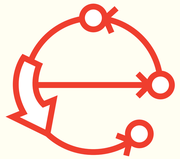
Monte



Monte is designed for agoric systems, promotes secure distributed computation, and is focused on readability and clarity.

Monte is a nascent dynamic programming language reminiscent of Python and E. It is based upon The Principle of Least Authority (POLA), which governs interactions between objects, and a capability-based object model, which grants certain essential safety guarantees to all objects.

Monte is designed for agoric systems, promotes secure distributed computation, and is focused on readability and clarity.

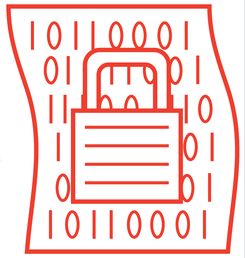
Monte's design incorporates:

Guard-based Type System
Values are typed according to guards, which are objects that describe the behavior of values.

Readable & Auditable Syntax
Python-inspired syntax includes customizeable pattern matching and object literals, while semantic features like static lexical scoping and left-to-right evaluation aid code review.

Builtin Concurrency
First-class promises and syntax for eventual message passing facilitate a natural and simple set of idioms for highly concurrent systems.

Cryptographic Sealing



This example features bindings to `libsodium` to create a relatively compact single-file encryption tool, based on `libsodium` boxes.

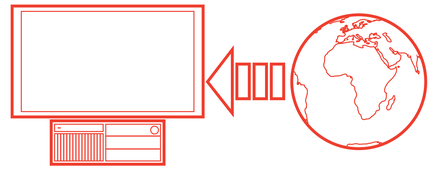
The `<-` syntax turns calls into eventual sends, which will result in deferred computation. Like when expressions, eventual sends are executed sometime after the computation that prepares them.

`switch` expressions perform pattern-matching against a sequence of patterns. Here, the specimen `argv`, representing command-line arguments, is deconstructed into an operation and positional parameters.

```
exports (main)

def main(argv, => currentRuntime, => makeFileResource) as DeepFrozen:
    def getFile(path :Str):
        return makeFileResource(path)<-getContents()

    def setFile(path :Str, bs :Bytes):
        return makeFileResource(path)<-setContents(bs)

    def keyMaker := currentRuntime.getCrypt().keyMaker()
    return switch (argv):
        match [=="makeKey", keyPath]:
            def [_, secretKey] := keyMaker()
            def secretBytes :Bytes := secretKey.asBytes()
            when (setFile(keyPath, secretBytes)) -> {0}
        match [=="seal", keyPath, plainPath, cipherPath, noncePath]:
            def secretBytes := getFile(keyPath)
            def plainBytes := getFile(plainPath)
            when (secretBytes, plainBytes) ->
                def secretKey := keyMaker.fromSecretBytes(secretBytes)
                def pair := secretKey.pairWith(secretKey.publicKey())
                def [cipherBytes, nonceBytes] := pair.seal(plainBytes)
                when (setFile(cipherPath, cipherBytes),
                      setFile(noncePath, nonceBytes)) -> {0}
        match [=="unseal", keyPath, cipherPath, noncePath, plainPath]:
            def secretBytes := getFile(keyPath)
            def cipherBytes := getFile(cipherPath)
            def nonceBytes := getFile(noncePath)
            when (secretBytes, cipherBytes, nonceBytes) ->
                def secretKey := keyMaker.fromSecretBytes(secretBytes)
                def pair := secretKey.pairWith(secretKey.publicKey())
                def plainBytes := pair.unseal(cipherBytes, nonceBytes)
                when (setFile(plainPath, plainBytes)) -> {0}
```

Principle of Least Authority
"Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job."
~ Jerry Saltzer

HTTP Fetch



This example shows off both Monte's disdain for ambient authority and its strong builtin concurrency primitives. The Python `requests` library makes this example look verbose in comparison, but the verbosity is merely a product of the explicit authority which Monte requires for unsafe operations.

Our entrypoint `main` requests two unsafe capabilities via keyword argument: `getAddrInfo`, which corresponds to `getaddrinfo()` at the OS layer, and `makeTCP4ClientEndpoint`, a maker of endpoints which can create IPv4 TCP clients. Outside of these capabilities, this entrypoint cannot misbehave in certain ways; it cannot access the filesystem, spawn subprocesses, nor even access the runtime utilities of the current process.

We use when expressions to control concurrency. The `->` arrow indicates a block of actions that will happen after the promise at the head of the `when` expression resolves. In this example, we have two promises that we wait for. First, we wait for `getAddrInfo` to perform a DNS lookup. (Or other equivalent blocking long-running I/O.) Then, we use the results of that lookup to perform an HTTP request.

```
import "lib/codec/utf8" =~ [=> UTF8 :DeepFrozen]
import "lib/gai" =~ [=> makeGAI :DeepFrozen]
import "http/client" =~ [=> makeRequest :DeepFrozen]
exports (main)

def main(argv, => getAddrInfo, => makeTCP4ClientEndpoint) as DeepFrozen:
    def addrs := getAddrInfo(b`example.com`, b``)
    return when (addrs) ->
        def gai := makeGAI(addrs)
        def [addr] + _ := gai.TCP4()
        def response := makeRequest(makeTCP4ClientEndpoint, addr.getAddress(),
                                    "/").get()
        when (response) ->
            traceln(response)
            traceln(UTF8.decode(response.body(), null))
            0
```

Object-based Capabilities
"Only connectivity begets connectivity — all access must derive from previous access."

~ Mark Miller

Additional boons of Monte include:

- Quasiliteral syntax, including quasiliteral patterns and an extensible interface.
- Composition-based inheritance.
- Immutable and persistent core data structures.
- Parameterized modules.
- Audition-based proving of customized object behaviors.

To find out more, visit
https://monte.readthedocs.org/
http://www.monte-language.org/
#monte on irc.freenode.net