Project Citron – A totally-not-definitive Guide

This is an *obviously* free book detailing the use of the Citron language; It serves as a tutorial for intermediate audience.

This text assumes a basic knowledge about common programming terms and concepts.

Start Here

Dedication

To myself.

what, did you expect anything interesting? # Summary

Preface

TODO # About Citron

Citron is a language that tries to explore the ideas and combine things that have barely ever been combined.

It is by no means meant to be used for serious projects and is *as of yet* in a beta stage.

Baby Steps

Clearly, every language requires a 'Hello, World!' program.

There are mainly three ways of executing citron code:

- with the main executable `ctr`
- with the scratchpad (neat thing, tbh) (see Scratchpad)
- with the (JIT) interpreter
- compiling it with `ctrc` and executing that [this only generates a simple program that links into the Citron runtime, it is not static]

Using the Interpreter Prompt

Presumably, you have a terminal window open from back when you compiled citron from source, so switch to it,

and start the interpreter by executing `citron` and pressing `<return>`

Once you have started the interpreter, it will greet you with a nice-looking ascii art,

and the main prompt (default)

Type `Pen writeln: 'Hello, World!'` followed by the return key.

You shall see the magical words `Hello, World!` printed to the screen. (and a `[_:Object]` which signifies the object `Pen`; in general, Objects without the method toString are shown in this format: `[<name in current context>:Type]`)

Here's an example of what you might expect to see:

```
$ citron
Terminal: 24x83 - 0x0

  _____ _ _
 / ____(_) |
| |     _| |_ _ __ ___  _ __
| |    | | __| '__/ _ \| '_ \  Project Citron
| |____| | |_| | | (_) | | | |
 \_____|_|\__|_|  \___/|_| |_|

0.0.8.8-boehm-gc [GCC/G++ 8.2.0]
 Pen writeln: 'Hello, World!'
Hello, World!
[_:Object]
```

How do I quit this interpreter?

Type `:q` and press return.

Or just press Ctrl + D.

or if you really like punching in expressions, evaluate `Eval end`

Choosing an editor

Currently, Atom, TextMate and Sublime 3 syntax files are generated and maintained.
However, minimal Smalltalk syntax highlighting will do in a pinch

Using a source file

Back to coding.

Open a new file [I have used `main.ctr` for most of the files] (generic extension is `.ctr`) and type this in:

```
Pen writeln: 'Hello, World!'.
```

Then simply run `ctr` with the sole argument being the path to the file.

```
$ ctr main.ctr
Hello, World!
```

Basics

Printing `Hello, World!` is always exciting and all, but it's never enough…maybe some inputs, processing, and some actually useful output is always preferred!

Comments

Comments always span a signle line, beginning with the character `#`

(Do note that the interactive interpreter will not allow comments)
For example:

```
Pen writeln: 'Hello, World'. #Totally ignore me, okay?
```

or:

```
#Pen is an Object, and 'writeln:' is a message
Pen writeln: 'HELLO!'.
```

Some people like comments, so make sure to use them lots, okay?

Literals

An example of a literal is a number, like `5`, `3.14`, `0xAF` or a string `'Hello'`.

Numbers

The only numeric type available to the programmer is `Number`, which can store up to a 64-bit double's worth.

(`BigInteger` is available for arbitrary precision Integer numeric values as well)

Strings

String literals are created in two ways:

- Strings with processed escape codes – which are enclosed in single quotes: `'Hello, world!\n`'
- 'Raw' Strings, which are quoted literally (mainly used for regular expressions, or embedding other languages), enclosed in `?>` and `<?` : `?>\s\t\w<?`

Both of these can span multiple lines.

String formatting

clearly everyone needs string formatting

Two solutions are provided:

- Embedding a variable inside the string: `Hello, $$name !'`
- Using the format methods: `'Hello, %s!' % [name]`
- or the other format method: `'Hello, %{name}' %~: (Map fromArray: [['name', 'What']]`

The first solution is used as a quick-and-dirty way to embed single variables inside a string, and are processed by the lexer to a string addition expression: `'Hello, ' + name + ' !'`

The second one uses a method of the String object, `%` which provides printf-like string formatting

```
var age is 21.
var name is 'Semicolon'.

# Note that this $$var is not very versatile, it will only break on whitespace and such
Pen writeln: 'Hello, $$name you seem to have an age of $$age'.
Pen writeln: 'Hello, %s you seem to have an age of %d' % [name, age].
```

Escape sequences

The normal, basic sequences are supported.

Also a way of embedding characters as hexadecimal is provided: `\x12 or \x{12}`

which will consume all the hexadecimal digits it can.

As well as unicode characters: `\u2002 or \u{2002}`

Tuples

Contrary to what you might be used to, Citron's tuples are generated at *parse time* and have a (pretty much) immutable structure

They are created like so:

```
[element0, element1] # A tuple with two elements. you can treat this like an Array object
[] # An empty tuple
```

Code Blocks

A code block is just like a normal function, except the fact that it is *anonymous*.

It may or may not be a closure.

There are two main code block types:

- Basic code block – which is always executed whenever it is required to. `{ (:param)* (expression.)* }`
- Lexically scoped code block – which always captures every value that is not in its formal parameters. `{\ (:param)* (expression.)* }`

To return from a block, the return token `^` is used.

A shorthand for a lexical code block with a single expression and at least one parameter (quite like a lambda) exists: `\(:param)+ expression`

which is translated to `{\(:param)+ ^expression.}` by the parser.

Object

Anything and everything is an Object in Citron. *No Exceptions*

(This does not mean that Citron *enforces* the idea of Object-Oriented Programming, but that is the most simple way of using the language)

Code Lines & Semantics

All the statements *must* end in a dot (`.`)

You may put all the statements in a single line (why would you want to?)

Whitespace is ignored, and has virtually no meaning.

Assignment

There are three main forms of assignment in Citron, all of which have the general from `expr (:= | is) expr`

1. assignment to an existing reference: `name is value`

2. creation and assignment to a new reference: `(var | my) name is value`
3. assignment by deconstruction: `type_var is value`, which we will get back to in another chapter.

Every reference can optionally have three (four) different modifiers (Only one can be active at any given time):

1. `var` : simply creates a new reference if no old ones exist, otherwise binds to them
2. `my` : creates or binds to a property of the current object.
3. `const` : tries to capture a variable from the environment around the currrent object
4. `frozen`: if the XFrozen pragma is active. expression is only evaluated once.

There is a shorthand for assignment to an object property (`my property is value`): `property => value`

which will prove useful in certain points (which we will explore in later chapters).

Control Flow

There are pretty much *no* keywords in citron, thus all control flow operations are carried out by interfacing the `Boolean` Object type.

The equivalent to a if-else structure

The method is `Boolean::'either:or:'`, alternatively, when one of the two branches is not needed, you may use `Boolean::'ifTrue:'` or `Boolean::'ifFalse:'`

All of these take a block of code (in a basic form, code contained in `{` and `}`), and execute it if their condition is matched.

An example

```
var a is 123.
a > 5 either: {
    Pen writeln: '$$a is a big number!'.
} or: {
    Pen writeln: '$$a is a cute little number!'.
}.
```

Should you return a value from the `either:or:` blocks, the whole expression will evaluate to that value.

However, remember that returning from `ifTrue:` or `ifFalse:` will return **two** leves instead of one.

Loops

Again, no keywords, so looping is achieved through interfacing Number (a for loop), or a block (a while loop).

Repeating an expression

`Number::'times:'` to the rescue!

That method will run the given block with the iteration index.

```
10 times: {:value
    Pen writeln: value.
}.
```

Looping over a range

`Number::'to:step:do:'` is a basic way of looping over a range.

```
5 to: 50 step: 5 do: {:value
    Pen writeln: value.
}.
```

While loops

To achieve a while loop, use `CodeBlock::'whileTrue:'` or `CodeBlock::'whileFalse:'`

```
var a is 123.
{^a sin > 0.} whileTrue: {
    Pen write: a + ' '.
    a -=: 1.
}.
```

Break/Continue

Simply use `Boolean::'break'` or `Boolean::'continue'`

Should the given boolean be true, they will break/continue the loop.

Iterating over collections

Most if not all collections support these methods (if applicable)

- `each:` Simply iterates over the container, passing index(key), value, collection
- `each_v:` Iterates over the container, but only passes value, collection
- `fmap:` Iterates over the container, and replaces the element it has passed with the return value. passes only value
- `imap:` Iterates over the container, and replaces the element it has passed with the return value. passes index, value

```
var arr is Array < 10 ; 20 ; 30 ; 40 ; 50.
arr fmap: {:x ^x + 2.}. # => Array < 12 ; 22 ; 32 ; 42 ; 52
arr imap: {:i:x ^x + i.}. # => Array < 10 ; 21 ; 32 ; 43 ; 54
arr each: {:i:x Pen writeln: 'Index $$i = $$x'. }. # => returns arr, prints a bunch of stuf

var map is Map fromArray: [ ['Test', 'test'], ['KEY', 'key'], [[], 'Whoa'], [1234, []] ].
map fmap: {:value ^value toString reverse.}. # => (Map new) put:'tseT' at:'Test', put:'YEK'
map each: {:key:value Pen writeln: '$$key = $$value'.}. # => returns map, prints a bunch of
```

Data Structures

There are three basic builtin *native* structures in Citron - *Array*, *Tuple* and *Map*

which are further extended by non-native extensions - *Set* and *Generator*

Array

`Array` is basically a list of values, implemented as a contiguous array.

Its' elements can have any type, and it can be created literally by the `push:` method.

```
Array new push: 3, push: 'test', push: Nil
```

or with a shorthand:

```
Array < 3 ; 'test' ; Nil
```

Basic example

```
var shoplist := Array < 'apple' ; 'mango' ; 'pure chocolate' ; 'old memes'.

Pen writeln: 'There are ' + (shoplist count) + ' items to buy.'.
Pen writeln: 'Those items are: %:L' % [' ', shoplist].

Pen writeln: 'I also want some programming socks!'.

shoplist push: 'programming socks'.

Pen write: 'Now I have to buy all of these shticks:\n'.
shoplist each: {:idx:name
    Pen writeln: '\t$$idx - $$name'.
}.

Pen write: 'Such a bad list, let me sort it first: '.
shoplist is shoplist sort: {:a:b
    # now compare items a and b
    ^(a length) > (b length). #Whichever has a longer name last
}.
```

```
Pen writeln: '%L' % [shoplist].

Pen writeln: 'I have bought this trash now: %s, and I have these left to buy: %L' % [shoplis
```

Which should give the output

```
There are 4 items to buy.
Those items are: apple mango pure chocolate old memes
I also want some programming socks!
Now I have to buy all of these shticks:
        0 - apple
        1 - mango
        2 - pure chocolate
        3 - old memes
        4 - programming socks
Such a bad list, let me sort it first: apple, mango, old memes, pure chocolate, programming
I have bought this trash now: apple, and I have these left to buy: mango, old memes, pure ch
```

Tuple

Tuples are - much like other languages that have them - like immutable arrays

Their syntax is quite simple: `[ element0, element1 ]`

Basic example

```
var zoo is ['level-headed lion', 'crazy snek', 'memeified pinguin'].
Pen writeln: 'There are %d animals in this zoo' % [zoo count].

#Any attempt at changing them deegrades them to an Array (or throws an exception)
var zoo1 is ['anime-loving pinguin'] + zoo.
Pen writeln: 'This zoo of %s is a very meme-like zoo, but this %s zoo is not!' % [zoo1, zoo]

#as normal, degraded tuples are just arrays
Pen writeln: zoo1 pop.

#But don't even try to pop something off them or such
Pen writeln: zoo pop.
```

Output

```
There are 3 animals in this zoo
This zoo of Array ← 'anime-loving pinguin' ; 'level-headed lion' ; 'crazy snek' ; 'memeifie
memeified pinguin
Uncaught error has occurred.
Cannot change immutable array's structure
#2 pop (test.ctr: 12)
#1 writeln: (test.ctr: 12)
```

Map

Maps are implemented as HashMaps, and respect the hash method provided by the object `object::'iHash'`

They do not have any literals associated with them.

Example

```
#They can be either constructed with Map::'put:at:'
var map is Map new put: 'World' at: 'Hello', put: 'Fish' at: 'Dead'.
#Or with Map::'cnew:'
var map1 is Map cnew: {
    Hello => 'World'.
    Dead => 'Fish'.
}.
#Or with Map::'fromArray:'
var map2 is Map fromArray: [
    ['Hello', 'World'],
    ['Dead', 'Fish']
].
Pen writeln: 'They serialize upon printing by default:\n' + map.

# You can add, modify, or remove assocs
map put: 'Guy' at: 'Dead'.
#That's sad
map deleteAt: 'Dead'.

#They can contain any object that implements iHash
map put: 1 at: 2, put: '1' at: 3, put: map at: 4. #Even themselves

Pen writeln: map.

#They can be iterated over:
map each: {:key:value
    Pen writeln: '%s, %s' % [key, value].
}.

#Or mapped to a map with different values
map2 is map fmap: \:key:value key + ', ' + value.

Pen writeln: map2.
```

Output

```
They serialize upon printing by default:
(Map new) put:'Fish' at:'Dead', put:'World' at:'Hello'
```

```
(Map new) put:':selfReference:' at:4, put:'1' at:3, put:1 at:2, put:'World' at:'Hello'
4, (Map new) put:':selfReference:' at:4, put:'1' at:3, put:1 at:2, put:'World' at:'Hello'
3, 1
2, 1
Hello, World
(Map new) put:'Hello, World' at:'Hello', put:'2, 1' at:2, put:'3, 1' at:3, put:'4, (Map new
```

Set

Sets are implemented basically as Maps, without values (they have a fixed value, which is shared between all sets).

No literals.

Example

```
# HashSet
import Library/Data/Set/HashSet: 'HashSet'.
# or as 'Set':
# import Library/Data/Set/HashSet: { HashSet => 'Set'. }

var set is HashSet new. # => {}
set add: 1, add: 2. # => {2, 1}

# Or maybe if you have a bunch
set addAll: ['test', 3]. # => {3, 'test', 2, 1}

# Remove an element
set remove: 'test'. # => {3, 2, 1}

# Check if an element exists
set contains: 'test'. # => False

# Bloom Filter
import Library/Data/Set/BloomFilter: 'BloomFilter'.

# These sets are neat, they provide a definite not-existence answer only
# That is, you cannot retrieve data from them, nor can you reliably remove data from them
# But they only contain a Number, and can be a fast low-confidence filter

var bf is BloomFilter new.
bf add: 'test', add: 'something else', add: 64. # => [BloomFilter]

bf contains: 4. # => False
bf contains: 'test'. # => True
```

Generators

Generators are lazy list generators, they have a few helpful basic functionalities as well.

`Number..Number` and `Number..Number..Number` exists as a literal (see example)

Example

```
# You can make a simple step generator with numbers:
var gen0 is 0..10. # => [StepGenerator]
# With a step value
var gen1 is 0..2..10. # => [StepGenerator]

# Or through messages to Generator
var gen2 is Generator from: 0 to: 10.
var gen3 is Generator from: 0 to: 10 step: 2.
var gen4 is Generator repeat: Nil. # This will just make `Nil's forever, it's useful to map
var gen5 is Generator elementsOf: [1,2,3,4]. # Makes a generator from a collection (Array,Ma

# You can get the next value
gen0 next. # => 0
gen0 next. # => 1

# You can map them to a new generator
var gen6 is gen0 fmap: \:x x + 3.
# Note that advancing one will advance the other too
gen6 next. # => 5
gen0 next. # => 3

# You can return a generator from a mapping, and `inext' will expand it
var gen7 is gen4 fmap: \:_ gen0 copy. # make the elements of gen0. forever.

gen7 inext. # => 4
gen7 inext. # => 5
(1..10) fmap: \:_ gen7 inext, toArray. # => Array ← 6 ; 7 ; 8 ; 9 ; 10 ; 6 ; 7 ; 8 ; 9 ; 10

# You can break in the middle of a mapping too, which will terminate the generator
```

List Comprehensions

Everyone likes list comprehensions, so here, you can have them too.

Literal: too many to list, take a look at the example

Example

```
var list0 is [x,, 1..10]. # Free variables are bound in order
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# You can specify any number of predicates
list0 is [x,, 1..10,, (x mod: 4, = 0)]. # Only multiples of 4
# => [4, 8]

list0 is [x,, 1..10,, (x mod: 4, = 0), (x > 4)]. # only multiples of 4 bigger then 4
# => [8]

# You can do without a source too
var conditionally10 is [10,,, False]. # => []
# Note the 3 commas

var a is 10.
var list1 is [a + x,, 1..3]. # bound names stay as they are
# => [13, 14, 15]

# More than one source (Generators not supported  yet)
var list2 is [x + y,, [1,2,3], [5,6,7]]. # => Array ← 6 ; 7 ; 8 ; 7 ; 8 ; 9 ; 8 ; 9 ; 10
var list3 is [x + y,, [1,2,3], [5,6,7],, x > y]. # => []

# You can specify the names as well
var list4 is [x + y,, (y: [1,2,3]), (x: [5,6,7]),, x < y, (x mod: y > 2)].

# You can return any citron object
var dispatch is Map fromArray: [[x, {\:arg arg at: x.}],, 1..5]. # returns a 2-tuple of key
# => (Map new) put:([:Block]) at:5, put:([:Block]) at:4, put:([:Block]) at:3, put:([:Block]

# Now call it!
dispatch at: 1, applyTo: [1,2,3]. # => 2
```

TODO: more Strings

Data Structures

There are three basic builtin *native* structures in Citron - *Array*, *Tuple* and *Map*

which are further extended by non-native extensions - *Set* and *Generator*

Array

`Array` is basically a list of values, implemented as a contiguous array.

Its' elements can have any type, and it can be created literally by the `push:` method.

```
Array new push: 3, push: 'test', push: Nil
```

or with a shorthand:

```
Array < 3 ; 'test' ; Nil
```

Basic example

```
var shoplist := Array < 'apple' ; 'mango' ; 'pure chocolate' ; 'old memes'.

Pen writeln: 'There are ' + (shoplist count) + ' items to buy.'.
Pen writeln: 'Those items are: %:L' % [' ', shoplist].

Pen writeln: 'I also want some programming socks!'.

shoplist push: 'programming socks'.

Pen write: 'Now I have to buy all of these shticks:\n'.
shoplist each: {:idx:name
    Pen writeln: '\t$$idx - $$name'.
}.

Pen write: 'Such a bad list, let me sort it first: '.
shoplist is shoplist sort: {:a:b
    # now compare items a and b
    ^(a length) > (b length). #Whichever has a longer name last
}.
Pen writeln: '%L' % [shoplist].

Pen writeln: 'I have bought this trash now: %s, and I have these left to buy: %L' % [shoplis
```

Which should give the output

```
There are 4 items to buy.
Those items are: apple mango pure chocolate old memes
I also want some programming socks!
Now I have to buy all of these shticks:
        0 - apple
        1 - mango
        2 - pure chocolate
        3 - old memes
        4 - programming socks
Such a bad list, let me sort it first: apple, mango, old memes, pure chocolate, programming
I have bought this trash now: apple, and I have these left to buy: mango, old memes, pure ch
```

Tuple

Tuples are - much like other languages that have them - like immutable arrays

Their syntax is quite simple: `[ element0, element1 ]`

Basic example

```
var zoo is ['level-headed lion', 'crazy snek', 'memeified pinguin'].
Pen writeln: 'There are %d animals in this zoo' % [zoo count].

#Any attempt at changing them deegrades them to an Array (or throws an exception)
var zoo1 is ['anime-loving pinguin'] + zoo.
Pen writeln: 'This zoo of %s is a very meme-like zoo, but this %s zoo is not!' % [zoo1, zoo

#as normal, degraded tuples are just arrays
Pen writeln: zoo1 pop.

#But don't even try to pop something off them or such
Pen writeln: zoo pop.
```

Output

```
There are 3 animals in this zoo
This zoo of Array ← 'anime-loving pinguin' ; 'level-headed lion' ; 'crazy snek' ; 'memeifie
memeified pinguin
Uncaught error has occurred.
Cannot change immutable array's structure
#2 pop (test.ctr: 12)
#1 writeln: (test.ctr: 12)
```

Map

Maps are implemented as HashMaps, and respect the hash method provided by the object `object::'iHash'`

They do not have any literals associated with them.

Example

```
#They can be either constructed with Map::'put:at:'
var map is Map new put: 'World' at: 'Hello', put: 'Fish' at: 'Dead'.
#Or with Map::'cnew:'
var map1 is Map cnew: {
    Hello => 'World'.
    Dead => 'Fish'.
}.
#Or with Map::'fromArray:'
var map2 is Map fromArray: [
    ['Hello', 'World'],
    ['Dead', 'Fish']
].
Pen writeln: 'They serialize upon printing by default:\n' + map.

# You can add, modify, or remove assocs
map put: 'Guy' at: 'Dead'.
#That's sad
map deleteAt: 'Dead'.

#They can contain any object that implements iHash
map put: 1 at: 2, put: '1' at: 3, put: map at: 4. #Even themselves

Pen writeln: map.

#They can be iterated over:
map each: {:key:value
    Pen writeln: '%s, %s' % [key, value].
}.

#Or mapped to a map with different values
map2 is map fmap: \:key:value key + ', ' + value.

Pen writeln: map2.
```

Output

```
They serialize upon printing by default:
(Map new) put:'Fish' at:'Dead', put:'World' at:'Hello'
(Map new) put:':selfReference:' at:4, put:'1' at:3, put:1 at:2, put:'World' at:'Hello'
4, (Map new) put:':selfReference:' at:4, put:'1' at:3, put:1 at:2, put:'World' at:'Hello'
3, 1
2, 1
Hello, World
(Map new) put:'Hello, World' at:'Hello', put:'2, 1' at:2, put:'3, 1' at:3, put:'4, (Map new)
```

Set

Sets are implemented basically as Maps, without values (they have a fixed value, which is shared between all sets).

No literals.

Example

```
# HashSet
import Library/Data/Set/HashSet: 'HashSet'.
# or as 'Set':
```

```
# import Library/Data/Set/HashSet: { HashSet => 'Set'. }

var set is HashSet new. # => {}
set add: 1, add: 2. # => {2, 1}

# Or maybe if you have a bunch
set addAll: ['test', 3]. # => {3, 'test', 2, 1}

# Remove an element
set remove: 'test'. # => {3, 2, 1}

# Check if an element exists
set contains: 'test'. # => False

# Bloom Filter
import Library/Data/Set/BloomFilter: 'BloomFilter'.

# These sets are neat, they provide a definite not-existence answer only
# That is, you cannot retrieve data from them, nor can you reliably remove data from them
# But they only contain a Number, and can be a fast low-confidence filter

var bf is BloomFilter new.
bf add: 'test', add: 'something else', add: 64. # => [BloomFilter]

bf contains: 4. # => False
bf contains: 'test'. # => True
```

Generators

Generators are lazy list generators, they have a few helpful basic functionalities as well.

`Number..Number` and `Number..Number..Number` exists as a literal (see example)

Example

```
# You can make a simple step generator with numbers:
var gen0 is 0..10. # => [StepGenerator]
# With a step value
var gen1 is 0..2..10. # => [StepGenerator]

# Or through messages to Generator
var gen2 is Generator from: 0 to: 10.
var gen3 is Generator from: 0 to: 10 step: 2.
var gen4 is Generator repeat: Nil. # This will just make `Nil's forever, it's useful to map
var gen5 is Generator elementsOf: [1,2,3,4]. # Makes a generator from a collection (Array,M
# You can get the next value
gen0 next. # => 0
gen0 next. # => 1

# You can map them to a new generator
var gen6 is gen0 fmap: \:x x + 3.
# Note that advancing one will advance the other too
gen6 next. # => 5
gen0 next. # => 3

# You can return a generator from a mapping, and `inext' will expand it
var gen7 is gen4 fmap: \:_ gen0 copy. # make the elements of gen0. forever.

gen7 inext. # => 4
gen7 inext. # => 5
(1..10) fmap: \:_ gen7 inext, toArray. # => Array ← 6 ; 7 ; 8 ; 9 ; 10 ; 6 ; 7 ; 8 ; 9 ; 10

# You can break in the middle of a mapping too, which will terminate the generator
```

List Comprehensions

Everyone likes list comprehensions, so here, you can have them too.

Literal: too many to list, take a look at the example

Example

```
var list0 is [x,, 1..10]. # Free variables are bound in order
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# You can specify any number of predicates
list0 is [x,, 1..10,, (x mod: 4, = 0)]. # Only multiples of 4
# => [4, 8]

list0 is [x,, 1..10,, (x mod: 4, = 0), (x > 4)]. # only multiples of 4 bigger then 4
# => [8]

# You can do without a source too
var conditionally10 is [10,,, False]. # => []
# Note the 3 commas

var a is 10.
var list1 is [a + x,, 1..3]. # bound names stay as they are
# => [11, 12, 13]
# More than one source
var list2 is [x + y,, [1,2,3], [5,6,7]]. # => Array ↤ 6 ; 7 ; 8 ; 7 ; 8 ; 9 ; 8 ; 9 ; 10
var list3 is [x + y,, [1,2,3], [5,6,7],, x > y]. # => []
var list4 is [x + y,, 1..3, 5..7]. # => Array ↤ 6 ; 7 ; 8 ; 7 ; 8 ; 9 ; 8 ; 9 ; 10

# You can specify the names as well
var list5 is [x + y,, (y: [1,2,3]), (x: [5,6,7]),, y < x, (x mod: y, < 1)]. # => Array ↤ 6
var list6 is [x + y,, (x: 1..3), (y: 5..7),, y - x < 5] # => Array ↤ 6 ; 7 ; 8 ; 8 ; 9 ; 10

# You can return any citron object
var dispatch is Map fromArray: [[x, {\:arg arg at: x.}],, 1..5]. # returns a 2-tuple of key
# => (Map new) put:([:Block]) at:5, put:([:Block]) at:4, put:([:Block]) at:3, put:([:Block]

# Now call it!
dispatch at: 1, applyTo: [1,2,3]. # => 2
```

TODO: more Strings

<- Prev Next ->

Objects

Objects are the main attraction in Citron (but are in no way forced)

You may instantiate any object by the default `Object::'new'` method (all objects should treat this method as a request for a new instance)

or directly execute code in the newly created object instance by using `Object::'cnew:'`

Objects can interface each other *only* by their methods (no way to access object fields directly if said object does not expose a method for it)

In each object's context o execution, an implicit `me` reference refers to the object itself, and the object's properties are accessible by `my` qualified references.

Dumb example

```
var Person is Object cnew: { name => 'Dummy'. }.
#To add a method, use Object::'on:do:'
Person on: 'new:' do: {:name ^me cnew: { name => name. }. }.
Person on: 'name' do: { ^my name. }, #You are most welcome to chain these should you wish t
 on: 'greet:' do: {:other
    #just assume other responds to 'name'
    Pen writeln: 'Hello, ' + other name + ', I am ' + my name.
}. #All such methods implicitly return `me' if nothing is explicitly returned.
```

```
var p0 is Person new: 'Idiot'.
var p1 is Person new: 'Fool'.

p0 greet: p1. #p1 supports ::'name'
#let's try with something that doesn't
p1 greet: 'Semicolon'.
```

Dumb output

```
Hello, Fool, I am Idiot
Uncaught error has occurred.
Unknown method String::'name' was called
#4 name (test.ctr: 7)
#3 + (test.ctr: 7)
#2 writeln: (test.ctr: 7)
#1 greet: (test.ctr: 15)
```

Inheritance

The usual single-inheritance rules apply (only to methods, and calling the parent constructor should be *explicit*)

```
var P is Object new on: 'method' do: { ^'Parent method'. }.
var C is P new.
#Overrides have no special syntax
C on: 'method' do: { ^'Child method'. }.
Pen writeln: C method. #=> 'Child method'
```

Multiple inheritance *is* supported, in a way.

You may 'inherit' from other objects and delegate method calls to their defintions:

```
var P0 is Object new on: 'test0' do: {^'test0'.}.
var P1 is Object new on: 'test1' do: {^'test1'.}.
var C is P0 new inheritFrom: P1.

#Now ::'test1' will properly work (it will not contain any references to P1)
C test1. #=> 'test1'
```

IO

Everyone adores I/O, well, not everyone, that poor processor that is always kept waiting sure doesn't.

The basic input methods are available with the `Program::'input'`, `Program::'waitForInput'` and `Program::'getCharacter'` (only available with termios) the more...sophisticated actions can be performed with `stdin` : `File special: 'sdin'` will give an auto-cleaning handle to it.

Input

```
var thing is Program waitForInput.
Pen writeln: 'in reverse: ' + thing reverse.
Pen writeln: 'SPACEY: ' + (thing characters join: ' ').
#This will raise an exception if any extra character remains after conversion
Pen writeln: 'Trying to make a number of it: ' + (thing toNumber).
```

Output for `123test`:

```
123test
in reverse:
tset321
SPACEY: 1 2 3 t e s t

Uncaught error has occurred.
cannot build number, extranous characters in string
#3 toNumber (test.ctr: 5)
#2 + (test.ctr: 5)
#1 writeln: (test.ctr: 5)
```

Palindromes, because every language needs one

```
var is_palindrome is {:str
    ^str reverse = str.
}.
{^True.} whileTrue: {
    Pen write: 'What be your text? '.
    Pen writeln: (is_palindrome applyTo: Program waitForInput trim, either: 'Yep, that\'s a
}.
```

Profoundly, the output:

```
What be your text? str
Nope, not a plaindrome
What be your text? 1001
Yep, that's a palindrome
What be your text? []
Nope, not a plaindrome
^C
```

Files

Files are fun things, they respond to `read`, `readBytes:`, `write:` and more!

Opening a file

You may implicitly open a file for reading or writing depending on the operation (no need to state `open:`)

```
#You may write to it normally
File new: 'test', write: 'This is a test\n' * 100.
#Or open it explicitly
File new: 'test', open: 'w+', write: 'This be a test\n', close. #Resources are cleaned by th
```

Special-purpose Files

there are three special file descriptors `stdin` `stdout` and `stderr`

You may access these by `File::'special:'`

Note that `stdin` does *not* allow writes, and the other two do not allow reads.

Exceptions

Exceptions in Citron are pretty much the same as all other languages; they occur when *exceptional* things occur!

Basic Errors

Errors can be generated with `Block::'error:'`

The interpreter itself also can generate errors for unkown keys, etc.

```
  Pen writenl: 'This will make an exception!'. #Note that the method name is spelled incorre
#=> Exception: Unknown method Object::'writenl:' was called
  pen writeln: 'This will do that too'. #Misspelled 'Pen'
#=> Exception: Key not found: pen
```

Raising and Handling Exceptions

All the exceptions generated by the interpreter are of type String, however a facility is provided to use and catch exceptions of different types:

```
{
    thisBlock error: Nil.
} catch: {:e Pen writeln: 'Nil!'.} type: Nil,
  run.

#=> Nil!
```

There is not much more involved, just note that exceptions are used only when no other way of handling the current situation exists.

For instance, a miss on a map lookup is not an exceptional thing, it will simply return Nil. ## Library / Imports

The import Object

Citron's answer to handling modular programs, is of course, the `import` Object.

Yes, you read that right; Object.

The basic import syntax is like so:

`import path/to/module/directory`

you may also directly import ctr files the same way:

`import path/to/file`

There are a few rules to the import mechanism:

- If a diretory is provided, there *must* be a `main.ctr` file inside that directory
- If the path to a file is provided, it *must* have a .ctr extension
- If a file is outside the search paths, it may be imported using absolute paths
- Only the directly exported names inside the module are imported, unless explicitly requested (more on this later)
- imported object will be added to the global scope of the program

You can change the name of the explicit imports by assigning them in a block (see example below)

Importing something from the standard library

The standard library is not very extensive, but it does have most of the essentials.

Example:

```
import Library/Data/Set/HashSet: 'HashSet'. # Will only import the object HashSet
import Library/Data/Set/HashSet: { HashSet => 'Set'. }. # Will only import the object HashS
import Library/Data/Map. # Will import all the exported values in that module
import Library/Functional/PatternMatch: \*. # Will import all the values in that module
```

Importing specific names from a module

The `import` object will try to import any name given to it in the following ways:

1. Import only the exported names: `import path/to/module`
2. Import one specific name from the module, in addition to the exports: `import path/to/module: 'whatever'`
3. Import a list of names from some module: `import path/to/module: <Array of string>`
4. Import a few names, specifying new names: `import path/to/module: { oldName => 'newName'. oldName2 => 'newName2'. }`

5. Import all the symbols in a module: `import path/to/module: \*`
    - `\*` is a Symbol, it is a kind of String that is allocated only once.

The standard library

The library is still a work in progress, but it should have the basic needs, and facilities to interface C APIs for less common needs.

The layout of the library closely resembles Haskell's, with the library being split into several categories.

The current categories (as of Citron 0.0.8) are:

- AST : contains simple interfaces to change or view the AST of a program
- Control : contains various modules to handle or change the program control flow
- Data : contains modules for data structures
- Executable : executable modules, which can be run with `citron -m <name>`
- Extensions : loads the old extensions (deprecated)
- Foreign : contains modules that handle the foreign functions interfacing
- Functional : contains modules that allow for a more functional style of coding
- Graphics : graphical things
- GUI : modules for creation and handling of user interfaces
- Net : modules that handle sockets and networks
- Utils : various utilities

The Control category

- Applicative : Adds several method to CodeBlock for convenience, for instance `apply:` and `compose:`
- Arrow : Adds an arrow method to Object to resolve methods and properties
- Class : WIP module that adds classification (To be replaced by Categorization)
- Error : Several Basic error types, implemented for your convenience
- Method : An object that generates generic message sender methods
- MethodResolve : A way to extract methods (raw, or with refs) from objects, and use them as code blocks

The Data category

- Array : mostly deprecated stuff. subject to removal
- IO : /StringIO -> Fake strings as files.
- Iterator : Deprecated. Use `Generator` instead

- List : Implementation of LinkedLists (for whatever reason)
- Map : simplify your use of Maps
- Range : Non-lazy list generators
- Ratio : Fractions as ratios
- Set
-
  - /HashSet : Implementation of HashSets
  - /BloomFilter : Implementation of BloomFilters
- String : Nothing.
- SwitchCase : Generate a map of alternatives and execute one of them.

The Functional category

- Alias : common functions for common messages (e.g. `+` as a polymorphic function)
- Applicative
-
  - /Maybe -> The Maybe monad as an applicative
- Category
-
  - Array -> Arrays as composable objects
  - Block -> Functions as composable objects
- Monad : semi context-aware entities that wrap a state
- TypeClass : The definition of all the above things, and a few common functions
- PatternMatch : Adds a Object::'match:', and a helper `match` function
-
  - More of this in its own page

The Executable category

- cpm : A package manager. read its help with `citron -m cpm`

The Foreign/C category

This category is dedicated to interfacing foreign functions, and has a complementary plugin (libctrctypes.so)

- Alloc : Allocate raw memory with specific sizes
- C_Types : common import with names for common C types
- errno : module to handle C errno
- NamedStruct : Structs with named members
- PackedArray : Contiguous Array with uniform native types

- Signal : Handle or trade signals with other programs

TODO: Finish this page

Meta Functions

If you've survived this far, congrats; the ride is going to get even crazier from here.

Meta functions, Basically, allow a function to modify the internal representation of an expression, and optionally return a replacement for it.

They are in a way modeled and built to remedy the need for macros, but ended up quite a bit more powerful (as they are somewhat context aware)

Their definition is absolutely the same as every other function, except they do…meta things.

The basic building blocks are the several compiler intrinsics listed below:

1. `$(expr)`
2. `$!(expr)`
3. `$[expr, expr, ...]`
4. `$'(expr)` or `$'expr`
5. ``$`(expr)`` or ``$`expr``

A brief explanation of their behaviour is as follows:

1. is replaced by the internal (AST) representation of the expression
2. splices the meta-expression back into the code; it is the reverse action of 1
3. is replaced by a tuple containing the AST representation of the contained expressions
4. quotes an expression (bare references are converted to symbols)
5. escapes a quoted expression and embeds a value into it

A simple example

```
var if-then-else is {:ast
    ^$!(ast head) either: { #if the first expression evaluates to a truthy value
        ^$!(ast at: 1). #evaluate the second expression
    } or: {
        ^$!(ast at: 2). #otherwise, evaluate the third expression
    }.
}.
var a is Nil.

if-then-else applyTo:
    $[ # To get a tuple of expressions
        1 = 2, # the condition
        a is 'No Way!', # the if-true branch
        a is 'That\'s right'. # the if-false branch
    ].
# This whole expression will evaluate to the result of "a is 'That\'s right'", which is the
```

Of course, this feels clunky, and looks weird, but there are more ways to deal with metafunctions, in a more elegant way.

(First, see Parser Pragmas)

Through the use of the `declare` pragma, it is possible to create a function that does not evaluate its arguments before being invoked, but rather explicitly after.

Example:

```
#:declare lazyev then

# These sorts of functions are always binary
var letIn is {\:_x:_y
    $!(_x). # Evaluate the first argument, and discard its value
    $!(_y). # Evaluate the second argument, and return its value
}.

(var a is 123) `letIn` (a + 64). # => 187
# which is equivalent to the following expression
letIn applyAll: [$(var a is 123), $(a + 64)]. # => 187

#:declare lazyev if-then

# let's write if-then in a more elegant format
var if-then is {\:if:then
    $!(if) ifTrue: {
        ^$!(then).
    }.
}.

var x is 123.

(x = 123) `if-then` (Pen writeln: 'x really is $$x'). # be careful not to shadow `x' in the
# -> x really is 123
```

Let's take a look at how PatternMatch's `match` function works

```
{:self:ast

var done is True.
var ret is Nil.
var cblk is thisBlock.
ast each_v: {:__PMATCH_V0 # For each sub-tuple in the main tuple
  __PMATCH_V0 is $!(__PMATCH_V0). #Parse the tuple
  done is True.
  ret is {
    self unpack: $!(__PMATCH_V0 head). # Try to unpack the object
    $!(__PMATCH_V0 tail init) all: {\:__PMATCH_V1 # Then assuming the unpacking worked, che
      Reflect runHere: {^$!(__PMATCH_V1).} forObject: self arguments: [].
    }, ifFalse: {
      cblk error: 'guard fail'. # If the guard returned false, let this pmatch fail
    }.
    ^Reflect runHere: {^$!(__PMATCH_V0 last).} forObject: self arguments: []. # Assuming eve
  } catch: {:e done is False. }, run. # if anything failed, move on through the tuple
  done break. # otherwise, stop
}.
done ifFalse: { cblk error: 'non-exhaustive pattern match'. }. # if after going through the
^ret. # return the value

}.
```

`@comptime` - when you definitely want it to be evaluated at comptime

meta functions are nice and all, but there are times that you want to be *sure* on the code being evaluated at comptime

a (perhaps deprived) example:

```
20 times: {:i
    Pen writeln: i factorial.
}.
```

You definitely don't want this sort of code if you can avoid it.

what you can do here is to create a lookup table:

```
# make a lookup table
@comptime[discard] # the actual value of this doesn't matter
var lookup-factorial is
    $($`(Array new: 20, fill: 20 with: \:x x factorial)).

20 times: {:i
    Pen writeln: (@comptime lookup-factorial) @ i.
}.
```

Now all that happens at runtime is a simple array lookup. although this will break should `i` be out of the bounds.

```
1
1
```

```
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
```

Parser Pragmas

The parser supports pragmas in the form of `#:pragma_name arguments` that modify its behaviour, and these are completely irrelevant to the actual control flow of the program.

Basic Pragmas

- `#:oneLineExpressions`
  - Promise to be a good child, and end your expressions in one line, and the parser will place the ending dots for you
- `#:regexLineCheck`
  - will force the use of regular expressions for newline detection (deprecated, really)

Not-so-basic Pragmas

- `#:callShorthand LEFT RIGHT`
  - Will replace the call shorthand `expr[]` with `expr LEFT RIGHT`
  - the only accepted tokens are any mix of `{ [ ( ) ] }`
- `#:declare <infixl|infixr|lazyev> [prec] <reference>`
  - Declares the fixity or laziness of a reference (only used when it is invoked through ``A `ref` B``
- `#:language <comma-separated list of extensions>`
  The currently valid extensions are:

- `XFrozen`
    - Adds a variable modifier `frozen`, which causes the LHS of an assignment to be evaluated *only once.* it is only valid in assignments.
- `XPureLambda`
    - Memoizes lambda expressions that are detected to be pure (side-effect free)
- `XNakedAsmBlocks`
    - Allows insertion of assembly code inside a block specially denoted such [more on this in Inline Assembly Blocks

Scratchpad

The citron evaluator also ships with a neat little scratchpad, you can launch it like so:

```
$ citron -m scratchpad
```

Its basic functionality is to provide a neat GUI for executing and inspecting citron code.

Unlike the evaluator, it is made to allow editing and executing several lines of code (autocomplete coming soon™)

Basic usage

You simply type in some code (it even has live syntax highlighting!) and to execute a (selected) chunk of code, you press F5.

And the resulting value is inserted in the next line.

Object inspection

The scratchpad allows you to inspect an object's values and methods (read only, no modification)

To do so, simply send an `inspect` message to any object:

```
Scratchpad inspect
```

and a new window with the properties and methods of that object will pop up

The treeview on the left first lists the inheritance chain of the object (if any), then a bunch of '—-'s, and then the properties.

Double clicking a property will show its value in the textview on the bottom, and middle-mouse-clicking on a property will `inspect` it.

The method list on the right is only browsable at the moment.

Inline assembly blocks

Just as every other decent high-level language (not), Citron supports inline assembly blocks (and it can treat them as pretty much normal blocks)

To use this feature, you must enable the `XNakedAsmBlock` parser pragma.

Then, the syntax is as follows:

`'{' 'asm' [dialect] {argument-types-or-names} (register-constraints) {assembly code} '}'`

Where `dialect` can be any of `intel`, `att`, `at&t`, or nothing at all (defaults to at&t);

The argument types are typical citron arguments where the name optionally denotes the type (in such cases where the name is not recognised, the type is assumed to be `double`)

Recognised argument types:

- `int`: a normal 64 bit integer
- `dbl`: a double precision real value
- `str`: a pointer to an array of characters

the return type is *always* a 64bit integer (or whatever the default size of the native output register is)

The register constraints are modelled exactly after LLVM's inline assembly expressions, so it is worth reading up on their documentation; however, here is the gist:

- constraints are separated with commas
- Each is a code describing one output, input, or clobbered register
  - output codes start with `=`
    - They can optionally be prefixed with `&` to be marked early clobbered
  - input codes have no prefix
  - clobber codes start with `~`
- The code itself is any one of the following formats
  - one constraint character
    - The most common are `r` (register) and `m` (memory)
  - '{' register-name '}'
    - example: `{rax}`
  - constraint constraint
    - to allow for alternatives
    - example: `{rax}m`: either allocate `%rax`, or fallback to memory
  - There must be exactly as many input constraints as there are arguments; any deviation will simply cause either a segfault or a comptime error

The assembly code can then refer to the constrainted registers/memory locs by their index in the constraint list: `$<index>`

For example, if the constraint list is `(={rax},m,r)`, to address the first input (the memory), `$1` can be used.

Note that because of this templating system, number literals in at&t syntax must be prefixed with two `$`, not one (`$$0x10`)

Example

```
#:language XNakedAsmBlock

var native-fn is {asm intel :int (={rax},r)
    add $1, $1 # double the input
    mov $0, $1 # put it in the output
}.

# Now you can treat `native-fn' as any boring old code block
native-fn applyTo: 4. # => 8
```