Fn Language Specification

Jack Pugmire

December 12, 2021

Contents

1 About this Document

This document fully specifies the syntax and semantics of the Fn programming language. Any part of it may be changed at any time. Since the interpreter is still unfinished, its actual behavior may differ from what's described here.

Section 2 contains a description of the semantics of Fn. It's meant to read like a user's manual, and should cover everything about the core language when it's done. I'm trying to make this readable.

After this comes the technical part of the document. It's a big old list of all the types of expressions in Fn with precise descriptions of their behaviors.

Following this is the lexical analysis section. This rounds out the specification by giving a character-by-character description of Fn syntax. I'm planning to rewrite this at some point. It's in rough shape after repeated cut-and-pastes from different documents.

At the end is some other miscellaneous stuff I copied from old design documents because I didn't want to delete it from the main repository yet.

There will be more sections (and subsections!) in the future.

2 Language Description

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 TODO 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`.

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 functino 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 Variable Capture

2.4.5 dollar-fn

2.5 TODO Namespaces and Import

2.5.1 Namespaces and Packages

All code in Fn runs inside some namespace, which is used to hold currently visible global variables and macros.

A **namespace** is a collection of macro and variable definitions. Namespaces are identified by a **name**, which is a string not containing any slashes, and a **package**, which is a string representing a logical collection of namespaces. Finally, the symbol `<package>/<name>` is called the **identifier** or **ID** of the namespace. This ID is required to be globally unique.

Examples of Namespace IDs:

```
fn/builtin              ; package is "fn", name is "builtin"
fn/internal/io          ; package is "fn/internal", name is "io"
my-project/util/linalg  ; package is "my-project/util", name is "linalg"
my-project/model        ; package is "my-project", name is "model"
```

When evaluating code from a file, the namespace name will always be the stem of the file. The package can be set via a package declaration, see 2.5.3.

The default REPL namespace is `fn/interactive`. Fn source code passed in as a command line argument is also evaluated in this namespace.

2.5.2 Import

The `import` special form allows definitions from an external namespace to be copied into the current one. The syntax for import looks like this:

```
(import <namespace-id>)                  ; invocation 1
(import <namespace-id> :as <alias>)      ; invocation 2
(import <namespace-id> :no-prefix true)  ; invocation 3
```

Say we have a namespace `foo/bar/baz` containing variables named `bob` and `alice`:

```
;;; baz.fn
(package foo/bar)
(def alice "Alice")
(def bob "Bob")
```

We have three ways to import this namespace, shown above. All three cause the definitions from `foo/bar/baz` to be copied into the current namespace. However, in each case the created bindings will have different names. The three cases are illustrated below:

```
;;; main.fn

;; invocation 1
(import foo/bar/baz)
; variables look like this:
baz:alice
baz:bob

;; invocation 2
(import foo/bar/baz :as b)
; variables look like this:
b:alice
b:bob

;; invocation 3
(import foo/bar/baz :no-prefix true)
; variables are imported directly (no colons)
alice
bob
```

2.5.3 Package Declarations

The first expression of a file (not counting comments) may be a **package declaration**. These have the form `(package <package-name>)`, where <package-name> is a symbol.

2.5.4 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`

3 Formal Semantics

This section is a formal description of every type of expression in Fn. It is currently incomplete and inaccurate. I don't know why you'd want to look at it.

```
program ::= expr* expr ::= immediate
        | variable
        | special-form
        | function-call
        | macro-call
```

3.1 Immediate Expressions and Variables

Syntax:

```
immediate ::= boolean
          | nil
          | number
          | string
variable ::= non-special-symbol
```

An immediate expression is a literal representing a constant value. On evaluation, immediate expressions immediately return the value they represent.

Variables are represented by non-special symbols, (where special symbols are those naming special forms, boolean values, or nil). If there exists a binding in the current environment for the provided symbol, then its value is returned. Otherwise an exception is raised.

3.2 Special forms

Special forms are so called because they have different semantics than function or macro calls.

3.2.1 and

Syntax:

```
and-expr ::= "(" "and" expr* ")"
```

Expressions are evaluated one at a time until a logically false value is encountered, then returns `false`. If the end of the list is reached, returns `true`.

3.2.2 cond

Syntax:

```
cond-expr ::= "(" "cond" cond-case+ ")"
cond-case ::= expr expr
```

For each cond-case, the following is done:

- evaluate the first expression
- if the first expression is logically true, return the value of the second expression
- otherwise, proceed to the next cond-case.

If the end of the list is reached, returns `nil`.

3.2.3 def

Syntax:

```
def-expr ::= "(" "def" identifier expr ")"
         | "(" "def" func-proto expr+")"
func-proto ::= "(" identifier param-list ")"
```

Create a (global) binding in the current namespace. The first syntax binds the identifier to the value of the expression. The second syntax creates a function with the specified name and parameter list and the expressions as its body. In either case, if the identifier is already bound, an exception is raised.

Returns `null`.

3.2.4 TODO defmacro

Syntax:

```
defmacro-expr ::= "(" "defmacro" identifier param-list expr+ ")"
```

3.2.5 defn

3.2.6 do

Syntax:

```
do-expr ::= "(" "do" expr* ")"
```

Evaluates provided expressions one at a time, returning the value of the last one, or `null` if no expressions are given.

3.2.7 do-inline

3.2.8 TODO dot

Syntax:

```
dot-expr ::= dotted-symbol
           | "(" "dot" symbol+ ")"
```

This operator is usually used with the dotted-symbol syntax, e.g. `table.key`.

The first symbol (leftmost in the inline notation) must name a variable bound to a table. The next symbol is used as a key to access an element of the table. If additional symbols are provided, then they are used as keys to recursively descend into a tree of tables. An exception is raised if one of the keys is invalid or if an attempt is made to access an object which is not a table.

3.2.9 dollar-fn

Syntax:

```
dollar-fn-expr ::= "(" "dollar-fn" expr ")"
                 | "$(" expr+ ")"
                 | "$[" expr+ "]"
                 | "${" expr+ "}"
                 | "$`" form
```

Creates an anonymous function which evaluates the provided expression. With the "$" syntax, this is the expression after the dollar sign. (The only expressions which may follow are parenthesized forms, quasiquote forms, or list/table expressions).

Within the provided expression, variables named `$N` where N is a nonnegative integer, are bound to the corresponding positional parameters starting from 0. In addition, `$` is bound to the first parameter `$0` and `$&` is used for a variadic parameter.

The parameter list for the created function accepts as many positional parameters as the highest value of N and a variadic parameter only if `$&` appears in the expression. (This is accomplished by performing code-walking, including macroexpansion, before compiling the `dollar-fn`).

3.2.10 if

Syntax:

```
if-expr ::= "(" "if" test-expr expr expr ")"
test-expr ::= expr
```

Evaluates test-expr. If the result is logically true, evaluates the second argument, otherwise evaluates final argument, returning the result.

3.2.11 TODO import

Syntax:

```
import-expr ::= "(" "import" import-designator
                             [:as identifier] ")"
import-designator ::= string | symbol
```

Import bindings from another namespace. Every variable and macro definition from the target namespace is copied into the current namespace. The newly created bindings have names of the form `namespace-name:variable-name`. Since it is illegal to create variables whose names contain the colon character, this ensures that no name collisions can occur, provided there is not already a namespace imported with this name.

If an identifier is provided via the `:as` argument, then that is used instead of the namespace name.

When `import` is used on a namespace that is not already loaded, the interpreter checks for appropriate files in the search path, then compiles and loads them as necessary.

1. Future changes

    - Support for unqualified imports will be added (i.e. imports without the `namespace~name:` prefix).
    - Import may be changed so as to not recursively import imports from the package named. With current behavior, we could get names like `ns1:ns2:function` when `ns1` imports `ns2`. This is mostly harmless, but we'd rather not overclutter namespaces if only for memory concerns. Also, it could definitely cause trouble if used with unqualified imports.
    - Along with unqualified imports, we may add the notion of exports to namespaces, so that only certain bindings in a namespace are even made available to external namespaces. This is probably good for convenience for users of a library, but it also can give implementors some peace of mind knowing that their internal functions won't be called directly by user code.

3.2.12 fn

Syntax:

```
fn-expr ::= "(" "fn" "(" param-list ")" expr+ ")"
```

Creates a function object using the provided parameter list and function body.

Functions have full variable capture semantics. Among other things, this means that captured variables are shared among functions, so that if one function mutates the variable, this change is reflected in the other functions accessing the variable.

3.2.13 let

Syntax:

```
let-expr ::= "(" "let" let-pair+ ")"
let-pair ::= identifier expr
```

Extends the current local environment. For each let-pair initially binds the provided identifier to null. Then, in the order provided, each expression is evaluated and the binding is updated to the resultant value.

The initial null-binding allows definition of recursive and even mutually recursive functions. Care must be taken because this null binding will shadow existing variables with the same name.

Returns null.

3.2.14 TODO letfn

3.2.15 or

Syntax:

```
or-expr ::= "(" "or" expr* ")"
```

Evaluates provided expressions one at a time until a logically true value is obtained. Then returns `true`. If the end of the list is reached, returns `false`.

3.2.16 quasiquote

Syntax:

```
quasiquote-expr ::= "‘" form
                  | "(" "quasiquote" form ")"
```

First, creates a fn object corresponding to form just like quote. Before returning the form, the following transformation is done:

- The form is walked like a tree.
- When an unquote-expr is encountered, instead of descending into it, evaluate its argument and insert the result into the tree at that point.
- When an unquote-splicing form is encountered, instead of descending into it, evaluate its argument. If the result is not a list or if this is root of the tree, raise an error. Otherwise, splice the elements of the list inline into the tree at this point.
- Along the way, we keep track of all symbols whose names begin with a hash character "#". For each unique hash symbol, a single gensym is created, and the hash symbols are replaced by the gensyms in the final expansion. For example, see the following code snippet:

```
‘(#sym1 #sym2 #sym2) ; is the same as
(with (sym1 (gensym)
       sym2 (gensym))
 [sym1 sym2 sym2])
```

Note that #-symbols have lexical scope, i.e. they are shared by quasiquote forms occurring within the outer form.

IMPLNOTE: the semantics of #-symbols may be too complicated for what they get us. Might be better to just make the developer deal with `gensym` directly, or to make #-symbols a feature of defmacro.

3.2.17 quote

Syntax:

```
quote-expr ::= "’" form
             | "(" "quote" form ")"
```

Returns the syntactic form as an fn object (a tree of atoms and lists).

3.2.18 unquote

Syntax:

```
unquote-expr ::= "," expr
             | "(" "unquote" expr ")"
```

Emits an error unless encountered within a quasiquote form.

3.2.19 unquote-splicing

Syntax:

```
unquote-splicing-expr ::= ",@" expr | "(" "unquote-splicing" expr ")"
```

Emits an error unless encountered within a quasiquote form.

3.2.20 TODO set!

Syntax:

```
set!-form ::= "(" "set!" place expr ")"
place ::= identifier
      | dot-expr
      | get-form
get-form ::= "(" "get" expr+ ")"
```

3.2.21 with

Syntax:

```
with-expr ::= "(" with-bindings expr+ ")"
with-bindings ::= "(" (id expr)* ")"
```

Behaves like `let`, but rather than operating on the enclosing lexical environment, instead creates a new child environment and adds bindings to that, then evaluates the provided expressions in the newly created environment.

Note that this is how `let` works in most LISP-like languages.

3.3 Function Calls

Syntax:

```
function-call ::= "(" func argument-list ")"
```

where `func` may be any expression other than a reserved symbol.

First, the function and then all the arguments are evaluated from left to right.

Arguments are bound to parameters as follows:

Positional arguments are bound to the function's parameters in the order provided. If there are more positional arguments than parameters, a list of the extras are bound to the variadic list parameter if one exists. If not, an error is generated.

After this, keyword arguments are bound to parameters by name. If two keyword arguments have the same name, an error is raised. If the name isn't one of the function's parameters, or if it names a parameter already provided by a positional argument, it is added to the variadic table parameter if one exists. If not, an error is raised.

If any parameters without default values remain unbound, an error is raised.

Then, the function is called. Foreign functions have behavior determined by the external code they call. Ordinary functions work by switching back to the lexical environment in which they were created, binding parameters as local variables, and executing the function body.

3.4 Macro Calls

Syntax:

```
macro-call ::= "(" macro-name form* ")"
macro-name ::= identifier | dot-expr
```

The expressions for macro arguments aren't evaluated, but are converted to data and passed to the macro function as arguments. The resultant value is treated as code and evaluated.

In order to prevent ambiguity, macros do not recognize keyword arguments. A keyword will be passed to the macro as a positional argument containing the keyword symbol.

4 Lexical Analysis

The component which processes Fn source into lexical tokens is called the **scanner**. Conceptually, we put in a sequence of bytes and get back a sequence of tokens.

Tokens are dividide into two groups: fixed width and variable width. The fixed width tokens are:

'(left paren

') right paren

'[left bracket

'] right bracket

'{ left brace

'} right brace

'\" quote

'` backtick

',@ comma at

', comma

'$` dollar backtick

'$(dollar paren

'$[dollar bracket

'${ dollar brace

Fixed width tokens are generated when the scanner encounters one of the corresponding quoted strings above.

Variable width tokens are:

<**number**> numeric literal

<**string**> string literal

<**symbol**> symbol

<**dot**> dot expression (2 or more symbols separated by '.')

4.1 Comments

A comments begins with the unescaped character **';'** and end at the end of the line.

Comments are skipped over without generating a token.

4.2 Numbers, Symbols, and Dots

Numbers are defined according to regular expressions as in the following pseudo-BNF grammar:

```
<number> ::= "[+-]?(<dec>|<hex>)"
<dec>    ::= "[0-9]+\.?<exp>?"
           | "[0-9]*\.[0-9]+<exp>?"
<exp>    ::= "[eE][+-]?[0-9]+"
<hex>    ::= "0[Xx][0-9A-Fa-f]+\.?"
           | "0[Xx][0-9A-Fa-f]*\.[0-9A-Fa-f]"
```

A symbol is defined to be a sequence of symbol characters and escaped characters which is not a number. Symbol characters are all printable ASCII characters other than whitespace or those contained in the string `";(){}[]\"\\'‘,."`. An escaped character is a sequence of two characters, the escape `'\\'` followed by an arbitrary character. When the symbol is internalized, the escape character is ignored, so its name will contain only the second character of the escape sequence.

If an escape character is followed by EOF, a scanning error is raised.

Note that by injecting an escape character, one may cause numbers to be treated as symbols. Adding an escape in front of any normal character normally no effect, but in this case, it causes the number reader to fail, so the number's characters will read as a symbol.

Finally, a dot token is a symbol token which contains the dot character `'.'`, subject to some additional restrictions. Dots can not occur in the first or last position of the string, and it can not contain successive dots. In addition, the substring before the first dot may not be a syntactically valid number. If any of these conditions is violated, a scanning error is raised.

4.3 TODO String Literals

String scanning starts when the scanner encounters the (unescaped) character `'"'` and ends when it encounters an unescaped `'"'`. Along the way, all bytes encountered are read verbatim, except for the escape character '\, which is followed by an escape sequence. The entire escape sequence is read into the string according to the following table [3]:

'\" single quote

[3] These string escapes are mainly the same as the ones in C.

'\"' double quote

'\?' question mark

'\a' ASCII bell

'\b' backspace

'\f' form feed

'\n' newline

'\r' ASCII carriage return

'\t' tab

'\v' vertical tab

'\NNN (NNN is a 1- to 3- digit octal number) byte NNN (octal)

'\xNN (NN any two-digit hex number) the byte NN (hexadecimal)

'\uNNNN (NNNN any 4-digit hex unicode code point) unicode code point (2 bytes)

'\UNNNNNNNN (NNNNNNNN any 8-digit hex unicode code point) unicode code point (4 bytes)

4.4 Source Encoding

Currently, Fn only supports ASCII encoded text files. Behavior on other/extended encodings is undefined. In the future, Fn will be extended so that UTF-8 characters can appear in strings and symbols.

5 TODO Built-in Functions

Fn provides a number of built-in functions in the namespace `fn.builtin`. Whenever a new namespace is created, it automatically inherits all the bindings from `fn.builtin`, so these bindings are always available as global variables.

Builtin functions are split into primitive and nonprimitive functions. This classification is not overly rigorous, but the rule is that primitive functions expose core language functionality which cannot be used any other way. On the other hand, all nonprimitive functions could theoretically be implemented in Fn source code by using the primitives.

5.1 Primitive Functions

apply (obj arg0 arg1 & args) Call obj as a function. Positional arguments for the call are generated by taking a list of all but the last two arguments (to apply), and concatenating that with the second-to-last argument, which must be a list. The last argument (to apply) is a table of keyword arguments.

gensym () Generate a symbol with a guaranteed unique id. This is used for macro writing.

symbol-name (x) get the name of a symbol as a string

length (obj) Depending on the type of obj, returns

- the length of a string in bytes,
- the number of elements in a list, or
- the number of keys in a table or namespace.

concat (seq0 & seqs) Concatenate strings or lists. All the arguments must be of the same type.

nth (n seq) Get the nth element of a list or string. If seq is a string the result will be a string of length 1 containing the (n-1)th byte of the string.

= check for equality

same? check for equality

not logical not

Type Checkers:

bool?

function?

int?

list?

namespace?

number?

null?

string?

symbol?

table?

Lists and Strings:

List (& objs) Create a list of the given objects

empty? (obj) Equivalent to `(= obj [])`.

cons (hd tl) Create a new list by prepending hd to tl.

head (list) get the first element of a list. Error on empty.

tail (list) drop the first element of a list. Returns empty on empty.

Tables and Namespaces:

Table (& kv-pairs) Create a table. The argument list is a sequence of pairs consisting of keys followed by values.

get (key obj) access a field from a table or namespace

get-keys (obj) get a list of keys from a table or namespace

has-key? (obj key) get a list of keys from a table or namespace

Arithmetic:

+

-

*

/

**

<

>

<=

>=

floor

ceil

5.2 Nonprimitive functions

(Note: a user type can implement any or all of these functions by adding methods for them. Sorry, that isn't documented yet).

All of these are non-destructive.

- has-key? (table)
- concat (& seqs)
- reverse (seq)
- insert (elt n seq)
- append (elt seq)
- prepend (elt seq)
- sort (seq (ascending true))
- sort-by (fun seq (ascending true))
- head (seq)
- tail (seq)
- nth (n seq)
- take (test seq)
- drop (test seq)
- take-while (test seq)
- drop-while (test seq)
- split-at (n seq)
- split-when (test seq)
- group (n seq)

- group-by (key seq)
- subseq (start end seq)
- dedup (seq) [remove duplicates]
- replace (n elt seq)
- empty? (seq)
- contains? (seq)
- length< (seq n) [compare lengths w/o nec. computing the thing]
- length> (seq n)
- length<= (seq n)
- length>= (seq n)
- map (fun seq0 & seqs)
- fold (fun init seq0 & seqs)
- filter (test seq)
- every? (test seq)
- any? (test seq)

6 Future Extensions

The version of Fn described in this document is version 0.1. The language **will** change before version 1.0. In particular, I have to, **have to**, add pattern matching.

Items marked with (Proposed) below are things that might not make it into the final language.

6.1 Pattern Destructuring in `def` and `let`

Once pattern matching is added to Fn, `let` and `def` will be able to do automatic destructuring on the pattern to bind variables. For instance,

```
(def [x y] [1 2])
```

binds `x` to `1` and `y` to `2`.

When matching fails, an error will be raised.

6.2 (Proposed) #-Syntax for Collections

So, we turn `#` into a syntax character and give `#()`, `#[]`, `#{}`, etc. syntactic meanings.

There are a couple of options for how to do this:

- `#()` could expand to `(values)` (if we implement multiple values as proposed), or to a set or tuple type.
- `${}` could expand to `(Set)` or `(Vector)`, or if we add typed tables, `#{type ...}` could be used to construct types (e.g. `(construct type ...)`).
- `#[]` could expand to `(Vector)` or `(values)`.
- Alternatively, if we lean into lazy data structures, then the `#` structures could be the respective lazy equivalents

I'm most likely to set `#[]` to `(Vector ...)`. Using `#{}` for `(Set ...)` would also make sense, but if typed tables get added, those will be more important for sets. If multiple values get added, then the `#()` syntax should really be used for that.

Drawbacks: this would make hash into a special character as opposed to just a symbol constituent.

6.3 Global Names for Definitions

A global variable name consists of a forward slash, the full namespace name, a colon, and an identifier in that namespace, in that order. For example:

```
/fn/builtin:+
```

is the global name for the builtin `+` function.

This is implemented as follows:

- make it illegal to name namespaces beginning with a slash
- make it illegal to create variables whose names contain colons. (Alternatively, can make it illegal to create variables whose names start with a slash).
- variable names that have this syntax are evaluated a little differently

To get the global name of a symbol, we'll provide a builtin function called `global-name`, e.g. `(global-name '+)` is `'/fn/builtin:+`.

6.4 (Proposed) ,:-Syntax for Global Names in Quasiquote

The syntax `,:var` expands `(unquote (global-name 'var))`. This will be necessary for all non-builtin functions included in macro expansions, (and should be used for builtin functions too).

6.5 (Proposed) Multiple Return Values

Idk, these are useful in Matlab and Haskell has this sort of thing going on with its tuples.

So we create a multiple values object with `(values x y ...)`. These objects are tricky; as soon as they get used they collapse to a single value (in this case the result of expression `x`).

However, when a multiple values object is returned to a `let` or `def`, then we can use destructuring to access the additional values. For example:

```
(defn my-fun () (values 1 2 3))
(def x (my-fun)) ; sets x = 1
(def (values y z) ; sets y = 1, z = 2
(def (values y z w) ; sets y = 1, z = 2, w = 3
```

Together with #-syntax for values, this might be a very useful feature. It would look like this:

```
(def #(y z) (my-fun)) ; sets y = 1, z = 2
(def (values y z w) (my-fun)) ; sets y = 1, z = 2, w = 3
```

6.6 Schemas, Metatables, and Table Calls

Tables are gonna get a metatable like in Lua. This will simply be a second, invisible table, accessible via a `metatable` builtin function.

The `metatable` may contain these two things (more may be added):

- an **on-call function**. If set, this allows the table to be called like a function by invoking the on-call function on the given arguments.
- a **schema** for the table

The on-call function is self-explanatory:

```
(def tab { 'value 27 })
(set! (get (metatable tab) 'on-call)
      (fn (x) (+ x tab.value)))
(tab 42) ;=> 69
```

Schemas are objects describing the following information:

- a schema name that can be used to do pattern matching on this kind of object
- optional type annotations for values in the table
- pattern matching functions
- (Proposed) protocols implemented by the table. (Note: this may belong directly in the metatable?)

So, giving a table a schema provides:

- data validation
- accessors
- pattern matching

All of this with very little overhead.

So giving a table a schema is a big deal. It basically promotes the the object from a table to a more complex type. For this reason, we call tables with non-nil schemas **structures**. It is intended that structures will be used extensively in Fn as the main way to organize data.

Through use of variable capture, functions contained in a table may access or mutate that table. This makes it possible to implement full, stateful object-oriented programming using structures containing functions, although I don't know why you'd want to.

However, we functional programmers still benefit from structures containing functions. In particular, this is a handy way to simultaneously save typing and avoid namespace pollution. For instance, consider:

```
;; database structure has a send function
(my-db.query my-query)
;; vs. a global function to do it
(db-query my-db my-query)
```

Moreover, we see this gives us something Python calls "duck typing". Any table with an appropriate query function will work in the first expression above. I think that's pretty neat.

You could even implement ADTs a la Haskell using schemas, but I don't expect they'd be nearly as useful without that sweet, sweet type system.

6.7 (Proposed) Table Immutability Rules

It would be really great to have some guarantee of a table's immutability.

Mutable tables are great for initialization and certain types of imperative algorithms. However, immutable tables are better for functional programming, and we like that.

I propose the following changes:

- tables are immutable by default
- mutable tables may be created via `(M-Table ...)`, and have mutable metatables
- builtin functions are provided to copy mutable tables to immutable ones and vice versa

In addition, it may be possible to allow mutable tables to be made immutable in certain situations, in order to save the copying operation. In order to prevent this from wreaking havoc, it might be best to provide some builtin function that exposes this functionality. E.g. `(build-table x)` where `x` is a function that takes a mutable table, mutates it, and returns it at the end. Provided the return value is the same table as was put in, it is then converted to a mutable table directly, with no copying.