USAGE

Initialize a project:

```
$ schemer init project-name
```

Run a project (in the projects' directory):

```
$ shemer run [--] arguments for program
```

Run a standalone file

```
$ shemer file-name.scm [--] arguments for program
```

FUNCTIONS

Chibi scheme is used as the language for schemer, it is extended with various functions.

`(text string x y)` dra ws *string* at *(x y)*

`(define-background-color c)` defines background color

`(define-font-color c)` defines font color

`(draw-square c x y w h)` dra ws a filled square, colored *c*, at *(x y)* with size *w h*

`(draw-line c x1 y1 x2 y2)` dra ws a line from *(x1 y1)* to *(x2 y2)* with color *c*

`(get-window-size)` returns *(window-width window-height)*

`(is-mouse-pressed)` returns *#t* or *#f*

`(get-mouse-pos)` returns *(mouse-x mouse-y)*

`(get-key-pressed)` returns currently pressed key

`(is-key-pressed x)` checks if *x* is pressed

`(system s)` calls system command *s* (see EXTENSIONS -> core -> *sys*)

`(dont-init-graphics)` - do not init raylib (see EXTENSIONS -> core -> *set-window-option*)

`(set-window-size w h)`- set windo w size to *w* by *h*

`(set-window-resizable b)`- set windo w to resizable or not (see EXTENSIONS -> core -> *set-window-option*)

`(rand)` - get a random float from 0 to 1

`(ffi-load ...)`- (see EXTENSIONS -> f fi)

`(ffi-call ...)`- (see EXTENSIONS -> f fi)

`(ffi-define ...)`- (see EXTENSIONS -> f fi)

`(use string)` adds a library or a file. if string is one of:

MAIN LOOP

When graphics are loaded, schemer executes *(on-load)*, which should be defined in your program.

Every frame, schemer will call *(update-screen)*, and it **hopes** it is defined.

TUTORIAL

todo. (lmao)

well, you probably want to

```
(use "core")
```

because it includes some *nice* functions, and init-7.scm from chibi-scheme that defines important stuff.

then you need to define

```
(define update-screen
  (lambda ()
    (text "hello" 50 50)))
```

and you're good to go

EXTENSIONS

Extensions are loaded using the `(use)` method. The builtin ones are defined in scm/* and compiled into schemer. Loading builtin extensions is as simple as `(use "ext-name")`, where *ext-name* is the file name without the extension (so for *scm/colors.scm* - the colors extension - you would write `(use "colors")`).

List of extensions and their description:

- *colors*, defines colors using kebab-case (look in *scm/colors.scm* for a list)
- *click*, defines functions that can easily register clicks for ui elements:
  - `(on-click rect f drawf)`- where *r ect* is a list *'((x1 y1) (x2 y1))* containing the rectangle that is the click area, *f* is a function that needs to be called when the *rect* is clicked, and *drawf* is a function that is called every time a frame is drawn, to draw the click obj.
  - `(handle-click)` - needs to be called every frame (in *(update-screen)*).
- *core*, imports the standard scheme functions, and defines:
  - `(print s)`- where s is the printed string
  - `(get-window-width)` - returns window width
  - `(get-window-height)` - returns window height
  - `(range-stepped from to step)`- returns a list of range from *fr om* to *to* stepped by *step*
  - `(range from to)`- returns a list of range from *fr om* to *to* stepped by 1
  - `(sum l)`- sums list l by applying +
  - `(avg l)`- return a verage of numbers in *l*
  - `(last l)`- returns last v alue of list *l*
  - `(flatten l)`- returns flattened *l*
  - `(bool->string b)`- returns boolean *b* as string
  - `(->string x)`- returns stringified x
  - `(sys x)`- e xecutes *(system)* with *x*, where *x* stringified with *->string*, so can be a list of arguments
  - `(set-window-option opt)`- sets windo w option `opt`, where opt is one of:
    - *"nowindow"* - don't initialize graphics, or close graphics
    - *"noresizable"* - disallow resizing the window
    - *"resizable"* - allow resizing the window
    - list of opts
  - `(set-nth l n v)`- returns *l* with v alue on position *n* changed to *v*
  - `(file->char-list path)`- returns file *path* as a list of characters
  - `(string-replace-char s c1 c2)`- replace *c1* with *c2* in *s*
  - `(close-window)` - closes the window, or - if called before loading - disables it.
  - `(achange asc k v)`- change the v alue referenced by *k* in the association *asc* to *v*, do nothing if *k* doesn't exist.
  - `(aput asc k v)`- same as abo ve, but append *'(k v)* to *asc* if *k* doesn't exist
  - `(keys asc)`- returns all k eys of association *asc*
  - `(filter f l)`- filters *l* by applying *f*

- `(has l x)`- checks if *x* is contained in *l*
- `(split-string s split)`- splits string *s* on *split*
- `(get-args)` - returns parsed *argv* as '((arg (("-arg" "value") ("-other" "value"))) (optarg ("optarg1" "optarg2"))) for program called like: `./program.scm -arg=value -other=value optarg1 optarg2`

- *plot*, defines functions, and variables for drawing plots:
  - `default-plot-options` - default list of options passed to *(plot)*
  - `(plot-set-xy opt v)`- sets x and y axis for opt *opt*, where v = *'((x-values) (y-values))*, e.g. *'((0 1 2 3 4 5) (0 1 2 3 4 5)))* (y = x)
  - `(plot opt)`- plots *opt*
- *shapes*, defines functions for drawing shapes. All of them return values that can be then applied as a *rect* for *(on-click)*:
  - `(rect c x1 y1 x2 y2)`- dra ws a not-filled rectangle of color *c* from *'(x1 y1)* to *'(x2 y2)*.
  - `(intersect? rect1 rect2)`- checks if *r ect1* intersects *rect2*
  - `(point-in-rect? pt rect)`- checks if point *pt* is in *r ect*
- *make*, defines functions intended to be used in *make.scm*
  - `(define-resource path)`- adds *path* to the b undle.
  - `(define-source path)`- same as abo ve
  - `(set-executable-name target s)`- sets the e xecutable name for *schemer build*
  - `(set-target target)`- sets the tar get for compilation. *tar get* can be one of: *win64* - for mingw cross-compilation, *local* for local CC call. ***warning:*** win64 cross-compilation is in a very pre-pre-alpha stage it may work, but it also may not, and the second option is more probable.
  - `(make)` - write required flies. Always call it at the bottom of *make.scm*
- *game2d*, defines some helper functions for 2d games:
  - `(sprite draw move)`- both *dr aw* and *move* are called each frame with *self* as an argument, where self is an *list-ref* of the sprite in an internal *g2d-sprites* list. It [*self*] can be then used to access values of the sprite.
  - `(spr-x spr)`- gets *x* from sprite *spr*
  - `(spr-y spr)`- gets *y* from sprite *spr*
  - `(sspr-x spr x)`- sets *x* for sprite *spr*
  - `(sspr-y spr y)`- sets *y* for sprite *spr*
  - `(spr-v spr v)`- gets additional v alue *v* from sprite *spr*
  - `(sspr-v spr v val)`- sets *val* as an additional v alue *v* for sprite *spr*
  - `(spr2rect spr)`- gets sprite *spr*'s rect, assuming that *(spr -v spr 'w)* is the sprites' width, and *(spr-v spr 'h)* is the sprites' height
  - `(game2d)` - loops over every sprite and calls their functions. Call it in *update-screen*.
- *ffi*, defines helpers for *ffi-load*, and *ffi-call*
  - `(ffi lib data)`- loads functions from *lib*. data should be a list defining functions, e.g.:*'((void putchar (int) c-putchar) (int strlen (ptr) c-strlen))* will define *(c-putchar)* that takes an *int*, and returns nothing, and *strlen*, that will take a *pointer* and return an *int*. the functions will be then defined in the top level, so you can then call them by their name (the last value in the *data* list, so in the given example *c-strlen*, and *c-putchar*). they can be later caller like this *(c-strlen "abcd")*. seee *xamples/ffi.scm*.

- `(find-library s)`- finds path of library *s* on de vice. e.g.:*(find-libr ary "c")* -> */usr/lib/libc.so.97.1*

EXAMPLES

- See *examples/*.
- paint (https://github.com/krzysckh/paint)

THE "COMPILER"

The executable built will be big and clumsy, because *compiler.c* is **not** a compiler, but more of a bundler. It bundles all defined files in one executable. It does it by compiling every resource (images, *.scm, etc.) to a *.c* file, that contains a list of bytes of the file, and some getters. It **IS** a braindead approach, but i don't like having dynamic dependencies scattered all over my system, so that was my idea.

All resources and sources should be defined in *make.scm*, using *(define-resource)*. The bundler will then know what files to bundle (lmao), and on runtime, when files are accessed through schemers' builtin functions with given filenames, it will **not** read them from disk, but load them through the getters (when compiled).

For example, if *make.scm* defines:

```
(use "make")
(define-resource "scm/init.scm")
(define-resource "res/image.png")

(make)
```

and *scm/init.scm* defines:

```
(define img #f)

(define on-load
  (lambda ()
    (set! img (load-image "res/image.png"))))

(define update-screen
  (lambda ()
    (show-image img 0 0)))
```

After running:

```
$ schemer build
$ ./a.out
```

res/image.png will be copiled to *build/resources/image.png.c*, and then built into *a.out*, and *then* accessed not from the disk, but using a getter defined in the *.c* file.