Joxa Documentation

Release v0.1.0

Eric B Merritt

February 24, 2016

Contents

Joxa is a small semantically clean, functional Lisp. It is designed as a general-purpose language encouraging interactive development and a functional programming style. Like other Lisps, Joxa treats code as data and has a full (unhygienic) macro system. Joxa has an advantage over other Lisp in that it runs on the Erlang Virtual Machine and inherits all of the benifits for concurrency and distribution provided by that platform.

Contents:

CHAPTER 1

Introduction

Joxa is a Lisp designed to support general programming with good declarative data description facilities. Joxa is intended to be used as a powerful, light-weight alternative for Erlang for any program any system where a language like Erlang is prefered. Joxa is implemented as a compiler and library, written in itself while still making extensive use of the Erlang libraries and infrastructure.

Joxa is free software, and is provided as usual with no guarantees, as stated in its license. Further information is a available on the Joxa website, www.joxa.org.

1.1 Examples

The very first thing that everyone wants to see when exploring a new language is what it looks like. So to feed that need lets jump right into some examples and descriptions.

1.1.1 Sieve of Eratosthenes

Here we see the Sieve of Eratosthenes implemented as a Joxa Namespace

```
(ns sieve-of-eratosthenes
        (require lists)
        (use (joxa.core :as core :only (!=/2))
             (erlang :only (rem/2 +/2))))

(defn sieve (v primes)
  (case v
    ([] primes)
    ((h . t)
      (sieve  (lists/filter (fn (x)
                               (!= (rem x h) 0)) t)
              (+ primes 1)))))

(defn+ sieve (v)
  (sieve (lists/seq 2 v) 1))
```

Now that we have seen the entire namespace lets start breaking it down

```
(ns sieve-of-eratosthenes
    (require lists)
        (use (joxa.core :as core :only (!=/2))
            (erlang :only (rem/2 +/2))))
```

The very first thing that must occur in file is the namespace special form. You can call a fully qualified macro to create the namespace, but that macro **must** create the namespace first. A namespace is defined with the ns special form.

The ns special form consists of three main parts (we will go into greater detail later in this document). The first part is the name of the namespace. Which is an atom that identifies that namespace. Though is not a requirement its generally a good idea for the namespace name match the file name.

The second part of the ns special form is the *require* form. The require form provides a list of those namespaces that will be used by the namespace. This is not strictly required (namespaces that are used in the namespace being defined but not required will be automatically required). However, it **is** very good documentation and I encourage you to require all your namespaces.

The third part is the use form. The use form allow you to import functions into the namespace. So you do not have to write the fully qualified name. This is especially useful for functions and macros defined as operators. Don't go crazy with it though. It is a spice that should be used only where it enhances clarity.

Any number of require and use statements can appear in the namespace in any order.

Next we see the function definition

```
(defn sieve (v primes)
  (case v
    ([] primes)
    ((h . t)
      (sieve  (lists/filter (fn (x)
                              (!= (rem x h) 0)) t)
              (+ primes 1)))))
```

We define a function called *sieve* that takes two arguments. The argument *v* and, next, the argument *primes*. We then have a single *case* expression that forms the body of the function. A case expression allows the author to do pattern matching on the second clause of the expression. While he rest of the clauses identify patterns and what will be evaluate based on the form of the output of the second clause. In this example, you can see that an empty list will return the argument *primes* unchanged while a cons cell will result in a recursive call of *sieve*, a call to the erlang module *lists* with an anonymous function. You can all see the use of the functions (not defined in the namespace) that we imported into the namespace with the use form.

Finally, we define our public api

```
(defn+ sieve (v)
  (sieve (lists/seq 2 v) 1))
```

There are two types of function definitions in Joxa; *exported* and *unexported* functions. Exported functions are available outside of the namespace while unexported functions are only available inside the namespace itself. The difference in declaration is the use of *defn+* for exported functions in place of *defn* for unexported functions. In this example you see us call the unexperted sieve function and the use again of the lists erlang module. In Joxa, functions must be defined before they are used. So the unexported *sieve/2* had to be defined before the exported *sieve/1* function.

1.1.2 Fibonacci

Here we see the Fibonacci implemented as a Joxa Namespace

```
(ns fibonacci
   (use (erlang :only (>/2 -/2 +/2))))

(defn+ fibo (n)
  (case n
    (0 0)
    (1 1)
    (_ (when (> n 0))
      (+ (fibo (- n 1))
         (fibo (- n 2))))))
```

CHAPTER 2

Install

If you are using Ubuntu its easiest to install Joxa from the PPA. This will get everything setup for you. For other distributions you simply need to drop the Joxa executable in your path. That executable is an `escript`. An escript is basically a binary executable. However, it depends on the existence (on your machine) of the Erlang Virtual Machine. So either install that now from source or install it from the packaging system on your distribution.

If you are using Windows, install a recent Erlang (R15B or newer), add Erlang's bin directory to your path, and drop `joxa.cmd` in there too.

How to properly install your project. Ideally, your project should be installable via a common (simplistic) method: PyPI for Python, PEAR for PHP, CPAN for Perl, RubyGems for Ruby, etc.

CHAPTER 3

Quick Start

A quickstart guide which walks new users through building a working application. This piece is critically important, as it will determine what new users think of your project. Having a good quickstart guide shows users that you care for them, and ensures that both you (as a maintainer) and your users have a good understanding of your project.

CHAPTER 4

The Joxa Language

4.1 Special Forms

4.1.1 let*

```
(let* (val val-expr ...) expr ...)
```

4.1.2 try*

```
(try* expr (catch (error-class error-type) catch-expr ...))
```

4.1.3 case

```
(case expr
    (pattern <optional-guard> expr ...)
    ...)
```

4.1.4 receive

```
(receive <optional-after>
    (pattern <optional-guard> expr ...)
    ...)
```

4.1.5 do

```
(do expr ...)
```

4.1.6 binary

Segment

Each segment has the following general syntax:

```
<< value (:size size) <type specifier list> >>
(binary value (:size size) <type specifier list>)
```

Any part of the binary except the *value* may be left out and receive sane defaults.

Default values will be used for missing specifications. The default values are described in Defaults.

Used in binary construction, the *value* part is any expression. Used in binary matching, the *value* part must be a literal or variable. You can read more about the *value* part in the section about constructing binaries and matching binaries.

The *size* part of the segment multiplied by the unit in the type specifier list (described below) gives the number of bits for the segment. In construction, *size* is any expression that evaluates to an integer. In matching, *size* must be a constant expression or a variable.

The type specifier list is a list of type specifiers separated by hyphens.

Type The type can be *:integer*, *:float*, *:binary* or *:bitstring*.

Signedness The signedness specification can be either *:signed* or *:unsigned*. Note that signedness only matters for matching.

Endianness The endianness specification can be either *:big*, *:little*, or *:native*. Native-endian means that the endian will be resolved at load time to be either big-endian or little-endian, depending on what is "native" for the CPU that the Erlang machine is run on.

Unit The unit size is given as unit:IntegerLiteral. The allowed range is 1-256. It will be multiplied by the Size specifier to give the effective size of the segment. In R12B, the unit size specifies the alignment for binary segments without size (examples will follow).

Example

```
(binary X (:size 4) :little :signed :integer (:unit 8))
<<X (:size 4) :little :signed :integer (:unit 8)>>
```

This element has a total size of 4*8 = 32 bits, and it contains a signed integer in little-endian order.

Defaults

The default type for a segment is integer. The default type does not depend on the value, even if the value is a literal. For instance, the default type in <<3.14>> is *:integer* not *:float*.

The default *size* depends on the type. For *:integer* it is 8. For *:float* it is 64. For binary it is all of the *:binary*. In matching, this default value is only valid for the very last element. All other binary elements in matching must have a size specification.

The default *:unit* depends on the the type. For *:integer*, *:float*, and *:bitstring* it is 1. For *:binary* it is 8.

The default signedness is *:unsigned*.

The default endianness is *:big*.

Constructing Binaries and Bitstrings

This section describes the rules for constructing binaries using the bit syntax. Unlike when constructing lists or tuples, the construction of a binary can fail with a badarg exception.

There can be zero or more segments in a binary to be constructed. The expression <<>> constructs a zero length binary.

Each segment in a binary can consist of zero or more bits. There are no alignment rules for individual segments of type *:integer* and *:float*. For *:binary* and *:bitstring* types without size, the unit specifies the alignment. Since the default alignment for the *:binary* type is 8, the size of a binary segment must be a multiple of 8 bits (i.e. only whole bytes).

```
<<(bin :binary) (bitstring :bitstring)>>
(binary (bin :binary) (bitstring :bitstring))
```

The variable *bin* in must contain a whole number of bytes, because the binary type defaults to (:unit 8). A *badarg* exception will be generated if bin would consist of (for instance) 17 bits.

On the other hand, the variable *bitstring* may consist of any number of bits, for instance 0, 1, 8, 11, 17, 42, and so on, because the default unit for bitstrings is 1.

The following example

```
<<(x (:size 1)) (y (:size 6))>>
(binary (x (:size 1)) (y (:size 6)))
```

will successfully construct a *:bitstring* of 7 bits. (Provided that all of *x* and *y* are integers.)

When constructing binaries, *value* and *size* can be any expression.

Including Literal Strings

As syntactic sugar, a literal string may be written instead of an element.

```
<<"hello">>
```

which is syntactic sugar for

```
<<\h \e \l \l \o>>
```

Matching Binaries

This section describes the rules for matching binaries using the bit syntax.

There can be zero or more segments in a binary pattern. A binary pattern can occur in every place patterns are allowed, also inside other patterns. Binary patterns cannot be nested.

The pattern <<>> matches a zero length binary.

Each segment in a binary can consist of zero or more bits.

A segment of type binary must have a size evenly divisible by 8 (or divisible by the unit size, if the unit size has been changed).

A segment of type bitstring has no restrictions on the size.

When matching value *value* must be either a variable or an integer or floating point literal. Expressions are not allowed.

:size must be an integer literal, or a previously bound variable.

Getting the Rest of the Binary or Bitstring

To match out the rest of a binary, specify a binary field without size:

```
(case foo
  (<<(a (:size 8)) (rest :binary)>>
     rest))
```

The size of the tail must be evenly divisible by 8.

To match out the rest of a bitstring, specify a field without size:

```
(case foo
   (<<(a (:size 8)) (rest :bitstring)>>
    rest))
```

There is no restriction on the number of bits in the tail.

Examples

```
<<\a \b \c>>
<<a b (c :size 16)>>

(case <<1 2 3>>
  (<<a b c>>
     {a b c})))

(case <<1 2 3>>
  (<<a b (c :size 16)>>
     {a b c})))

(case <<(1 :size 16) 2 (3 :binary)>>
  (<<(d :size 16) e (f :binary)>>
     {d e f})))

 <<"This is a test">>
(binary "This is a test")

(binary \a \b \c)
(binary a b (c :size 16))

(case (binary 1 2 3)
  ((binary a b c)
     {a b c})))

(case (binary 1 2 3)
  ((binary a b (c :size 16))
     {a b c})))

(case (binary (1 :size 16) 2 (3 :binary))
  ((binary (d :size 16) e (f :binary))
     {d e f})))
```

4.1.7 $filename

```
($filename)
```

4.1.8 $namespace

```
($namespace)
```

4.1.9 $line-number

```
($line-number)
```

4.1.10 $function-name

```
($function-name)
```

4.1.11 apply

```
(apply fun [args ...])
```

4.1.12 quote

```
(quote expr ...)
'expr
:atom
```

4.1.13 quasiquote

```
`expr
```

4.1.14 string

```
(string "values")
```

4.1.15 list

```
(list expr ...)
[expr ...]
```

4.1.16 tuple

```
(tuple expr ...)
{expr ...}
```

4.1.17 macroexpand-1

```
(macroexpand-1 expr ...)
```

4.1.18 fn

```
(fn (arg ...) expr ...)
```

4.2 Namespaces

ns declarations are used to define the namespace in which a set of definitions live. The generally also define the context, that is what other namespaces are available, what functions from other namespaces are imported and what attributes are defined. A basic namespace declaration looks as follows.

```
(ns my-super-module)
```

This defines a namespace the *defn* and *defmacro* definitions that follow are part of that namespace. The namespace must be defined before the functions using that namespace. You may also have as many namespaces as you would like per file, though that is not encouraged.

4.2.1 Namespace Body

The namespace body may consist of any number of *require*, *use* and clauses in any order and in any conversation.

4.2.2 Requiring Namespaces

Other namespaces are *not* available in your namespace until you declare your need in a *require* or *use* clause. For example the following namespace will fail during compile.

```
(ns my-converter)

(defn+ convert-string (str)
    (erlang/binary_to_list str))
```

This would fail during compilation because you have not declared your that you are going to use the erlang namespace. We can fix this by adding a require clause.

```
(ns my-converter
   (require erlang))

(defn+ convert-string (str)
    (erlang/binary_to_list str))
```

Suddenly everything compiles happily.

There are several variations to the require clause that you can use. The variation you use is really up to you. For example to require multiple namespaces you could have them all in the same require clause or each on individual require clauses.

```
(require erlang string test)

(require erlang)
(require string)
(require test)
```

in general it is much more common to include everything in a single require clause.

Aliasing with Require

Sometimes namespaces names are very long and its annoying to use them in the namespace body. To avoid this you can add an *:as* element to the require clause. This allows you to use both the original name and the aliased name in your namespace body. For example, if we use erl_prim_loader we might want to rename it as loader.

```
(ns my-example
   (require (erl_prim_loader :as loader)))

(defn name-example ()
   (erl_prim_loader/get_path))

(defn alias-example ()
   (loader/get_path))
```

Both of these examples are functionally equivalent.

Making Erlang Modules Appear Like Joxa Namespaces (Joxification)

Its much more common in Joxa to use the - in names as opposed to the _ as is common in Erlang. To make thing more comfortable for the namespace definer Joxa offers the *joxify* element for require clauses. the *joxify* element basically aliases defined names from a name containing _ to a name containing -. It also does this for all the functions in the module.

Lets use our *erl_prim_loader* example again.

```
(ns my-example
   (require (erl_prim_loader :joxify)))

(defn name-example ()
   (erl_prim_loader/get_path))

(defn alias-example ()
   (erl-prim-loader/get-path))
```

Again both of these are functionally Equivalent.

4.2.3 Attribute Clauses

Attribute clauses are the simplest of the three clauses There are simply a three element list where the first element is the identifier 'attr', the second element is a Joxa term that provides the key value and the third is a Joxa term that provides the value.

Attributes follow the form:

```
(attr <key> <value>)
```

These allow you to define attributes on the namespace. Some of which are consumable by the compiler, others just informational, all though are consumable via the module_info. You should note that both the key and the value must be literal values, no evaluation occurs there.

4.2.4 Using Namespaces

The use clause is a way of importing functions into the namespace so that you can use them without prepending the namespace. Use clauses are, by far, the most complex of the namespace clauses as they both manipulate and subset the functions being imported while at the same time aliasing the function if desired. As you can see below each clause may consist of a namespace name, or a list that contains a few subclauses. The sub-clause is always headed by a namespace name, followed by an action, followed by the subject of that action. The action/subject may be repeated to further refine and modify the imported values. The sub-clause action/subject may occur in any order. Even though some do not make sense when used together. So, for example you could have the following

```
(use string)

(use (string :only (tokens/2)))

(use (string :exclude (substr/3
                        join/2
                        join/3)))

(use (string :rename ((substr/3 str-substring)
                       (join/2 str-join))))

(use (string :as str
             :only (join/2
                    substr/3)))

(use (string :as str
             :only (tokens/2)))

(use (string :as str
             :exclude (substr/3
                        join/2
                        join/3)))

(use (string :as str
             :joxify
             :rename ((substr/3 str-substring)
                       (join/2 str-join))))
```

You should think about use clauses as a series of actions that occur from left to right. Lets take an example and work through it. The following is a fairly complex example that highlights some things that we might want to do.

```
(use (string :exclude (substr/4 join/2)
             :joxify
             :rename ((sub-word/3 str-subword) (join/2 str-join))))
```

Lets break this down into actions.

1. The namespace declaration. In this case *string*, this goes to the namespace and gets a list of all the functions that that namespace exports. That list of functions is then passed to the next 'operation'.
2. Exclude, this excludes the specified functions from the function list that was imported. Every action/sub-clause pair after this exclude will only operate on the functions that have not been excluded. The opposite of exclude is *only*. Only subsets the list of functions to just those specified in the only clause.
3. Joxify, This does the exact same thing that joxify does in require. However, it does it only on the module name and the functions that we currently have in the list. After this point the functions in the list can only be referred to by the joxified name.
4. Rename. This does what you would think. It renames a function giving it a different name. This does this on the list of functions being passed forward. In this example we are renaming *sub-word/3* to *str-subword*. However if we tried to rename *substr/4* which we excluded it would have no effect since its not in the list of imports being carried forward. *NOTE* note the joxification of *sub-word/3*. Since we specified *joxify* earlier we must must refer to it as *sub-word/3* instead of *sub_word/3*.

4.2.5 Author's Note

When you use *require* vs *use* is entirely up to you. Joxa is a young language and there has not yet been time to hash out what is the best practice here. I have had the good fortune to code in may languages and several of those languages have supported 'import' clause's like use. In the best of those languages the general practice is to use the *use* clause only when you are importing *operators* the require clause for everything else. In the case of Joxa I will define operators as anything thats used in a conditional statement, including guards. The main thing you want to remember is that *use* impairs locality of code just a bit (that is knowing where the code that is being executed is coming from). There are times (like conditionals) when the clarity of the code is improved enough to make that locality hit worth while, but in general thats not true. In the end, just remember that the more transparent code is the easier it is to maintain and extend and choose *use* and *require* with an eye towards transparency.

4.3 Functions

4.3.1 *&rest* Arguments to Functions

Rest arguments in a language like Joxa, where arity is basically part of the namespace, take a bit of thought to get your mind around. Basically, Joxa like Lisp has the ability to group all remaining arguments into a list at the discretion of the function implementer. This changes the way those functions are called and perhaps referred to.

Defined Functions

In module defined functions rest arguments work like you would expect. For example:

```
(defn+ i-am-a-rest-fun (arg1 arg2 &rest arg3)
    {arg1 arg2 arg3})
```

In this case, any time *i-am-a-rest-fun* is called, the arguments are collapsed down for the third argument. This happens for any call that has more then three arguments.

In this case of namespaces *i-am-a-rest-fun/3* can actually be referred to by any arity that is 3 or greater. For example *i-am-a-rest-fun/545* still refers to *i-am-a-rest-fun/3* because those extra arguments are simply collapsed to the three. With that in mind you could define *i-am-a-rest-fun/2* without a problem. However, you could never define *i-am-a-rest-fun/5* because *i-am-a-rest-fun/3* overrides anything with arguments three or greater. to give a concrete example, you could define:

```
(defn+ i-am-a-rest-fun (arg1 arg2)
    {arg1 arg2})
```

and it would be valid and make sense. However, you could not define

```
(defn+ i-am-a-rest-fun (arg1 arg2 arg3 arg4)
    {arg1 arg2 arg3 arg4})
```

Because *i-am-a-rest-fun/3* already fills that namespace completely.

Anonymous Functions

Anonymous functions work exactly like defined functions. I could do

```
(fn (one two &rest three)
   {one two three})
```

I can then assign that to the variable *foo* and call *foo* as:

```
(foo 1 2 3 4 5 6 7 8 9)
```

and it would do the correct thing.

Variables that Refer to Functions

For the most part variables that reference rest functions work exactly like you would expect. However, in the case where the 'restful-ness' of a variable can not be defined at compile time, a function is created that does the resolution at run time. This mostly happens when variables are passed as arguments to functions. At the moment the argument boundary can not be crossed, so when those variables are used as functions, they are wrapped in a function that does the runtime resolution and calls the correct function with the correct args. This may affect performance.

Apply

Apply also works exactly as you would expect. Any resolvable rest call has the arguments handled correctly at compile time. Any un-resolvable rest call has a function created to correctly handle the arguments at runtime.

Importing Rest Functions via Use

The *use* clause in module declarations take a bit of thinking. To refer to a function in a use clause use the actual arity. In our function above you would use *(use :only i-am-a-rest-fun/3)*

4.4 Type Specs

4.4.1 Mutually Recursive Modules

In Joxa code must exist at compile time before it is called. That means that if you are compiling a module and it calls other modules those other modules must exist to be called (at compile time). If they are not it is a build failure. Unfortunately, this makes mutually recursive functions somewhat difficult. In general mutually recursive modules are something to be avoided. However, at times they are needed and there is no way to get around that need. When this occurs Joxa provides a facility to get around it. This is very similar to its forward declarations via defspecs. That way is to define a spec for the remote function. Lets take a look at an example of this

```
(ns Joxa-exp-nmr-ns1)

(defn+ final ()
   :got-it)

;; Forward declaration for ns2
(defspec Joxa-exp-nmr-ns2/recurse-ns1 () (erlang/any))

(defn+ recurse-ns2 ()
  (joxa-test-nmr-ns2/recurse-ns1))

;; ======

(ns joxa-exp-nmr-ns2)

(defspec joxa-exp-nmr-ns1/final () (erlang/any))

(defn+ recurse-ns1 ()
   (joxa-exp-nmr-ns1/final))
```

Notice that *joxa-exp-nmr-ns1* has a dependency on *joxa-exp-nmr-ns2* and vice versa. In normal Joxa code this would not be compilable because the code that is being called must be available before it is called. However, we have gotten around this problem by providing remote defspecs. In *joxa-exp-nmr-ns1* we pre-declare *joxa-exp-nmr-ns2/recurse-ns1* while in *joxa-exp-nmr-ns2* we pre-declare *joxa-exp-nmr-ns1/final*. This allows the Joxa compiler to check the function against the specs instead of the real module. Of course, there is no way for the compiler to know if those functions actually exist, so if you make a mistake you may actually get runtime errors. So be careful.

CHAPTER 5

Standard Library

5.1 Core

5.1.1 *!=*

This is a 'not equal' operator for Joxa. It is basically equivelent to (not (= ...)) or the erlang =:=

Example

```
joxa-is> (joxa-core/!= 1 2)
:true

joxa-is> (joxa-core/!= 1 1)
:false
```

5.1.2 lte

Less then or equal to. Basically equivalent to =<. However this has some issue in Joxa's syntax but *lte* solves these problems.

Example

```
joxa-is> (joxa-core/lte 1 2)
:true

joxa-is> (joxa-core/lte 1 1)
:true

joxa-is> (joxa-core/lte 10 1)
:false
```

5.1.3 gte

Greater then or equal to. Basically equivalent to >=. However this has some issue in Joxa's syntax but *gte* solves these problems.

Example

```
joxa-is> (joxa-core/gte 1 2)
:false

joxa-is> (joxa-core/gte 1 1)
:true

joxa-is> (joxa-core/gte 10 1)
:true
```

5.1.4 and

This is a boolean *and* operation. The main difference between this and *erlang/and/2* is that this allows an unlimited number of arguments, in the tradition of lisp.

Example

```
joxa-is> (joxa-core/and :true :true :false)
:false

joxa-is> (joxa-core/and :true :true :true (joxa-core/!= 1 2))
:true
```

5.1.5 or

This is a boolean *or* operation. The main difference between this and *erlang/or/2* is that this allows an unlimited number of arguments, in the tradition of lisp.

Example

```
joxa-is> (joxa-core/or :true :true :false)
:true

joxa-is> (joxa-core/or :true :true :true (joxa-core/!= 1 2))
:true

joxa-is> (joxa-core/or :false :false :false)
:false
```

5.1.6 *+*

This is the multi-argument version of *erlang/+*. It does a simple arithmetic addition for an unlimited number of arguments.

Example

```
joxa-is> (joxa-core/+ 1 2 3 4 5 6 7)
28
```

5.1.7 -

This is the multi-argument version of *erlang/-*. It does a simple arithmetic subtraction for an unlimited number of arguments.

Example

```
joxa-is> (joxa-core/+ 1 2 3 4 5 6 7)
-26
```

5.1.8 incr

This increments a numeric value (either float or integer)

Example

```
joxa-is> (joxa-core/incr 1)
2

joxa-is> (joxa-core/incr 1.0)
2.0
```

5.1.9 decr

This decrements a numeric value (either float or integer)

Example

```
joxa-is> (joxa-core/decr 1)
0

joxa-is> (joxa-core/incr 1.0)
0.0
```

5.1.10 if

```
if test-form then-form else-form => result*
```

Arguments and Values

test-form a form.

then-form a form.

else-form a form.

results if the test-form yielded true, the values returned by the then-form; otherwise, the values returned by the else-form.

Description

if allows the execution of a form to be dependent on a single test-form.

First test-form is evaluated. If the result is true, then then-form is selected; otherwise else-form is selected. Whichever form is selected is then evaluated.

Examples

```
joxa-is> (joxa-core/if :true 1 2)
1
joxa-is> (joxa-core/if :false 1 2)
2
```

5.1.11 when

```
when test-form form* => result*
```

Arguments and Values

test-form a form.

forms an implicit do.

results the values of the forms in a when form if the test-form yields *:true* or in an unless form if the test-form yields *:false*; otherwise *:ok*.

Description

when allows the execution of forms to be dependent on a single test-form.

In a when form, if the test-form yields true, the forms are evaluated in order from left to right and the values returned by the forms are returned from the when form. Otherwise, if the test-form yields false, the forms are not evaluated, and the when form returns *:ok*.

Examples

```
joxa-is> (joxa-core/when :true :hello)
hello
joxa-is> (joxa-core/when :false :hello)
ok
joxa-is> (joxa-core/when :true (io/format "1") (io/format "2") (io/format "3"))
123
```

5.1.12 unless

```
unless test-form form* => result*
```

Arguments and Values

test-form a form.

forms an implicit do.

results the values of the forms in a when form if the test-form yields *:true* or in an unless form if the test-form yields *:false*; otherwise *:ok*.

Description:

unless allows the execution of forms to be dependent on a single test-form.

In an unless form, if the test-form yields false, the forms are evaluated in order from left to right and the values returned by the forms are returned from the unless form. Otherwise, if the test-form yields *:false*, the forms are not evaluated, and the unless form returns *:ok*.

Examples

```
joxa-is> (joxa-core/unless :true :hello)
ok
joxa-is> (joxa-core/unless :false :hello)
hello
joxa-is> (joxa-core/unless :true (io/format "1") (io/format "2") (io/format "3"))
ok
joxa-is> (joxa-core/unless :false  (io/format "1") (io/format "2") (io/format "3"))
123
```

5.1.13 gensym

```
gensym => new-atom
gensym x => new-atom
```

Arguments and Values

x a string.

new-symbol a fresh, atom.

Description

Creates and returns a fresh, atom.

The name of the new-symbol is the concatenation of a prefix, which defaults to "G", and a suffix, which is a randomly generated number.

If x is supplied, then that string is used as a prefix instead of "G" for this call to gensym only.

Examples

```
joxa-is> (joxa-core/gensym)
'#:GAEECC9'
joxa-is> (joxa-core/gensym "T")
'#:|T66BA871|'
```

5.1.14 try

5.1.15 let-match

5.1.16 define

```
define name value => form
```

Arguments and Values

name an atom that represents the defined name

forms an arbitrary value

form a new defmacro that evaluates to the value

Description:

Defines a new macro that creates a compile time mapping between the name and the value.

Examples

```
joxa-is> (joxa-core/define :true :hello)
ok
joxa-is> (joxa-core/unless :false :hello)
hello
joxa-is> (joxa-core/unless :true (io/format "1") (io/format "2") (io/format "3"))
ok
joxa-is> (joxa-core/unless :false  (io/format "1") (io/format "2") (io/format "3"))
123
```

5.2 Lists

dolist

hd

tl

foldl

map

5.3 Records

Records in Joxa are equivalent and compatible with records in Erlang. However, like many things in Joxa they are used in very different ways.

Before we get started there are a few things to keep in mind. Records are designed to be contained in a single namespace. Defining multiple records in the same namespace will cause the record system to stomp on itself. That is the record macros generate many functions and macros to access various parts of the record. With multiple records in the same namespace those function names that are generated will conflict.

5.3.1 Overview

Having a namespace per record is no big deal sense you can have multiple namespaces per file. So to get started lets look at a trivial, contrived example

```
(ns example-person
       (require erlang lists)
       (use (joxa-records :only (defrecord/2))))

(joxa-records/defrecord+ person name age {sex male}
                         {address "unknown"} city)


(ns example-walker
     (require example-person))

(defn+ create-irish-walker ()
  (example-person/make
                     "Robert"
                     1024
                     :male
                     "Somewhere in Ireland"))

(defn+ is-robert? (person)
  (case person
     ((example-person/t name "Robert")
        :true)
     (_
       :false)))

(defn+ is-robert-male? (person)
  (case person
    ((example-person/t name "Robert"
                            sex male)
       :true)
    (_
      :false)))

(defn+ get-name-age-address (person)
   (example-person/let person
                      (name local-name
                       age local-age
                      address local-address)
     {local-name local-age local-address}))
```

This gives a quick overview of some of the things you can do with records. Now lets jump into some detail.

5.3.2 Definition

joxa-records is the namespace that contains the record system for Joxa. The two functions that you will interact the most are *defrecord* and *defrecord+*. *defrecord* and *defrecord+* both have the exact same api. Just like with *defn* and *defmacro* the + added to defrecord means that the record functions and macros will be exported.

defrecord takes a name for the record followed by a list of field descriptions. In our example we called our record *person*

```
(joxa-records/defrecord+ person name age {sex male}
                         {address "unknown"} city)
```

We then followed it up with the fields *name*, *age*, *sex*, *address* and city. As you can see, the *sex* and *address* fields are a bit different. That is because in these cases we are providing a default value. So in record definitions you can provide just a name, or a name and a default value. Just as a note, the name must be an atom and the value a literal.

This is really all there is to it. The *defrecord* macro generates a bunch of functions and macros used to interact with the record.

5.3.3 Creating Records

Once the record is defined we want to create it in the code that is making use of the record. When a record is defined two functions are defined that are used to create records. The first is the *make* function. The second is the *make-fields* function.

Lets start by looking at the make function

```
(example-person/make
                 "Robert"
                 1024
                 :male
                 "Somewhere in Ireland"),
```

The make function is fairly strait forward, simple pass values for the record in the order in which those fields where defined in the *defrecord* definition. In this case you must pass a value for every field in the record.

The make fields function is a bit more flexible. In this case you call *make-fields* with a property list that provides a value for each field that you are interested in defining a value for. Undefined fields will simple get the value *undefined* or the default value specified on record definition. As we can see in the following example

```
(example-person/make-fields
                [{:name "Robert"}
                 {:age 1024}
                 {:address "Somewhere in Ireland"}])
```

We do not provide a value for the *sex* field or the *city* field. In the case of *sex* the value will default to *mail* while in the case of *city* it will default to *undefined*.

5.3.4 Getters and Setters

defrecord generates several different ways of getting and setting values from a function. The most strait forward of these is the field name accessors. For each field defined Joxa generates a function to get and set the value. The getter is a function with the name of the field that takes a single value (the record). The setter is the name of the field post-fixed by a *!* that takes the record as the first argument as the new value as the second argument. So for example if we wanted to get and set the *age* field of the person record we could do the following

```
(let (age (example-person/age foo-record))
    (example-person/age! foo-record (+ age 1)))
```

defrecord also creates a set of anonymous getters and setters that take the name of the field as an atom. These are the *element* and *element!* functions. To accomplish the same thing we did above, but with these anonymous functions we could do the following

```
(let (age (example-person/element :age foo-record))
    (example-person/element! foo-record :age (+ age 1)))
```

This makes it quite a bit easier to pragmatically manipulate a record.

Finally, the record system provides a way for the use to get access to several fields at the same time. This is accomplished through a specialized let function. So lets say we wanted to get the *name*, *age* and *address* fields from the record all at once. We could use the generated let as follows

```
(example-person/let person-record
                   (name local-name
                    age local-age
                    address local-address)
  {local-name local-age local-address})
```

The first argument is the record that will have fields extracted. The second argument is a list of field name, reference name pairs while the rest is the body of the let. So in this case the value of the *name* field in the *person-record* will be bound to the reference *local-name* and be made available in the body of the let. The same is true for *age* and *address*.

5.3.5 Pattern Matching

Joxa has pattern matching and, of course, you want to be able to trivially match on records. To that end the Joxa record system provides a macro that generated a matchable thing. That macro is the *t* macro. The *t* macro takes a list of field name, data pairs that are used to construct a pattern for that record. Lets look at some examples. In the first example we want to create something that will match on a record with the *name* "Robert" and nothing else

```
(case person-rec
 ((example-person/t name "Robert")
     :matched)
 (_
     :did-not-match))
```

If we want to match on more fields we can simple add more to the field/value list

```
(case person
 ((example-person/t name "Robert"
                         sex male)
     :matched)
 (_
     :did-not-match))
```

or even

```
(case person
  ((example-person/t name "Robert"
                          sex :male
                          city :chicago)
      :matched)
  (_
      :did-not-match)))
```

5.3.6 Meta Data

Finally the record system wants to give you the ability to do unanticipated things when the need arises. So two functions are defined to give you metadata data about the record. These functions are *field-info/0* and *field-info/1*. Field info is a tuple of three values that gives you the name of the field, the position of the field in the tuple and its default value. In our example-person record the result of *field-info/0* is

```
[{name,2,undefined},
 {age,3,undefined},
 {sex,4,male},
 {address,5,"Somewhere in Ireland"},
 {city,6,undefined}]
```

As you can see it gives you metadata for all the fields. *field-info/1* returns the same metadata but only for a single field. So if we called *field-info* with *name* we would get

```
{name,2,undefined}
```

5.3.7 Future Directions

There is still a lot that can be added to records. Things like

- Pre and Post hook functions
- Types and automatic type validators

and more. However, the core defined here shouldn't change significantly.

CHAPTER 6

Joxa Style Guide

This work is derived from Riastradh's Lisp Style Rules by Talor R. Cambell

This is document describes a recommended style for Joxa. Its an distilled from the best practices of the existing Lisp world and the lessons learned in Joxa itself. Its not meant to be a rigid set of rules for the style extremists. It is meant to help you get the most out of Joxa.

This guide is written primarily as a collection of guidelines, with rationale for each rule (If a guideline is missing rationale, please inform the author!). Although a casual reader might go through and read the guidelines without the rationale, such a reader would derive little value from this guide. In order to apply the guidelines meaningfully, their spirit must be understood; the letter of the guidelines serves only to hint at the spirit. The rationale is just as important as the guideline.

6.1 Standard Rules

These are the standard rules for formatting Lisp code; they are repeated here for completeness, although they are surely described elsewhere. These are the rules implemented in Emacs Lisp modes, and utilities such as Paredit.

6.1.1 Parentheses

Terminology

This guide avoids the term *parenthesis* except in the general use of *parentheses* or *parenthesized*, because the word's generally accepted definition, outside of the programming language, is a statement whose meaning is peripheral to the sentence in which it occurs, and *not* the typographical symbols used to delimit such statements.

The balanced pair of typographical symbols that mark parentheses in English text are *round brackets*, i.e. *(* and *)*. There are several other balanced pairs of typographical symbols, such as *square brackets* (commonly called simply *brackets* in programming circles), i.e. *[* and *]*; *curly braces* (sometimes called simply *braces*), i.e. *{* and *}*; *angle brackets* (sometimes *brokets* (for *broken brackets*)), i.e. < and >.

In any balanced pair of typographical symbols, the symbol that begins the region delimited by the symbols is called the *opening bracket* or the *left bracket*, such as *(* or'[' or *{* or <. The symbol that ends that region is called the *right bracket* or the *closing bracket*, such as > or *}* or *]* or *)*.

6.1.2 Spacing

If any text precedes an opening bracket or follows a closing bracket, separate that text from that bracket with a space. Conversely, leave no space after an opening bracket and before following text, or after preceding text and before a closing bracket.

Unacceptable

```
(foo(bar baz)quux)
(foo ( bar baz ) quux)
```

Acceptable:

```
(foo (bar baz) quux)
```

Rationale

This is the same spacing found in standard typography of western text. It is more aesthetically pleasing.

6.1.3 Line Separation

Absolutely do *not* place closing brackets on their own lines.

Unacceptable

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1

                      )
           )
      )
  )
 )
```

Acceptable

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1)))))
```

Rationale

The parentheses grow lonely if their closing brackets are all kept separated and segregated.

Exceptions to the Above Rule Concerning Line Separation

Do not heed this section unless you know what you are doing. Its title does *not* make the unacceptable example above acceptable.

When commenting out fragments of expressions with line comments, it may be necessary to break a line before a sequence of closing brackets

```
(define (foo bar)
  (list (frob bar)
        (zork bar)
        ;; (zap bar)
        ))
```

Finally, it is acceptable to break a line immediately after an opening bracket and immediately before a closing bracket for very long lists, especially in files under version control. This eases the maintenance of the lists and clarifies version diffs. Example

```
(define colour-names         ;Add more colour names to this list!
  '(
    blue
    cerulean
    green
    magenta
    purple
    red
    scarlet
    turquoise
    ))
```

6.1.4 Parenthetical Philosophy

The actual bracket characters are simply lexical tokens to which little significance should be assigned. Lisp programmers do not examine the brackets individually, or, Azathoth forbid, count brackets; instead they view the higher-level structures expressed in the program, especially as presented by the indentation. Lisp is not about writing a sequence of serial instructions; it is about building complex structures by summing parts. The composition of complex structures from parts is the focus of Lisp programs, and it should be readily apparent from the Lisp code. Placing brackets haphazardly about the presentation is jarring to a Lisp programmer, who otherwise would not even have seen them for the most part.

Indentation and Alignment

The operator of any form, i.e. the first subform following the opening round bracket, determines the rules for indenting or aligning the remaining forms. Many names in this position indicate special alignment or indentation rules; these are special operators, macros, or procedures that have certain parameter structures.

If the first subform is a non-special name, however, then if the second subform is on the same line, align the starting column of all following subforms with that of the second subform. If the second subform is on the following line, align its starting column with that of the first subform, and do the same for all remaining subforms.

In general, Emacs will indent Lisp code correctly. Run *C-M-q* (indent-sexp) on any code to ensure that it is indented correctly, and configure Emacs so that any non-standard forms are indented appropriately.

Unacceptable

```
(+ (sqrt -1)
  (* x y)
  (+ p q))

(+
   (sqrt -1)
   (* x y)
   (+ p q))
```

Acceptable

```
(+ (sqrt -1)
   (* x y)
   (+ p q))

(+
 (sqrt -1)
 (* x y)
 (+ p q))
```

Rationale

The columnar alignment allows the reader to follow the operands of any operation straightforwardly, simply by scanning downward or upward to match a common column. Indentation dictates structure; confusing indentation is a burden on the reader who wishes to derive structure without matching parentheses manually.

Non-Symbol Indentation and Alignment

The above rules are not exhaustive; some cases may arise with strange data in operator positions.

6.1.5 Lists

Unfortunately, style varies here from person to person and from editor to editor. Here are some examples of possible ways to indent lists whose operators are lists:

Questionable

```
((car x)                            ;Requires hand indentation.
   (cdr x)
   foo)

((car x) (cdr x)                    ;GNU Emacs
 foo)
```

Preferable

```
((car x)                            ;Any Emacs
 (cdr x)
 foo)
```

Rationale

The operands should be aligned, as if it were any other procedure call with a name in the operator position; anything other than this is confusing because it gives some operands greater visual distinction, allowing others to hide from the viewer's sight. For example, the questionable indentation

```
((car x) (cdr x)
 foo)
```

can make it hard to see that *foo* and *(cdr x)* are both operands here at the same level. However, GNU Emacs will generate that indentation by default.

6.1.6 Strings

If the form in question is meant to be simply a list of literal data, all of the subforms should be aligned to the same column, irrespective of the first subform.

Unacceptable

```
("foo" "bar" "baz" "quux" "zot"
       "mumble" "frotz" "gargle" "mumph")
```

Questionable, but acceptable

```
(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
   3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1 6 9 3 9 9 3)
```

Acceptable

```
("foo" "bar" "baz" "quux" "zot"
 "mumble" "frotz" "gargle" "mumph")

("foo"
  "bar" "baz" "quux" "zot"
  "mumble" "frotz" "gargle" "mumph")
```

Rationale

Seldom is the first subform distinguished for any reason, if it is a literal; usually in this case it indicates pure data, not code. Some editors and pretty-printers, however, will indent unacceptably in the example given unless the second subform is on the next line anyway, which is why the last way to write the fragment is usually best.

6.1.7 Names

Naming is subtle and elusive. Bizarrely, it is simultaneously insignificant, because an object is independent of and unaffected by the many names by which we refer to it, and also of supreme importance, because it is what programming – and, indeed, almost everything that we humans deal with – is all about. A full discussion of the concept of name lies far outside the scope of this document, and could surely fill not even a book but a library.

Symbolic names are written with English words separated by hyphens. Scheme and Common Lisp both fold the case of names in programs; consequently, camel case is frowned upon, and not merely because it is ugly. Underscores are unacceptable separators except for names that were derived directly from a foreign language without translation.

Unacceptable

```
XMLHttpRequest
foreach
append_map
```

Acceptable

```
xml-http-request
for-each
append-map
```

6.1.8 Funny Characters

Question Marks: Predicates

Affix a question mark to the end of a name for a procedure whose purpose is to ask a question of an object and to yield a boolean answer. Such procedures are called *predicates*. Do not use a question mark if the procedure may return any object other than a boolean.

Examples .. code-block:: clojure

pair? procedure? proper-list?

Pronounce the question mark as if it were the isolated letter *p*. For example, to read the fragment *(pair? object)* aloud, say: *pair-pee object.*

Exclamation Marks: Destructive Operations

Affix an exclamation mark to the end of a name for a procedure (or macro) whose primary purpose is to modify an object. This is common in lisps that support destructive operations. Joxa, of course, does not. However, this syntax is useful in situations where the intent is to modify an object.

Examples

```
set-car! append!
```

Pronounce the exclamation mark as *bang*. For example, to read the fragment (append! list tail) aloud, say: *append-bang list tail*.

Asterisks: Variants, Internal Routines

Affix an asterisk to the end of a name to make a variation on a theme of the original name.

Example

```
let -> let*
```

Prefer a meaningful name over an asterisk; the asterisk does not explain what variation on the theme the name means.

with- and *call-with-*: Dynamic State and Cleanup

Prefix *WITH-* to any procedure that establishes dynamic state and calls a nullary procedure, which should be the last (required) argument. The dynamic state should be established for the extent of the nullary procedure, and should be returned to its original state after that procedure returns.

Examples

```
with-input-from-file
with-output-to-file
```

Prefix *call-with-* to any procedure that calls a procedure, which should be its last argument, with some arguments, and is either somehow dependent upon the dynamic state or continuation of the program, or will perform some action to clean up data after the procedure argument returns. Generally, *CALL-WITH-* procedures should return the values that the procedure argument returns, after performing the cleaning action.

call-with-input-file and *call-with-output-file* both accept a pathname and a procedure as an argument, open that pathname (for input or output, respectively), and call the procedure with one argument, a port corresponding with the file named by the given pathname. After the procedure returns, call-with-input-file and call-with-output-file close the file that they opened, and return whatever the procedure returned.

Generally, the distinction between these two classes of procedures is that *call-with-...* procedures should not establish fresh dynamic state and instead pass explicit arguments to their procedure arguments, whereas *with-...* should do the opposite and establish dynamic state while passing zero arguments to their procedure arguments.

6.1.9 Comments

Write heading comments with at least four semicolons; see also the section below titled ‘Outline Headings’.

Write top-level comments with three semicolons.

Write comments on a particular fragment of code before that fragment and aligned with it, using two semicolons.

Write margin comments with one semicolon.

The only comments in which omission of a space between the semicolon and the text is acceptable are margin comments.

Examples

```
;;;; Frob Grovel

;;; This section of code has some important implications:
;;;   1. Foo.
;;;   2. Bar.
;;;   3. Baz.

(defn (fnord zarquon)
  ;; If zob, then veeblefitz.
  (quux zot
        mumble             ;Zibblefrotz.
        frotz))
```

6.2 General Layout

Contained in the rationale for some of the following rules are references to historical limitations of terminals and printers, which are now considered aging cruft of no further relevance to today's computers. Such references are made only to explain specific measures chosen for some of the rules, such as a limit of eighty columns per line, or sixty-six lines per page. There is a real reason for each of the rules, and this real reason is not intrinsically related to the historical measures, which are mentioned only for the sake of providing some arbitrary measure for the limit.

6.2.1 File Length

If a file exceeds five hundred twelve lines, begin to consider splitting it into multiple files. Do not write program files that exceed one thousand twenty-four lines. Write a concise but descriptive title at the top of each file, and include no content in the file that is unrelated to its title.

Rationale

Files that are any larger should generally be factored into smaller parts. (One thousand twenty-four is a nicer number than one thousand.) Identifying the purpose of the file helps to break it into parts if necessary and to ensure that nothing unrelated is included accidentally.

6.2.2 Top-Level Form Length

Do not write top-level forms that exceed twenty-one lines, except for top-level forms that serve only the purpose of listing large sets of data. If a procedure exceeds this length, split it apart and give names to its parts. Avoid names formed simply by appending a number to the original procedure's name; give meaningful names to the parts.

Rationale

Top-level forms, especially procedure definitions, that exceed this length usually combine too many concepts under one name. Readers of the code are likely to more easily understand the code if it is composed of separately named parts. Simply appending a number to the original procedure's name can help only the letter of the rule, not the spirit, however, even if the procedure was taken from a standard algorithm description. Using comments to mark the code with its corresponding place in the algorithm's description is acceptable, but the algorithm should be split up in meaningful fragments anyway.

Rationale for the number twenty-one: Twenty-one lines, at a maximum of eighty columns per line, fits in a GNU Emacs instance running in a 24x80 terminal. Although the terminal may have twenty-four lines, three of the lines are occupied by GNU Emacs: one for the menu bar (which the author of this guide never uses, but which occupies a line nevertheless in a vanilla GNU Emacs installation), one for the mode line, and one for the minibuffer's window. The writer of some code may not be limited to such a terminal, but the author of this style guide often finds it helpful to have at least four such terminals or Emacs windows open simultaneously, spread across a twelve-inch laptop screen, to view multiple code fragments.

6.2.3 Line Length

Do not write lines that exceed eighty columns, or if possible seventy-two.

Rationale

Following multiple lines that span more columns is difficult for humans, who must remember the line of focus and scan right to left from the end of the previous line to the beginning of the next line; the more columns there are, the harder this is to do. Sticking to a fixed limit helps to improve readability.

Rationale for the numbers eighty and seventy-two: It is true that we have very wide screens these days, and we are no longer limited to eighty-column terminals; however, we ought to exploit our wide screens not by writing long lines, but by viewing multiple fragments of code in parallel, something that the author of this guide does very often. Seventy-two columns leave room for several nested layers of quotation in email messages before the code reaches eighty columns. Also, a fixed column limit yields nicer printed output, especially in conjunction with pagination; see the section 'Pagination' below.

6.2.4 Blank Lines

Separate each adjacent top-level form with a single blank line (i.e. two line breaks). Do not place blank lines in the middle of a procedure body, except to separate internal definitions; if there is a blank line for any other reason, split the top-level form up into multiple ones.

Rationale

More than one blank line is distracting and sloppy. If the two concepts that are separated by multiple blank lines are really so distinct that such a wide separator is warranted, then they are probably better placed on separate pages anyway; see the next section, *Pagination*.

6.2.5 Dependencies

When writing a file or module, minimize its dependencies. If there are too many dependencies, consider breaking the module up into several parts, and writing another module that is the sum of the parts and that depends only on the parts, not their dependencies.

Rationale

A fragment of a program with fewer dependencies is less of a burden on the reader's cognition. The reader can more easily understand the fragment in isolation; humans are very good at local analyses, and terrible at global ones.

6.2.6 Naming

This section requires an elaborate philosophical discussion which the author is too ill to have the energy to write at this moment.

Compose concise but meaningful names. Do not cheat by abbreviating words or using contractions.

Rationale

Abbreviating words in names does not make them shorter; it only makes them occupy less screen space. The reader still must understand the whole long name. This does not mean, however, that names should necessarily be long; they should be descriptive. Some long names are more descriptive than some short names, but there are also descriptive names that are not long and long names that are not descriptive. Here is an example of a long name that is not descriptive, from SchMUSE, a multi-user simulation environment written in MIT Scheme:

```
frisk-descriptor-recursive-subexpr-descender-for-frisk-descr-env
```

Not only is it long (sixty-four characters) and completely impenetrable, but halfway through its author decided to abbreviate some words as well!

Do not write single-letter variable names. Give local variables meaningful names composed from complete English words.

Rationale

It is tempting to reason that local variables are invisible to other code, so it is OK to be messy with their names. This is faulty reasoning: although the next person to come along and use a library may not care about anything but the top-level definitions that it exports, this is not the only audience of the code. Someone will also want to read the code later on, and if it is full of impenetrably terse variable names without meaning, that someone will have a hard time reading the code.

Give names to intermediate values where their expressions do not adequately describe them.

Rationale

An *expression* is a term that expresses some value. Although a machine needs no higher meaning for this value, and although it should be written to be sufficiently clear for a human to understand what it means, the expression might mean something more than just what it says where it is used. Consequently, it is helpful for humans to see names given to expressions.

Example

A hash table maps foos to bars; *(dict/get dict foo :false)* expresses the datum that dict maps foo to, but that expression gives the reader no hint of any information concerning that datum. *(let ((bar (dict/get dict foo :false))) ...)* gives a helpful name for the reader to understand the code without having to find the definition of HASH-TABLE.

Index variables such as i and j, or variables such as A and D naming the car and cdr of a pair, are acceptable only if they are completely unambiguous in the scope.

Avoid functional combinators, or, worse, the point-free (or *point-less*) style of code that is popular in the Haskell world. At most, use function composition only where the composition of functions is the crux of the idea being expressed, rather than simply a procedure that happens to be a composition of two others.

Rationale

Tempting as it may be to recognize patterns that can be structured as combinations of functional combinators – say, 'compose this procedure with the projection of the second argument of that other one', or *(compose foo (project 2 bar))* –, the reader of the code must subsequently examine the elaborate structure that has been built up to obscure the underlying purpose. The previous fragment could have been written *(fn (a b) (foo (bar b)))*, which is in fact shorter, and which tells the reader directly what argument is being passed on to what, and what argument is being ignored, without forcing the reader to search for the definitions of foo and bar or the call site of the final composition. The explicit fragment contains substantially more information when intermediate values are named, which is very helpful for understanding it and especially for modifying it later on.

The screen space that can be potentially saved by using functional combinators is made up for by the cognitive effort on the part of the reader. The reader should not be asked to search globally for usage sites in order to understand a local fragment. Only if the structure of the composition really is central to the point of the narrative should it be written as such. For example, in a symbolic integrator or differentiator, composition is an important concept, but in most code the structure of the composition is completely irrelevant to the real point of the code.

If a parameter is ignored, give it a meaningful name nevertheless and say that it is ignored; do not simply call it 'ignored'.

When naming top-level bindings, assume namespace partitions unless in a context where they are certain to be absent. Do not write explicit namespace prefixes, such as foo/bar for an operation BAR in a module foo, unless the names will be used in a context known not to have any kind of namespace partitions.

Rationale

Explicit namespace prefixes are ugly, and lengthen names without adding much semantic content. Joxa has its package system to separate the namespaces of names. It is better to write clear names which can be disambiguated if necessary, rather than to write names that assume some kind of disambiguation to be necessary to begin with. Furthermore, explicit namespace prefixes are inadequate to cover name clashes anyway: someone else might choose the same namespace prefix. Relegating this issue to a module system removes it from the content of the program, where it is uninteresting.

6.2.7 Comments

Write comments only where the code is incapable of explaining itself. Prefer self-explanatory code over explanatory comments. Avoid 'literate programming' like the plague.

Rationale

If the code is often incapable of explaining itself, then perhaps it should be written in a more expressive language. This may mean using a different programming language altogether, or, since we are talking about Lisp, it may mean simply building a combinator language or a macro language for the purpose.

6.3 Attribution

This guide was derived from

Riastradh's Lisp Style Rules by Taylor R. Campbell

licensed under:

CHAPTER 7

Contributing

How to contribute to your project. This section should include (in detail):

- How to check out your project's source code.
- Which branch to use for development.
- What style rules to follow when adding code.
- How to run all of the project's unit tests, integration tests, etc.
- An example workflow.

CHAPTER 8

Getting Help

Joxa is in an early stage. So, for now, the Joxa community is the Erlware community. We use the erlware-questions and erlware-dev mailing lists (see below). We also make heavy use of the github issues and wiki. Make use of all these resources for your information needs.

8.1 Resources

1. [Erlware Questions](http://groups.google.com/group/erlware-questions) Is a general list for questions and discussion around Erlware projects, including Joxa. It should be your first stop if you have questions.
2. [Erlware Dev](http://groups.google.com/group/erlware-dev) If you are interested in developing and contributing to an Erlware project, including Joxa, this is the place you should go.
3. [Joxa Issues](https://github.com/erlware/joxa/issues)
4. [Joxa Wiki](https://github.com/erlware/joxa/wiki)

CHAPTER 9

Frequently Asked Questions

Joxa is a very small functional language. Its actually designed less to be a language as a tool set in which to build domain specific languages through the use of macros and libraries.

Though it is based on the Erlang VM it is not, and has no intention of being, Erlang.

9.1 What is the difference between Joxa and LFE (both Lisps for the Erlang VM)

This is best explained in the following post: http://blog.ericbmerritt.com/2012/02/21/differences-between-joxa-and-lfe.html

9.2 How Do You Create Mutually Recursive Functions

All functions in Joxa have to be declared before they can be used. For recursive functions this works fine, however, for two functions that recurse on each other there doesn't seem to be much you can do.

9.3 Type Specs are your answer

Do a *defspec* of the function before using it. Specs, aside from providing type information to the compiler, also serve as a pre-declaration. For example, lets say you had this function:

```
(defn even? (number)
  (case number
      (0
          :true)
      (_
          (odd? (- (erlang/abs number) 1)))))

(defn odd? (number)
   (case number
       (0
           :false)
       (_
           (even? (- (erlang/abs number) 1)))))
```

This obviously wont work because *odd?* will not be declared when *even?* is defined. You can get around this problem by declaring a *defspec* for *odd?*.

```
(defspec odd? ((erlang/integer)) (erlang/boolean))

(defn even? (number)
  (case number
      (0
          :true)
      (_
          (odd? (- (erlang/abs number) 1)))))

(defn odd? (number)
   (case number
       (0
           :false)
       (_
           (even? (- (erlang/abs number) 1)))))
```

With this it works because you have declared your intent to implement *odd?* on which *even?* depends.

9.4 Will compiler.jxa ever be able to use macros?

Probably not, its a problem in the erts code loading scheme. Macros take iterative compilation that is, each form needs to be available at compile time so you have to compile each form and load it individually. When you load the compiler, it overrides the joxa.compiler module currently loaded and since the new thing is incomplete it breaks.

I think there might be some possibility using of the new/old positions in the code loader but that is a long shot. So for the compiler, and the compiler only, macros are not usable. Thats why the bootstrap flag is there it aborts iterative compilation and just does it all in one fell swoop.

CHAPTER 10

Indices and tables

- genindex
- search