dot expression
;; Note that the symbols in the dot expressions are unquoted. Arguments to dot
;; must be unquoted symbols or a compilation error occurs.

;; get is more flexible than dot and allows arbitrary key and value expressions
(get tab2 (+ 1 2)) ;=> 'three
(get {'k 'v} 'k) ;=> 'v

;; dot makes it convenient to descend on tables with symbolic key names
tab1.child.name ;=> "Table Jr."
;; equivalent expression:
(dot tab1 child name)
```

Since tables are mutable, the main way to populate them is to use the `set!` operator (the same one as for local variables). In this case, the first argument may be any legal `get` or `dot` expression on a table.

Lastly, tables size can be checked with `length`, a list of keys can be retrieved with `table-keys`, and two or more tables can be combined with `concat` (if any of the tables have keys in common, the last table in the argument list takes priority).

2.3.4 Quoting

"Quoting" refers to the process of converting Fn source code into native Fn data. This allows us to easily process and manipulate Fn source code using the same facilities as for normal data.

Quoting is the secret sauce that makes Fn's macro system work. It's the main reason why Fn has the syntax it has.

The `quote` special operator has syntax:

```
(quote <expr>) ;; or, equivalently
'<expr>
```

where `<expr>` can be any expression (in fact, it need not be a legal expression by itself). These two notations are exactly the same. The interpreter expands the second into the first before evaluation.

The value returned by quote is guaranteed to only consist of lists, symbols, numbers, and strings. We refer to the latter three as **atoms**. Here are some examples:

```
'(a b c) ;; returns ['a 'b 'c]
'"string" ;; returns "string"
'(+ a (/ x 2)) ;; returns ['+ 'a ['/ 'x 2]]

''quot ;; is equivalent to
(quote (quote quot)) ;; which returns ['quote 'quot]
```

Note that `<expr>` only needs to be syntactically valid (i.e. not freak out the parser). Illegal expressions can be quoted just fine:

```
'() ;; returns [] (the empty list)
'(2 (3 4)) ;; returns [2 [3 4]]
'(quote) ;; returns ['quote]
```

This makes `quote` very handy for creating nested lists of atoms. (`quote` also has a big sister named `quasiquote`, which is covered in the section on macros, and allows for much more flexibility).

`quote` is also the primary way to create symbols. As noted in subsection 2.3.1, this can lead to problems when we want a symbol whose name is a syntactically valid number. Adding an escape character to the symbol name designates to the parser that the token should be read as a symbol rather than a number. In fact, we can even use this trick to give variables numbers for names:

```
;; probably don't do this
(def \2 3)
2  ;; returns 2
\2 ;; returns 3
```

My recommendation: just don't use symbol names that are syntactically legal numbers.

2.4 TODO Control Flow and Functions

2.4.1 Conditional Execution

The conditional control flow primitives are `if` and `cond`.

`if` takes exactly three arguments: a test expression, an expression to evaluate if the test is true, and an expression to evaluate if the test is false. In Fn, `nil` and `false` are considered to be false values, while all others are treated as `true`.

```
;; if and cond syntax

;; cond takes pairs of expressions and consequences
(cond
  false 1
  nil   2
  'foo  3
  true  4)
; => returns 3, because 'foo is the first true value

(defn fizzbuzz (x)
  (cond
    (= (mod x 15) 0) 'FizzBuzz
    (= (mod x 5) 0)  'Buzz
    (= (mod x 3) 0)  'Fizz
    true             x))
(fizzbuzz 6)  ; = Fizz
(fizzbuzz 7)  ; = 7
(fizzbuzz 45) ; = FizzBuzz
(fizzbuzz 65) ; = Buzz
```

`cond` is an alternative conditional syntax which is analogous to "if/else if" blocks in other programming languages. `cond` takes pairs of arguments and treats the first one as a test. If the test is true, it returns the result of the second argument in the pair. Otherwise it proceeds to the next pair, returning `nil` if the end is reached.

2.4.2 Creating Functions

Functions are created using `fn`.

A short syntax is also provided for creating functions via the dollar sign, which expands into a `dollar-fn` special form.

For example:

```
(fn (x) (* x x))
$(* $ $)
(dollar-fn (* $ $))
```

All three of the above take in a single argument and square it. Note that dollar-fn uses `$` (or equivalently, `$0`) for the name of the first parameter. (Other positional parameters can be accessed with `$1`, `$2`, and so on). See subsection 2.4.5 for more details.

`fn` on the other hand has an explicit parameter list. The syntax for parameter lists is this:

```
param-list      ::=  '(' <req-param>* <opt-param>* <var-params>? ')'
req-param       ::= <identifier>
opt-param       ::= (<identifier> <init-form>)
var-params      ::= <var-list-param> <var-table-param>?
                  | <var-table-param> <var-list-param>?
var-list-param  ::= '&' <identifier>
var-table-param ::= ':&' <identifier>
```

In other words, parameter lists consist of zero or more required parameters, zero or more optional parameters, and optionally end with variadic table and list arguments.

Each of these parameters has an associated identifier (i.e. a symbol that is a legal name). In the function's body, the respective arguments are bound to these names. See subsection 2.4.3 for information about how argument lists are processed during function calls.

2.4.3 Function Calls

Fn allows arguments to be named in function calls very similarly to Python. Named arguments are passed using keywords, which are simply symbols whose names begin with `:`. These symbols are not legal identifiers, so their appearance in function calls is unambiguous. We also place the restriction that positional arguments may not follow named ones. (Believe me, I tried to make it work without that, and it's a mess at every level).

First we will deal with the case where there are no variadic parameters. See the following example.

```
;; this function has 3 positional parameters, the last of which is optional
(defn arg-demo (x y (z 2))
```

```
  (* z (+ x y)))

;; here are a couple of ways we could call this function
(arg-demo 2 3)            ; x = 2, y = 3, z = 2, result = 10
(arg-demo 2 3 4)          ; x = 2, y = 3, z = 4, result = 20
(arg-demo :x 2 :y 3)      ; x = 2, y = 3, z = 2, result = 10
(arg-demo :z 2 :y 3 :x 2) ; x = 2, y = 3, z = 2, result = 10
(arg-demo :z 3 1 2)       ; error! positional argument following named argument
```

To be precise, function parameters (still considering the case where there are no variadic parameters) are bound using the following procedure:

- the unnamed arguments are bound to positional parameters in order
- the named arguments are bound to their respective parameters, raising an error if any duplicates or unrecognized names are found
- unbound optional parameters are set to their default values. If any required parameters remain unbound, an error is raised

Now, variadic arguments change some of the rules. We have two types of variadic parameters in Fn: variadic tables, and variadic lists.

For tables, the semantics are very simple. Functions with a variadic table parameter can accept any named argument, not just the names corresponding to their functions (duplicated names are still not allowed). Moreover, it's now possible for a named argument to have the same name as a positional argument.

```
;; demo function ignores first arg and returns table
(def var-tab-demo ((x nil) :& tab) ; variadic table arguments denoted with :&
  tab)

(var-tab-demo 0)         ; result = {}
(var-tab-demo :y 2 :x 1) ; result = {'y 1}
(var-tab-demo 1 :x 2)    ; result = {'x 2}
```

As can be seen above, the variadic table is constructed by taking all unrecognized named arguments and inserting them into the table as key-value pairs. Moreover, if a named argument is recognized, but was already provided as a positional argument, then that goes to the variadic table as well.

Variadic lists are analogous to variadic tables, but where those act on trailing named arguments, variadic lists act on trailing positional arguments.

As such, it is impossible to use named arguments while at the same time passing a non-empty variadic list argument, except in the case where there is also a variadic table parameter to catch the trailing arguments.

```
;; demo function ignores first arg and returns list
(def var-lst-demo (x & list) ; variadic lists denoted with &
  list)

(var-lst-demo 0 1 2)  ;=> [1 2]
(var-lst-demo 0)      ;=> []
(var-lst-demo 0 :x 2) ;=> syntax error
(var-lst-demo :x 0)   ;=> []
(var-lst-demo :x 0 1) ;=> syntax error

;; demo function using both variadic parameters
(def var-mixed-demo (x & list :& table)
  [list table])

;; names not explicitly in the parameter list get sent to the variadic table
(var-mixed-demo :x 4 :y 2) ;=> [[] {'y 2}]
;; with a variadic table argument, duplicate names are allowed if one is a
;; positional arg:
(var-mixed-demo 'a 'b :x 4 :y 2) ;=> [['b] {'x 4 'y 2}]
;; as always, keywords cannot precede positional arguments
(var-mixed-demo :x 4 :y 2 'a 'b) ;=> syntax error
```

2.4.4 TODO Variable Capture

2.4.5 TODO dollar-fn

2.5 TODO Namespaces and Import

2.5.1 Namespaces and Packages

A namespace is a collection of global variables and macros. They are identified by a name which is required to be globally unique. All Fn code lives inside of some namespace.

Some examples of namespace names are:

```
fn/builtin
fn/repl
project/main
project/subsystem/parser
```

The slashes above are delimiters. In particular, everything to the left of the last slash is called the **package name**, and the part to the right is called the **short name**. For instance the namespace `project/subsystem/parser` has package `project/subsystem` and short name `parser`.

We also have the concept of a **subpackage**. In the above examples, `project/subsystem` is a subpackage of `project`. That is, the subpackages of a `<pkg>` are all packages with names of the form `<pkg>/...`.

2.5.2 Namespace Determination

When a file is interpreted, the short name of the namespace is set to the stem of the filename. There is no way to change this. The package, however, may be anything at all, and is specified by putting a `package` declaration at the top of the file:

```
(package myproject/util)
```

If no package declaration is present, the package `fn/user` is used by default.

Unlike other special forms in Fn, there can only be one package declaration per file, and it must come at the beginning of the file, although it can be preceeded by comments and whitespace.

The REPL allows the current namespace to be changed interactively. By default it uses the namespace `fn/repl`.

2.5.3 Importing Namespaces

Importing a namespace means taking its bindings into the current namespace.

2.5.4 Builtin Namespace

When a new namespace is created, an unqualified import of `fn/builtin` is automatically performed.

2.5.5 Global Names

After a namespace has been imported once, its bindings can be referenced even without importing it explicitly. This is done by using symbols whose names are structured like with `/<namespace>:<symbol>`. For example, `/fn/builtin:map` refers to the function `map` in the `fn/builtin` namespace.

2.6 TODO Macros

2.6.1 Macro Basics

2.6.2 Quasiquotation

2.6.3 Variable Capture and `gensym`