Programming in Octaspire Dern

www.octaspire.com

Table of Contents

About

Octaspire Dern is a programming language. It is a dialect of Lisp, having taken influences from languages scheme, emacs lisp and C. The name of the language means an **obscure language**. [1: https://en.wiktionary.org/wiki/dern]

Dern should support imperative and functional programming. It has atomic types **integer**, **real**, (utf-8) **string**, (utf-8) **character**, (utf-8) **symbol**, **boolean**, **nil**, (input/output) **port**, **hash-map**, **environment**, first class **function**, **special** and **builtin**. As non-atomic type it has **vector**. So, where in scheme `'(1 2 3)` is a list, in Dern it is a vector.

Every variable and function definition in Dern must be documented by a **documentation string**. Dern also makes sure that every formal function parameter is documented in the documentation of function definition. The documentation can be accessed with **doc**-function.

Dern can also be extended with functions written in C; C-functions can be registered as **builtins** or **specials**. Arguments to builtins are evaluated normally, but to specials are not, so specials can be used to implement special forms like **if**, etc.

Dern has also library system, that allows one to load libraries using builtin `require`. On Unix-like systems Dern also supports loading of binary libraries (.so files). For the end user, it doesn't matter whether the library she loaded was binary or written in Dern; it can be used exactly the same way.

Dern is dynamically typed language and has **mark and sweep** garbage collector. Functions in dern are first class values. Dern is written in standard **C99** and depends only on C99 compiler, standard library and **octaspire-core** (container, utf-8, and utility library). Dern should compile cleanly without any warnings using `-Wall -Wextra` on any compiler supporting C99. Currently it is tested with **gcc**, **clang**, **Tiny C Compiler (tcc)** and **Portable C compiler (pcc)**.

By using the amalgamated version of Dern, you need only one file. This same single source file can be compiled into (1) stand-alone unit test runner, (2) interactive Dern-REPL and (3) used as single header file library in programs that want to embed the Dern-language. The amalgamated source file **octaspire_dern_amalgamated.c** can be found from the **etc**-directory of the version controlled source distribution. The whole amalgamated release can be found from the **release**-directory of the version controlled source distribution and from octaspire.com/dern/release.tar.bz2.

Dern is portable and is tested and known to run in Linux, FreeBSD, OpenBSD, NetBSD, OpenIndiana, DragonFly BSD, MidnightBSD, MINIX 3, Haiku, and Windows. The `how-to-build` -directory of the amalgamated source release contains build script for all tested platforms.

NOTE	This is the first language I have ever designed or implemented. There are also a lot of features not yet implemented and probably bugs not yet fixed. The interpreter is currently also just a tree walker, and not a faster bytecode vm. Also, English in not my native language, so please bear with me, especially when this first piece/draft of documentation is written in a hurry.

Dern uses Semantic Versioning 2.0.0 version numbering scheme. As long as the MAJOR version number is zero anything can change at any time, even in backwards incompatible manner.

See Dern in action

You can also see Dern in "real" use at: Maze and Lightboard.

Octaspire Maze and Lightboard are games (work in progress) that are being written using Dern and C99.

Building the amalgamated source release

The amalgamated source release is the recommended way of using Dern, if you don't need to modify Dern itself. To use the amalgamated release, you will need only a C compiler and C standard library supporting C99.

Linux, FreeBSD, OpenBSD, NetBSD, OpenIndiana, DragonFly BSD, MidnightBSD, MINIX 3, Haiku

```
curl -O octaspire.com/dern/release.tar.bz2
tar jxf release.tar.bz2
cd release/*
curl -O https://octaspire.github.io/dern/checksums
sha512sum -c checksums
sh how-to-build/YOUR_PLATFORM_NAME_HERE.XX
```

replace **YOUR_PLATFORM_NAME_HERE.XX** with **FreeBSD.sh**, **NetBSD.sh**, **OpenBSD.sh**, **OpenIndiana.sh**, **DragonFlyBSD.sh**, **MidnightBSD**, **linux.sh**, **minix3.sh** or **haiku.sh**. More scripts for different platforms will be added later.

Plan9

```
hget -o release.tar.bz2 http://octaspire.com/dern/release.tar.bz2
bunzip2 release.tar.bz2
tar xf release.tar
cd release/*
rc how-to-build/Plan9.sh
```

Please note, that Dern in Plan9 is currently EXPERIMENTAL, can crash and should be used only for testing and development/fixing purposes.

Windows using MinGW

1. Download and install **MinGW** from www.mingw.org into directory `C:\MinGW`. Please note, that you might need to add `C:\MinGW` and `C:\MinGW\bin` into the `PATH`. If you cannot install into `C:\MinGW`, you can install MinGW to some other place. Remember the installation path, because later you can write it into the `WindowsMinGW.bat` file, so that the script finds MinGW.
2. Download and install **7-Zip** from www.7-zip.org.
3. Download www.octaspire.com/dern/release.tar.bz2 and extract it with 7-Zip. You might need to extract it twice; first into file `release.tar` and then again to get the directory.
4. Start **shell** and change directory to the extracted release directory and then into directory `version-x.y.z`, where x, y and z are some version numbers.
5. When you are in the directory `version-x.y.z` run command `how-to-build\WindowsMinGW.bat`

More scripts for different tools might be added later.

Hello world

Here we have a version of the classic *Hello World*-program in Octaspire Dern. Instead of just printing *Hello, World!*, it is a bit more complex to give you some feeling for the language. If you are in Unix-like system and have **octaspire-dern-repl** in somewhere on your PATH, you can make the script executable using the shebang. You can also run the file by `octaspire-dern-repl hello-world.dern` or by writing it or parts of it directly to the interactive REPL.

hello-world.dern

```
#!/usr/bin/env octaspire-dern-repl
This is a multiline comment.    !#

; 1. Print once 'Hello, World!' and newline
(println [Hello, World!])
(println)

; 2. Print 11 times 'Hello x World!' where x goes from 0 to 10
(for i from 0 to 10 (println [Hello {} World!] i))
(println)

; 3. Print greetings to everybody on the vector
(define names [Christmas card list] '(John Alice Mark))
(for i in names (println [Happy holidays, {}!] i))
(println)

; 4. Add new name, 'Lola', to the names to be greeted
(+= names 'Lola)
(for i in names (println [Happy holidays, {}!] i))
(println)

; 5. Remove one name 'Mark', from the names to be greeted
```

```
(-= names 'Mark)
(for i in names (println [Happy holidays, {}!] i))
(println)

; 6. Define new function to greet people and use it
(define greeter [My greeter function] '(greeting [the greeting] name [who to greet])
(fn (greeting name)
    (println [{}, {}!] greeting name)))

(greeter 'Hi 'Alice)

; 7. Redefine greeter-function with early exit using 'return'
(define grumpy [is our hero grumpy, or not] true)

(define greeter [My greeter function] '(greeting [the greeting] name [who to greet])
(fn (greeting name)
    (if grumpy (return [I am grumpy and will not greet anyone. Hmpfh!]))
    (println [{}, {}!] greeting name)
    (string-format [I greeted "{}", as requested] name)))

(println (greeter 'Hi 'Alice))
(= grumpy false)
(println (greeter 'Hi 'Alice))
(println)

; 8. Add names and custom greetings into a hash map and use it to greet people
(define names [My custom greetings] (hash-map 'John 'Hi
                                              'Lola 'Hello
                                              'Mike 'Bonjour))

(for i in names (greeter (nth 1 i) (nth 0 i)))
```

Values

```
 128              ; These are integers
-100
 3.14             ; These are real
-1.12
[Hello]           ; These are strings (utf-8)
[Hello|newline|]
|a|               ; These are characters (utf-8)
|newline|
|tab|
|bar|
true              ; These are booleans
false
nil               ; Nil
'(1 2 |a| [cat])                              ; These are vectors
'()
(hash-map 'John [likes cats]                  ; This is hash map
          'Lisa [likes dogs]
          'Mike '([likes numbers] 1 2 3 4)
           1    |a|
           [Hi] 2)
```

The text after character `;` is a **single line comment**. Single line comments run until the end of the line. Dern has also **multiline comments** that are written between `\#!` and `!#`. Note that string delimiters in Dern are `[` and `]` and not "; this way dern code can be written inside C-programs without escaping.

Single and multiline comments, making script files executable

Below are examples of single and multiline comments:

```
; This is single line comment.

#! This is multiline comment.
   It can contain multiple lines...
   ... !#
```

Multiline comments can be used to make script files executable in UNIX-like systems:

```
#!/usr/bin/env octaspire-dern-repl
!#

(println [Hello World])
```

Binding names to values with define

```
(define pi [value for pi] 3.14)
(define names [names list] '(John Lisa Mark))
(define double [doubles numbers] '(x [this is doubled]) (fn (x) (* 2 x)))
```

Here we bind three values to a name: one real, one vector and one function taking one argument. Here is an example of using those names:

```
pi
names
(double 1)
```

And to see the documentation for these values:

```
(doc pi)
(doc names)
(doc double)
```

The documentation of the function contains also documentation for the parameters.

Function `doc` can also be used with builtins and specials defined by the standard library or user in C.

NOTE	Please note that at the time of writing most of the functions in Dern's standard library are not yet documented properly. This is a work in progress.

Binding in other environments than the current one

By using an explicit environment argument as the first argument to `define`, we can bind names to values in other environments than the current one. Example:

```
(define myEnv [my own environment] (env-new))
(define myEnv pi [value for pi] 3.14)

pi                  ; <error>: Unbound symbol 'pi'
(eval pi myEnv)     ; 3.14
```

In the example above, `pi` is undefined in the current (global) environment, but it is defined in the `myEnv`-environment. We use special `eval` to evaluate `pi` in the `myEnv`-environment.

Iteration

Dern has two looping constructs: `while` and `for`. For can be used numerically, with a container (vector, string, hash-map, etc.) and with (input) **ports**. Below is couple of examples:

```
(define i [my counter] 0)
(while (<= i 10) (println [Counting at {}...] i) (++ i))
```

Numerical for:

```
(for i from 0 to 10 (println [Hello {} World!] i))
```

Container for:

```
(define names [names list] '(John Mark Lisa))
(for i in names (println [Hello {} World!] i))
```

Both the **numerical for** and **container for** support the use of optional `step` to change the way the iterator is incremented:

```
(for i from 0 to 10 step 3 (println [Hello {} World!] i))

(define names [names list] '(John Mark Lisa))
(for i in names step 2 (println [Hello {} World!] i))
```

Comparing and changing values, predicates

Here are few examples:

```
(<  1 2)   ; true
(<  2 2)   ; false
(>  2 1)   ; true
(<= 1 1)   ; true
(>= 1 1)   ; true
(== 3 3)   ; true
(== 3 1)   ; false
(!= 3 1)   ; true
(+ 1)      ;  1
(+ 1 1)    ;  2
(- 1)      ; -1
(- 1 2 3)  ; -4

(not true)     ; false

(uid +)        ; unique id of +

(len '(1 2 3))          ; length of vector:   3
(len [abc])             ; length of string:   3
(len (hash-map 1 |a|))  ; length of hash-map: 1

(define number [my number] 1)
(++ number)                     ; number is 2
(-- number)                     ; number is 1
(+= number 2)                   ; number is 3

(+ [Hello] [ ] [World.] [ Bye.]) ; Hello World. Bye.

(define greeting [my greeting] [Hello])
(+= greeting [ World!])                ; Hello World!
(+= greeting |!|)                      ; Hello World!!

(+= '(1 2 3) '(4 5 6))                 ; (1 2 3 (4 5 6))

(define capitals [country -> capital] (hash-map [United Kingdom] [London] [Spain]
[Madrid]))
(+= capitals [Nepal] [Kathmandu])
(+= capitals '([Norway] [Oslo] [Poland] [Warsaw]))
(+= capitals (hash-map [Peru] [Lima]))

(-= 10 1 2 3)                 ; 4
(-= |x| 2)                    ; |v|
(-= |x| |!|)                  ; |W|
(-= [abba] |a|)               ; [bb]
(-= (hash-map 1 |a| 2 |b|) 1) ; (hash-map 2 |b|)
(-= '(1 1 2 2 3) 1 2)         ; (3)
```

Operators `++`, `--`, `+=`, `-=`, `==` and `!=` are similar to those in C. Note also that **the operands need not to be numbers**. You can, for example, use `+=` to push values to the back of a vector, add characters into

a string, write values into a port, etc.

WARNING	All the examples above should work, but support for non-numeric types is not finished on most of the operators. Using those operators with non-numeric arguments aborts the program or returns error. Complete support for non numeric operands for the above operators should be implemented in the standard library eventually.

Branching and selection

Here are some examples using `if`:

```
(if true  [Yes])         ; Yes
(if false [Yes])         ; nil
(if false [Yes] [No])    ; No

(if true  (println [Yes]) (println [No]))         ; Prints Yes

(if true  (do (println [Yes]) (println [OK])))    ; Prints Yes|newline|OK
```

Here are some examples using `select`:

```
(select true [Yes])           ; Yes

(select false [No]
        true  [Yes])          ; Yes

(select default [Yes])        ; Yes

(select false   [No]
        default [Yes])        ; Yes

(select false   [No]
        true    [Maybe]
        default [Yes])        ; Maybe

(select false [Yes])          ; nil


(define f1 [f1] '() (fn () true))
(define f2 [f2] '() (fn () false))

(select (f1)  [Yes]
        (f2)  [No]
        false [Maybe])                    ; Yes

(select (f1)  (println [Sun is shining])
        (f2)  (println [It rains])
        false [Maybe]
        false 2
        false 3.14
        false |a|
        false [There can be as many selectors as needed])   ; Prints: Sun is shining
```

Formatted and regular printing

Here are few examples:

```
(print   [Hi])   ; Prints Hi without newline
(println [Hi])   ; Prints Hi and newline

(define name1  [some name 1] 'Jim)
(define name2  [some name 2] 'Alice)
(define number [some number] 30)

(println [Hi {} and {}! It is {} degrees outside.] name1 name2 number)  ; Prints Hi
Jim and Alice! It is 30 degrees outside.
```

Formatted string creation

Here are few examples:

```
(define name1  [some name 1] 'Jim)
(define name2  [some name 2] 'Alice)
(define number [some number] 30)

(string-format [Hi {} and {}! It is {} degrees outside.] name1 name2 number)  ;
Creates a sting [Hi Jim and Alice! It is 30 degrees outside.]
```

Functions with variable number of arguments

Here are few examples:

```
(define f [f] '(x [x] ... [varargs]) (fn (x ...) x))

(f 1 2 3)   ; (1 2 3)


(define f [f] '(x [x] y [rest of the args] ... [varargs]) (fn (x y ...) (println x)
(println y)))

(f 1 2 3)   ; Prints 1|newline|(2 3)
```

Environments

Here are few examples:

```
(env-global)
(env-current)
(env-new)
```

Returning from functions early

The value of the last expression of function is usually the return value from that function. However, by using `return` one can return early and have multiple exit points from a function. Small example:

```
(define errorCode [0 means no error.] 1)

(define start-engine [Start engine if all OK] '() (fn ()
    (if (!= errorCode 0) (return [Cannot start the engine]))
    ; .... Start the engine here...))
```

`Return` can be called with zero or one argument. If no arguments are given, then `return` will return the value `nil`. Short example:

```
((fn () (return nil)))   ; Evaluates into 'nil'.
((fn () (return)))       ; Evaluates into 'nil'.
```

Evaluating values

Special `eval` can be used to evaluate a given value. It can be called with one or two arguments. The second argument, if present, must be an environment that is used while evaluating. If no environment is given, the global environment is used instead.

`Eval` is useful, for example, in situations where you build the name of the function to be called at runtime. Small example:

```
(define level-next [next level] '() (fn ()
    (level-reset)

    (define lnum [level number] (+ level-current-number 1))

    (if (> lnum number-of-levels) (= lnum 1))

    (define name-of-fn-to-call [name of the level builder function to call] 'level-)
    (+= name-of-fn-to-call lnum)
    (eval ((eval name-of-fn-to-call)))))
```

Input and output ports

Input and output can be done through ports. Ports can be created and attached to different sources and sinks of data (for example the file system).

VM will prevent access to those resources that are not configured to be allowed. For example, trying to open a file in the REPL will result in error if the **octaspire-dern-repl** was not started with flag `-f` or `--allow-file-system-access`.

Here is small example:

```
(define f [f] (io-file-open [/path/goes/here.xy]))

(port-read f)
(port-read f 3)

(port-write f 65)
(port-write f '(65 66 67))
```

Ports can be explicitly closed, but it is not required; port will close automatically when the garbage collector collects it. Some ports might also support **seeking**, **distance measurement**, **length measurement** and **flushing**. Here is another small example:

```
(define f [f] (io-file-open [/path/goes/here.xy]))

(port-seek f -1)  ; Seek to the end
(port-write f 65)

(port-seek f 0)   ; Seek to the beginning
(port-write f 65)

(port-seek f -2)  ; Seek to one octet from the end
(port-write f 66)

(port-seek f 1)   ; Seek to one octet from the beginning
(port-write f 65)

(port-seek f  1 'from-current)  ; Seek one octet forward  from the current position
(port-seek f -1 'from-current)  ; Seek one octet backward from the current position

(port-dist f)     ; Tell the distance (in octets) from the beginning of the port

(port-length f)   ; Tell the size (in octets) of the port

(port-flush f)    ; Buffer is flushed to disk. Happens also automatically on close.
(port-close f)    ; Close port. This happens also automatically.

(port-length f)   ; -1
```

Input ports can be iterated with `for` in similar way that containers are iterated:

```
(define f [f] (io-file-open [/path/goes/here.xy]))

(for i in f (println i))           ; Print every octet

(port-seek f 0)                    ; Seek to the beginning
(for i in f step 2 (println i))    ; Print every other octet

(port-seek f 0)                    ; Seek to the beginning
(for i in f step 3 (println i))    ; Print every third octet
```

`io-file-open` will open a file for reading and writing, `input-file-open` will open a file only for reading and `output-file-open` will open file only for writing.

Below is short example about querying a port for supported operations:

```
(define f [f] (io-file-open [/path/goes/here.xy]))

(port-supports-output? f)          ; true
(port-supports-input?  f)          ; true

(define f [f] (output-file-open [/path/goes/here.xy]))

(port-supports-output? f)          ; true
(port-supports-input?  f)          ; false

(define f [f] (input-file-open [/path/goes/here.xy]))

(port-supports-output? f)          ; false
(port-supports-input?  f)          ; true
```

You can use `port-write` and `+=` to write to a port octets with values `integer`, `character`, `string` and `vector` of these types. Example:

```
(define f [f] (io-file-open [/path/goes/here.xy]))

(+= f |a| |b| [ cat] |!|)  ; ab cat!

(port-write f '(65 |A| [ Hi!])) ; AA Hi!
```

Converting between types

TODO

Searching and indexing

TODO

Loading libraries with `require`

Dern has support for loading libraries or "plugins" during run time with the builtin `require`. Before loading the requested library, `require` checks whether the library is already loaded, and loads it only if it isn't already loaded.

It first tries to find a source library (.dern file) with the given name. If it finds, it loads that. Next it tries to find a binary library (.so file in Unix) and loads that if found.

So, in the example below, `require` tries first to find file named **mylib.dern** and then, if the system is Unix, file named **libmylib.so**.

Here is small example:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

If `mylib`-library is required later again, there is no need to search and load it again, because `require` know that a library with that name is already loaded.

Below is a small example of a binary library for **Linux**, **FreeBSD**, **NetBSD**, **Haiku** and **MINIX 3** systems.

mylib.c

```
/***
  To build this file into a shared library in Linux system:

  gcc -c -fPIC mylib.c -I ../../../include -I ../../../external/octaspire_core/include
  gcc -shared -o libmylib.so mylib.o
***/
#include <stdio.h>
#include <octaspire/core/octaspire_helpers.h>
#include "octaspire/dern/octaspire_dern_vm.h"
#include "octaspire/dern/octaspire_dern_environment.h"

octaspire_dern_value_t *mylib_say(
    octaspire_dern_vm_t *vm,
    octaspire_dern_value_t *arguments,
    octaspire_dern_value_t *environment)
{
    OCTASPIRE_HELPERS_UNUSED_PARAMETER(environment);

    if (octaspire_dern_value_as_vector_get_length(arguments) != 1)
```

```
    {
        return octaspire_dern_vm_create_new_value_error_from_c_string(
            vm,
            "mylib-say expects one argument");
    }

    octaspire_dern_value_t const * const messageVal =
        octaspire_dern_value_as_vector_get_element_at_const(arguments, 0);

    if (messageVal->typeTag != OCTASPIRE_DERN_VALUE_TAG_STRING)
    {
        return octaspire_dern_vm_create_new_value_error_from_c_string(
            vm,
            "mylib-say expects string argument");
    }

    printf("%s\n", octaspire_dern_value_as_string_get_c_string(messageVal));

    return octaspire_dern_vm_create_new_value_boolean(vm, true);
}

bool mylib_init(octaspire_dern_vm_t * const vm, octaspire_dern_environment_t * const
targetEnv)
{
    octaspire_helpers_verify(vm && targetEnv);

    if (!octaspire_dern_vm_create_and_register_new_builtin(
        vm,
        "mylib-say",
        mylib_say,
        1,
        "mylib says something",
        targetEnv))
    {
        return false;
    }

    return true;
}
```

See directory doc/examples/plugin in the source distribution for an example with Makefiles for different systems.

Building and using a binary library in Haiku

Run these commands from the **build**-directory of the source distribution:

```
make -C ../doc/examples/plugin -f Makefile.Haiku
LIBRARY_PATH=$LIBRARY_PATH:../doc/examples/plugin ./octaspire-dern-repl -c
```

Write into the REPL:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

Building and using a binary library in MINIX 3

Run these commands from the **build**-directory of the source distribution:

```
make -C ../doc/examples/plugin -f Makefile.MINIX3
LD_LIBRARY_PATH=../doc/examples/plugin ./octaspire-dern-repl -c
```

Write into the REPL:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

Building and using a binary library in Linux

Run these commands from the **build**-directory of the source distribution:

```
make -C ../doc/examples/plugin
LD_LIBRARY_PATH=../doc/examples/plugin ./octaspire-dern-repl -c
```

Write into the REPL:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

Building and using a binary library in FreeBSD

Run these commands from the **build**-directory of the source distribution:

```
make -C ../doc/examples/plugin -f Makefile.FreeBSD
LD_LIBRARY_PATH=../doc/examples/plugin ./octaspire-dern-repl -c
```

Write into the REPL:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

Building and using a binary library in NetBSD

Run these commands from the **build**-directory of the source distribution:

```
make -C ../doc/examples/plugin
LD_LIBRARY_PATH=../doc/examples/plugin ./octaspire-dern-repl -c
```

Write into the REPL:

```
(require 'mylib)
(mylib-say [Hello world from library])
```

Embedding in C programs

This section is not ready yet. In the meantime you can see Dern in "real" use at: Maze and Lightboard.

Octaspire Maze and Lightboard are games (work in progress) that are being written using Dern and C99.

Tool support

etc-directory of the source distribution contains syntax files for **vim**, **emacs** and **pygments**.

Building the development repository

To build Dern without the unit tests, replace **cmake ..** with **cmake -DOCTASPIRE_DERN_UNIT_TEST=OFF ..** in the instructions that follow.

Raspberry Pi, Debian and Ubuntu

To build Dern from the regular source distribution in Raspberry Pi (Raspbian), Debian or Ubuntu (16.04 LTS) system:

```
sudo apt-get install cmake git
git clone https://github.com/octaspire/dern.git
cd dern/build
cmake ..
make
```

Arch Linux

To build on Arch Linux (Arch Linux ARM) system:

```
sudo pacman -S cmake git gcc make
git clone https://github.com/octaspire/dern.git
cd dern/build
cmake ..
make
```

Haiku

To build on Haiku (Version Walter (Revision hrev51127) x86_gcc2):

```
pkgman install gcc_x86 cmake_x86
git clone https://github.com/octaspire/dern.git
cd dern/build
CC=gcc-x86 cmake ..
make
```

FreeBSD

To build on FreeBSD (FreeBSD-11.0-RELEASE-arm-armv6-RPI2) system:

```
sudo pkg install git cmake
git clone https://github.com/octaspire/dern.git
cd dern/build
cmake ..
make
```

NetBSD

To build on NetBSD (NetBSD-7.1-i386) system:

```
sudo pkgin install cmake git
git clone git://github.com/octaspire/dern
cd dern
perl -pi -e 's/https/git/' .gitmodules
cd build
cmake ..
make
```

MINIX 3

To build from the regular source distribution on MINIX 3 (minix_R3.3.0-588a35b) system:

```
su root
pkgin install cmake clang binutils git-base
exit
git clone git://github.com/octaspire/dern
cd dern
perl -pi -e 's/https/git/' .gitmodules
cd build
cmake ..
make
```

Other systems

On different systems the required commands can vary. In any case, you should install a **C compiler**, **cmake** and **git**. Depending on the system, you might need to install also either **make** or **ninja**.

This is all there should be to it; **octaspire core** is included as a git submodule and it should be updated and be build automatically, so when make finishes, everything should be ready.

Running Dern

To run the unit tests:

```
test/octaspire-dern-test-runner
```

To start the REPL with color diagnostics (requires support for ANSI color escapes):

```
./octaspire-dern-repl -c
```

To see the allowed options run:

```
./octaspire-dern-repl -h
```

NOTE Man pages are not ready yet.

Using the amalgamated source

etc-directory of the source distribution contains amalgamated version of the source code. All the headers, implementation files and unit tests are concatenated with a script into a single file. This one file is all that is needed to use Octaspire Dern. The same single file can be used to (by giving different compiler flags):

- as an include in a project that wants to embed the Dern language
- as a stand-alone Dern REPL
- as a stand-alone unit test runner