Monte Documentation

Release 0.1

Monte Project

November 13, 2017

Contents

Monte is a programming language inspired by the E and Python programming languages. Monte aims to be:

- A reliable scaffold for secure distributed computing
- An example of capability-safe programming language design
- A model for misuse-resistant programming

CHAPTER 1

Introduction

Why Monte?

Don't we have enough languages already? This is a fair question. Here we'll explain why we created Monte and what's interesting about it.

Because Security Matters

Secure distributed computing should not be hard. Computers are getting faster, smaller, more connected, and more capable, but when it comes to security, everything is broken. We propose to reconsider the identity-based access control approach dominant in today's dominant languages and frameworks [1]. Monte takes the object-capability paradigm of E [2] and expands on the approach, delivering a powerful and expressive language.

Monte, like E before it, has dramatic advantages for secure distributed systems:

- Capability-based security enables the concise composition of powerful patterns of interoperation–patterns that enable extensive cooperation even in the presence of severely limited trust.
- Monte promises benefit from a *promise-pipelining* architecture which ensures that *most deadlocks cannot occur*. *[0]
- Monte offers cryptographic services directly to its users, easing the use of good cryptographic primitives.

Because Readability Matters

The origin of Monte's name

The Monte language has its roots in the E and Python languages. We took "Monty" from "Monty Python", and put an "e" in there. Thus, "Monte".

Monte wraps its strengths in a Python-like syntax to make it quickly comfortable for a large number of software engineers.

[1] Disciplined use of existing languages such as Java and JavaScript can be used to build object capability systems, but the standard practices and libraries are not compatible with this discipline.

[2] Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)

See also a history of E's ideas.

[0] As with all sufficiently complex concurrency systems, deadlock is possible. That said, it has not been observed outside of specially-constructed pathological object graphs.

Monte is a pure object-based language in the Smalltalk tradition, making it easy to write modular, readable, maintainable software using the strategies familiar from Python, JavaScript, Ruby, Java, and other object-based languages. All values are objects and all computation is done by sending messages to objects. It has the kind of powerful string handling that will be recognized and seized upon by the Perl hacker.

Because Stability Matters

Monte is dynamically typed [3], like Smalltalk, rather than statically typed, like Java. Users of Perl and Python will immediately recognize this is an advantage; Java and C++ programmers may not be so sure. Fortunately, Monte inherits two forms of contract-based programming from E: guards and *interfaces*.

Monte is dynamic in three ways:

Dynamic Typing The type of a variable might not be known until runtime, and "types are open".

Dynamic Binding It is possible to pass a message to an object that will never able to handle that message. This provides a late-binding sort of polymorphism.

Dynamic Compiling Monte can compile and run Monte code at runtime, as part of its core runtime.

While "arbitrary code execution" is a notorious security vulnerability, Monte enables the fearless yet powerful use of multi-party limited-trust mobile code.

Object Capability Discipline

A *capability* is a reference to an object and represents authority to invoke methods on the object. The key to supporting dynamic code execution without vulnerability is *object capability discipline*, which consists of:

Memory safety and encapsulation There is no way to get a reference to an object except by creating one or being given one at creation or via a message; no casting integers to pointers, for example.

From outside an object, there is no way to access the internal state of the object without the object's consent (where consent is expressed by responding to messages).

Primitive effects only via references The only way an object can affect the world outside itself is via references to other objects. All primitives for interacting with the external world are embodied by primitive objects and anything globally accessible is immutable data. There is no `open(filename)` function in the global namespace, nor can such a function be imported. The runtime passes all such objects to an entrypoint, which then explicitly delegates to other objects.

We'll demonstrate how this leads to natural expression of the *Principle of Least Power* briefly in *A Taste of Monte: Hello Web* and in more detail in *Secure Distributed Computing*.

Why not Monte?

Monte assumes automatic memory management; the current reference implementation uses the PyPy garbage collector, and any other implementation will have to choose a similar scheme. As such, it is not a good language for low level machine manipulation. So do not try to use Monte for writing device drivers.

Monte's performance is currently quite unfavorable compared to raw C, and additionally, Monte's target niches are largely occupied by other dynamic languages with JIT-compiler-based runtimes, so it is not a design goal to compete with C or other memory-unsafe languages.

[3] in formal type theory, Monte is *unityped*.

Note: While Monte's usable and most architectural issues are resolved, it is still undergoing rapid development. See *Roadmap: Montefesto* for details.

Getting Started

Quick Start Docker Image

If you have Docker installed, the quickest way to get to an interactive prompt to run some Monte code is `docker run -it montelang/repl`. This container provides the essentials needed for most examples in this documentation.

A container with a shell and the full set of Monte development tools is available on Docker Hub as well, `montelang/monte-dev`.

Installation

If you don't want to use Docker, the other supported environment requires the packaging/build tool Nix. It can be installed on Linux and OSX from their installer script:

```
curl https://nixos.org/nix/install | sh
```

Alternately, you can install it manually from tarball, deb or rpm.

Once you have Nix set up, you can use prebuilt packages courtesy of Matador Cloud:

```
nix-channel --add https://hydra.matador.cloud/project/monte/channel/latest monte
```

Once that's set up, you can always fetch the latest build by running:

```
nix-channel --update
nix-env -i monte
```

Alternately, you can use Nix to build from source. After fetching the Monte runtime from GitHub, you can start the build from the root directory of the source tree:

```
nix-env -f . -iA monte
```

Interacting with the Monte REPL

Monte has a traditional "Read - Evaluate - Print Loop", or REPL, for exploration. Invoke it as `monte repl`. For example:

```
>>> 1 + 1
2

>>> "abc".size()
3
```

Getting Help about an Object

Monte strives to provide useful error messages and self-documenting objects:

```
> help(Ref)
Result: Object type: RefOps
Ref management and utilities.
Method: broken/1
Method: isBroken/1
Method: isDeepFrozen/1
...
```

Editor Syntax Highlighting

Emacs and Flycheck

The monte-emacs repository provides emacs syntax highlighting on-the-fly syntax checking with flycheck.

Vim

The monte-vim repository provides vim syntax highlighting, and linter integration is available via a private Syntastic repository.

Atom

Use Atom to install the package language-monte.

Support and Feedback

We welcome feedback:

- issues in monte pypy vm implementation (typhon)
- issues in monte documentation

Or come say hi on IRC, in *#monte* on *irc.freenode.net*!

Acknowledgements

Monte design and documentation borrow heavily from E in a Walnut by Marc Stiegler and The E Language and ELib by Mark Miller.

Notes

CHAPTER 2

A Taste of Monte: Hello Web

Let's see what a simple web server looks like in Monte:

```
import "http/server" =~ [=> makeHTTPEndpoint :DeepFrozen]
exports (main)

def helloWeb(request) as DeepFrozen:
    "Build a simple HTML response."

    return [200, ["Content-Type" => "text/html"], b`<p>Hello!</p>`]

def main(argv, => makeTCP4ServerEndpoint) :Int as DeepFrozen:
    "Obtain a port and create an HTTP server on that port."

    def portNum :Int := _makeInt(argv.last())
    def ep := makeHTTPEndpoint(makeTCP4ServerEndpoint(portNum))
    traceln(`serving on port $portNum`)
    ep.listen(helloWeb)
    return 0
```

The `makeHTTPEndpoint` import reads much like Python's `from http.server import makeHTTPEndpoint`, though the mechanics of a module declaration in monte are a bit different: it uses pattern matching to bind names to objects imported from modules.

DeepFrozen Module Exports

One of the constraints of *object capability discipline* is that there is no global mutable state; so exported objects must be `DeepFrozen`, i.e. transitively immutable. Since `main` calls `helloWeb`, `helloWeb` must be `DeepFrozen` as well. We'll discuss this and other static properties of Monte code in the *Auditors* section.

We declare that this module `exports` its `main` function, as is conventional for executable programs.

Todo

Document how to compile and run such a script.

Blocks in Monte are typically written with indentation, like Python, though blocks in general may be written with curly-braces as well.

Note: Tabs are a syntax error in Monte.

Expressions

The def-expr for defining the `helloWeb` function is similar to Python's syntax for defining functions.

Expression Languages

Unlike Python and C, which use a mix of statements and expressions, Monte is an expression language, like Scheme. So `def body := ...` is an expression with a value, just like string literals and method calls.

The expression inside the call to `traceln(...)` does string interpolation, similar to Perl, Ruby, and bash. It is a *quasiliteral* expression:

```
> def portNum := 8080
> `serving on port $portNum`
"serving on port 8080"
```

Another quasiliteral is `b` Hello! ``, which denotes a `Bytes` object rather than a character string.

Objects and Message Passing

Monte is a pure object language, which means that all values in Monte are objects. All operations on objects are done by passing messages. This includes ordinary method calls like `argv.last()` as well as *function calls* such as `traceln(portNum)` and even syntax for constructing lists like `[200, [], body]` and maps like `["C" => "t"]`.

Cooperation Without Vulerability

Suppose our server takes an arbitrary expression from the web client and evaluates it:

```
import "http/server" =~ [=> makeHTTPEndpoint :DeepFrozen]
import "http/tag" =~ [=> tag :DeepFrozen]
import "formData" =~ [=> fieldMap :DeepFrozen]
exports (main)

object calculator as DeepFrozen:
    to run(request):
        return switch (request.getVerb()):
            match =="GET":
                calculator.get(request)
            match =="POST":
                calculator.post(request)

    to get(request):
        def body := b`
        <form method="POST">
          <label>Arbitrary code to execute:<input name="code" /></label>
        </form>
        `
        return [200, ["Content-Type" => "text/html"], body]

    to post(request):
        def code := fieldMap(request.getBody())["code"]
```

```
        def result := eval(code, safeScope)
        # NB: The `tag` object does automatic HTML escaping. No extra effort
        # is required to prevent XSS. ~ C.
        def html := tag.pre(M.toString(result))
        return [200, ["Content-Type" => "text/plain"], b`$html`]

def main(argv, => makeTCP4ServerEndpoint) :Int as DeepFrozen:
    def portNum := _makeInt(argv.last())
    def ep := makeHTTPEndpoint(makeTCP4ServerEndpoint(portNum))
    traceln(`serving $calculator on port $portNum`)
    ep.listen(calculator)
    return 0
```

With conventional languages and frameworks, this would be injection, #1 on the list of top 10 web application security flaws:

> Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.

But using object capability discipline, untrusted code has only the authority that we explicitly give it. This rich form of cooperation comes with dramatically less vulnerability [1]. The environment in this example is safeScope, which is the same environment modules are evaluated in – it provides basic runtime services such as constructors for lists, maps, and other structures, but no "powerful" objects. In particular, `makeTCP4ServerEndpoint` is not in scope when the remote code is executed, so the code cannot use it to access the network. Neither does the code have any access to read from nor write to files, clobber global state, nor launch missiles.

Notes

[1] We implicitly grant authority to compute indefinitely. Object capability discipline does not address denial of service. Monte's vats include a conventional mechanism to put a finite limit on computation.

CHAPTER 3

Practical Security: The Mafia game

Let's look a bit deeper at Monte, working up to an implementation of the Mafia party game.

Objects

Monte has a simpler approach to object composition and inheritance than many other object-based and object-oriented languages.

A Singleton Object

We will start our exploration of objects with a simple singleton object. Methods can be attached to objects with the `to` keyword:

```
>>> object origin:
...     to getX():
...         return 0
...     to getY():
...         return 0
... # Now invoke the methods
... origin.getY()
0
```

Unlike Java or Python, Monte objects are not constructed from classes. Unlike JavaScript, Monte objects are not constructed from prototypes. As a result, it might not be obvious at first how to build multiple objects which are similar in behavior.

Functions are objects too

Functions are simply objects with a `run` method. There is no difference between this function:

```
>>> def square(x):
...     return x * x
... square.run(4)
16
```

... and this object:

```
>>> object square:
...     to run(x):
...         return x * x
... square(4)
16
```

Warning: Python programmers beware, methods are not functions. Methods are just the public hooks to the object that receive messages; functions are standalone objects.

Todo

document docstrings

Todo

document named args, defaults

Object constructors and encapsulation

Monte has a very simple idiom for class-like constructs:

```
>>> def makeCounter(var value :Int):
...     return object counter:
...         to increment() :Int:
...             return value += 1
...         to makeOffsetCounter(delta :Int):
...             return makeCounter(value + delta)
...
... def c1 := makeCounter(1)
... c1.increment()
... def c2 := c1.makeOffsetCounter(10)
... c1.increment()
... c2.increment()
... [c1.increment(), c2.increment()]
[4, 14]
```

And that's it. No declarations of object contents or special references to `this` or `self`.

Assignment Expressions

Monte is an expression language. The expression `value += 1` returns the resulting sum. That's why `return value += 1` works.

Inside the function `makeCounter`, we simply define an object called `counter` and return it. Each time we call `makeCounter()`, we get a new counter object. As demonstrated by the `makeOffsetCounter` method, the function (`makeCounter`) can be referenced from within its own body. (Similarly, our counter object could refer to itself in any of its methods as `counter`.)

The lack of a `this` or `self` keyword may be surprising. But this straightforward use of lexical scoping saves us the often tedious business in python or Java of copying the arguments from the parameter list into instance variables: `value` is already an instance variable.

The `value` passed into the function is not an ephemeral parameter that goes out of existence when the function exits. Rather, it is a true variable, and it persists as long as any of the objects that uses it persist. Since the counter uses this variable, `value` will exist as long as the counter exists.

Augmented Assignment

Just as you would read `x += 1` short-hand for `x := x + 1`, read the augmented assignment `players without= (victim)` as `players := players.without(victim)`.

A natural result is the **complete encapsulation** required for *object capability discipline*: `value` is not visible outside of `makeCounter()`; this means that *no other object can directly observe nor modify it*. Monte objects have no public attributes or fields or even a notion of public and private. Instead, all names are private: if a name is not visible (i.e. in scope), there is no way to use it.

We refer to an object-making function such as `makeCounter` as a "Maker". As a more serious example, let's make a sketch of our game:

```
>>> def makeMafia(var players :Set):
...     def mafiosoCount :Int := players.size() // 3
...     var mafiosos :Set := players.slice(0, mafiosoCount)
...     var innocents :Set := players.slice(mafiosoCount)
...
...     return object mafia:
...         to getWinner():
...             if (mafiosos.size() == 0):
...                 return "village"
...             if (mafiosos.size() >= innocents.size()):
...                 return "mafia"
...             return null
...
...         to lynch(victim):
...             players without= (victim)
...             mafiosos without= (victim)
...             innocents without= (victim)
...
... def game1 := makeMafia(["Alice", "Bob", "Charlie"].asSet())
... game1.lynch("Bob")
... game1.lynch("Charlie")
... game1.getWinner()
"mafia"
```

Traditional Datatypes and Operators

Monte includes basic data types such as `Int`, `Double`, `Str`, `Char`, and `Bool`. All integer arithmetic is unlimited precision, like in Python.

The operators `+`, `-`, and `*` have their traditional meanings for `Int` and `Double`. The normal division operator `/` always gives you a `Double` result. The floor divide operator `//` always gives you an `Int`, truncated towards negative infinity. So:

```
>>> -3.5 // 1
-4
```

Comments

Monte uses the same `# ...` syntax for comments as Python and bash.

Strings are enclosed in double quotes. Characters are enclosed in single quotes.

The function `traceln` sends diagnostic output to the console. The `if` and `while` constructs look much like their Python equivalents, as do lists such as `[4, 14]`.

Operator precedence is generally the same as in Java, Python, or C. In a few cases, Monte will throw a syntax error and require the use of parentheses.

With that, let's set aside our game sketch and look at a more complete rendition, `mafia.mt`:

```
# An implementation of the Mafia party game state machine.

import "lib/enum" =~ [=> makeEnum]
exports (makeMafia, DAY, NIGHT)

def [MafiaState :DeepFrozen,
     DAY :DeepFrozen,
     NIGHT :DeepFrozen] := makeEnum(["day", "night"])


def makeMafia(var players :Set, rng) as DeepFrozen:
    # Intial mafioso count.
    def mafiosoCount :Int := players.size() // 3

    def sample(population :List, k :(Int <= population.size())) :List:
        def n := population.size()
        def ixs := [].diverge()
        while (ixs.size() < k):
            if (!ixs.contains(def ix := rng.nextInt(n))):
                ixs.push(ix)
        return [for ix in (ixs) population[ix]]

    var mafiosos :Set := sample(players.asList(), mafiosoCount).asSet()
    var innocents :Set := players - mafiosos

    var state :MafiaState := NIGHT
    var day := 0
    var votes :Map := [].asMap()

    object mafia:
        to _printOn(out) :Void:
            def mafiaSize :Int := mafiosos.size()
            def playerSize :Int := players.size()
            out.print(`<Mafia: $playerSize players, `)
            def winner := mafia.getWinner()
            if (winner == null):
                out.print(`$state $day>`)
            else:
                out.print(`winner $winner>`)

        to getState() :MafiaState:
            return state

        to getQuorum() :Int:
            return switch (state) {
```

```
                match ==DAY { (mafiosos.size() + innocents.size() + 1) // 2}
                match ==NIGHT {mafiosos.size()}
            }

        to getMafiaCount() :Int:
            return mafiosoCount

        to getWinner():
            if (mafiosos.size() == 0):
                return "village"
            if (mafiosos.size() >= innocents.size()):
                return "mafia"
            return null

        to advance() :Str:
            if (mafia.getWinner() =~ outcome ? (outcome != null)):
                return outcome
            if ([state, day] == [NIGHT, 0]) {
                state := DAY
                day += 1
                return "It's morning on the first day."
            }
            if (mafia.lynch() =~ note ? (note != null)):
                state := switch (state) {
                    match ==DAY {NIGHT}
                    match ==NIGHT { day += 1; DAY}
                }
                votes := [].asMap()
                return note
            return `${votes.size()} votes cast.`


        to vote(player ? (players.contains(player)),
                choice ? (players.contains(choice))) :Void:
            switch (state):
                match ==DAY:
                    votes with= (player, choice)
                match ==NIGHT:
                    if (mafiosos.contains(player)):
                        votes with= (player, choice)

        to lynch() :NullOk[Str]:
            def quorum :Int := mafia.getQuorum()
            def counter := [].asMap().diverge()
            for _ => v in (votes):
                if (counter.contains(v)):
                    counter[v] += 1
                else:
                    counter[v] := 1
            traceln(`Counted votes as $counter`)

            escape ej:
                def [victim] exit ej := [for k => v in (counter) ? (v >= quorum) k]
                def count := counter[victim]
                def side := mafiosos.contains(victim).pick(
                    "mafioso", "innocent")
                players without= (victim)
                mafiosos without= (victim)
```

```
                innocents without= (victim)
                return `With $count votes, $side $victim was killed.`
            catch _:
                return null

    return ["game" => mafia, "mafiosos" => mafiosos]
```

Unit Testing

This module also uses Monte's unit test facilities to capture a simulated game:

```
import "unittest" =~ [=> unittest]
import "lib/entropy/entropy" =~ [=> makeEntropy :DeepFrozen]
import "lib/entropy/pcg" =~ [=> makePCG :DeepFrozen]


def sim1(assert):
    def names := ["Alice", "Bob", "Charlie",
                  "Doris", "Eileen", "Frank",
                  "Gary"]
    def rng := makeEntropy(makePCG(731, 0))
    def randName := fn { names[rng.nextInt(names.size())] }
    def [=> game, =>mafiosos] := makeMafia(names.asSet(), rng)
    assert.equal(`$game`, "<Mafia: 7 players, night 0>")
    assert.equal(mafiosos, ["Eileen", "Frank"].asSet())

    def steps := [game.advance()].diverge()
    while (game.getWinner() == null):
        # Rather than keep track of who is still in the game,
        # just catch the guard failure.
        try:
            game.vote(randName(), randName())
        catch _:
            continue
        def step := game.advance()
        if (step !~ `@n votes cast.`):
            steps.push(step)
            steps.push(`$game`)

    assert.equal(steps.snapshot(),
                 ["It's morning on the first day.",
                  "With 4 votes, innocent Alice was killed.",
                  "<Mafia: 6 players, night 1>",
                  "With 2 votes, mafioso Eileen was killed.",
                  "<Mafia: 5 players, day 2>",
                  "With 3 votes, mafioso Frank was killed.",
                  "<Mafia: 4 players, winner village>"])
unittest([sim1])
```

We still cannot import access to a true source of entropy; `makePCG` constructs a pseudo-random number generator given an initial seed, and `makeEntropy` makes an object that takes the resulting sequence of bytes and packages them up conveniently as integers etc. In *Secure Distributed Computing*, we will develop the part of the game that provides a truly random seed. But for unit testing, the seed is an arbitrarily chosen constant.

Additional flow of control

Other traditional structures include:

- `try{...} catch errorVariable {...} finally {...}`
- `throw(ExceptionExpressionThatCanBeAString)`
- `break`, `continue`
- `switch (expression) {match pattern1 {...} match pattern2 {...} ... match _ {defaultAction}}`

String Interpolation with quasi-literals

Monte's *quasi-literals* enable the easy processing of complex strings as described in detail later; `out.print(`currently $state>`)` is a simple example wherein the back-ticks denote a quasi-literal, and the dollar sign denotes a variable whose value is to be embedded in the string.

Dynamic "type checking" with guards

Monte guards perform many of the functions usually thought of as type checking, though they are so flexible that they also work as concise assertions. Guards can be placed on variables (such as `mafiososCount :Int`), parameters (such as `players :Set`), and return values (such as `getState() :MafiaState`).

Guards are not checked during compilation. They are checked during execution and will throw exceptions if the value cannot be coerced to pass the guard.

Optimizing Monte Compilers

Monte does not specify a compilation model. Some guards can be optimized away by intelligent Monte compilers, and linters may warn about statically-detectable guard failures.

Monte features strong types; monte values resist automatic coercion. As an example of strong typing in Monte, consider the following statement:

```
def x := 42 + true
```

This statement will result in an error, because `true` is a boolean value and cannot be automatically transformed into an integer, float, or other value which integers will accept for addition.

We can also build guards at runtime. The call to `makeEnum` returns a list where the first item is a new guard and the remaining items are distinct new objects that pass the guard. No other objects pass the guard.

Todo

show: Guards play a key role in protecting security properties.

Final, Var, and DeepFrozen

Bindings in Monte are immutable by default.

The *DeepFrozen guard* ensures that an object and everything it refers to are immutable. The `def makeMafia(...) as DeepFrozen` expression results in this sort of binding as well as patterns such as `DAY :DeepFrozen`.

Using a `var` pattern in a definition (such as `mafiosos`) or parameter (such as `players`) lets you assign to that variable again later.

There are no (mutable) global variables, however. We cannot import a random number generator. Rather, the random number generator argument `rng` is passed to the `makeMafia` maker function explicitly.

Assignment and Equality

Assignment uses the `:=` operator, as in Pascal. The single equal sign = is never legal in Monte; use `:=` for assignment and == for testing equality.

== and != are the boolean tests for sameness. For any pair of refs x and y, "x == y" will tell whether these refs designate the same object. The sameness test is monotonic, meaning that the answer it returns will not change for any given pair of objects. Chars, booleans, integers, and floating point numbers are all compared by their contents, as are Strings, ConstLists, and ConstMaps. Other objects only compare same with themselves, unless their definition declares them:ref:*Transparent<selfless>*, which lets them expose their contents and have them compared for sameness.

Data Structures for Game Play

Monte has `Set`, `List`, and `Map` data structures that let us express the rules of the game concisely.

A game of mafia has some finite number of players. They don't come in any particular order, though, so we write `var players :Set` to ensure that `players` is always bound to a `Set`, though it may be assigned to different sets at different times.

We use `.size()` to get the number of players, and once we get the `mafiosos` subset (using a `sample` function), the set of `innocents` is the difference of `players - mafiosos`.

We initialize `votes` to the empty `Map`, written `[].asMap()` and add to it using `votes with= (player, choice)`.

To `lynch`, we use `counter` as a map from player to votes cast against that player. We initialize it to an empty mutable map with `[].asMap().diverge()` and then iterate over the votes with `for _ => v in votes:`.

Functional Features (WIP)

Monte has support for the various language features required for programming in the so-called "functional" style. Monte supports closing over values (by reference and by binding), and Monte also supports creating new function objects at runtime. This combination of features enables functional programming patterns.

Monte also has several features similar to those found in languages in the Lisp and ML families which are often conflated with the functional style, like strict lexical scoping, immutable builtin value types, comprehension syntax, and currying for message passing.

Comprehensions in Monte are written similarly to Python's, but in keeping with Monte's style, the syntax elements are placed in evaluation order: `[for KEY_PATTERN => VALUE_PATTERN in (ITERABLE) if (FILTER_EXPR) ELEMENT_EXPR]`. Just as Python has dict comprehensions, Monte provides map comprehensions – to produce a map, `ELEMENT_EXPR` would be replaced with `KEY_EXPR => VALUE_EXPR`.

A list of players that got more than a quorum of votes is written `[for k => v in (counter) ? (v >= quorum) k]`. Provided there is one such player, we remove the player from the game with `players without=(victim)`.

Destructuring with Patterns

Pattern matching is used in the following ways in Monte:

1. The left-hand side of a `def` expression has a pattern.

   A single name is typical, but the first `def` expression above binds `MafiaState`, `DAY`, and `NIGHT` to the items from `makeEnum` using a list pattern.

   If the match fails, an *ejector* is fired, if provided; otherwise, an exception is raised.

2. Parameters to methods are patterns which are matched against arguments. Match failure raises an exception.

   A final pattern such as `to _printOn(out)` or with a guard `to sample(population :List)` should look familiar, but the such-that patterns in `to vote(player ? (players.contains(player)), ...)` are somewhat novel. The pattern matches only if the expression after the `?` evaluates to `true`; at the same time, `player` is usable in the such-that expression.

3. Each matcher in a `switch` expression has a pattern.

   In the `advance` method, if `state` matches the `==DAY` pattern–that is, if `state == DAY`–then `NIGHT` is assigned to `state`. Likewise for the pattern `==NIGHT` and the expression `DAY`.

   An exception would be raised if neither pattern matched, but that can't happen because we have `state :MafiaState`.

4. Match-bind comparisons such as `"<p>" =~ `<@tag>`` test the value on the left against the pattern on the right, and return whether the pattern matched or not.

5. Matchers in object expressions provide flexible handlers for message passing.

The `[=> makeEnum]` pattern syntax is short for `["makeEnum" => makeEnum]`, which picks out the value corresponding to the key `"makeEnum"`. The module_expansion section explains how imports turn out to be a special case of method parameters.

CHAPTER 4

Monte Idioms Quick Reference

These examples show Monte syntax for conventional constructs as well as workhorse idioms that are somewhat novel to Monte.

Simple Statements

```
>>> def a := 2 + 3
... var a2 := 4
... a2 += 1
... def b := `answer: $a`
... traceln(b)
... b
"answer: 5"
```

Basic Flow

```
>>> if ('a' == 'b'):
...     "match"
... else:
...     "no match"
"no match"
```

```
>>> var a := 0; def b := 4
... while (a < b):
...     a += 1
... a
4
```

```
>>> var resource := "reserved"
... try:
...     3 // 0
... catch err:
...     `error!`
... finally:
...     resource := "released"
... resource
"released"
```

```
>>> def x := [].diverge()
... for next in (1..3):
...     x.push([next, next])
... x.snapshot()
[[1, 1], [2, 2], [3, 3]]
```

```
>>> def map := ['a' => 65, 'b' => 66]
... var sum := 0
... for key => value in (map):
...     sum += value
... sum
131
```

Function

```
>>> def addTwoPrint(number):
...     traceln(number + 2)
...     return number + 2
...
... def twoPlusThree := addTwoPrint(3)
... twoPlusThree
5
```

Singleton Object (stateless)

```
>>> object adder:
...     to add1(number):
...         return number + 1
...     to add2(number):
...         return number + 2
... def result := adder.add1(3)
... result
4
```

Objects with state

```
>>> def makeOperator(baseNum):
...     def instanceValue := 3
...     object operator:
...         to addBase(number):
...             return baseNum + number
...         to multiplyBase(number):
...             return baseNum * number
...     return operator
... def threeHandler := makeOperator(3)
... def threeTimes2 := threeHandler.multiplyBase(2)
... threeTimes2
6
```

Objects self-referencing during construction

```
>>> def makeRadio(car):
...     `radio for $car`
... def makeCar(name):
...     var x := 0
...     var y := 0
...     def car # using def with no assignment
...     def myWeatherRadio := makeRadio(car)
...     bind car:
...         to receiveWeatherAlert():
...             # ....process the weather report....
...             traceln(myWeatherRadio)
...         to getX():
...             return x
...         to getY():
...             return y
...         # ....list the rest of the car methods....
...     return car
... makeCar("ferrari").getX()
0
```

Delegation

```
>>> def makeExtendedFile(myFile):
...     return object extendedFile extends myFile:
...         to append(text):
...             var current := myFile.getText()
...             current := current + text
...             myFile.setText(current)
...
... makeExtendedFile(object _ {})._respondsTo("append", 1)
true
```

File I/O and Modules

Access to files is given to the *main* entry point:

```
>>> def main(argv, => makeFileResource):
...     def fileA := makeFileResource("fileA")
...     fileA <- setContents(b`abc\ndef`)
...     def contents := fileA <- getContents()
...     when (contents) ->
...         for line in (contents.split("\n")):
...             traceln(line)
...
... main._respondsTo("run", 1)
true
```

Web Applications

Access to TCP/IP networking is also given to the *main* entry point. The `http/server` module builds an HTTP server from a TCP/IP listener:

```
import "http/server" =~ [=> makeHTTPEndpoint :DeepFrozen]
exports (main)

def hello(request) as DeepFrozen:
    return [200, ["Content-Type" => "text/plain"], b`hello`]

def main(argv, => makeTCP4ServerEndpoint) as DeepFrozen:
    def tcpListener := makeTCP4ServerEndpoint(8080)
    def httpServer := makeHTTPEndpoint(tcpListener)
    httpServer.listen(hello)
```

Data Structures

ConstList

```
>>> var a := [8, 6, "a"]
... a[2]
"a"

>>> var a := [8, 6, "a"]
... a.size()
3

>>> var a := [8, 6, "a"]
... for i in (a):
...     traceln(i)
... a := a + ["b"]
... a.slice(0, 2)
[8, 6]
```

ConstMap

```
>>> def m := ["c" => 5]
... m["c"]
5

>>> ["c" => 5].size()
1

>>> def m := ["c" => 5]
... for key => value in (m):
...     traceln(value)
... def flexM := m.diverge()
... flexM["d"] := 6
... flexM.size()
2
```

FlexList

```
>>> def flexA := [8, 6, "a", "b"].diverge()
... flexA.extend(["b"])
... flexA.push("b")
```

```
... def constA := flexA.snapshot()
[8, 6, "a", "b", "b", "b"]
```

FlexMap

```
>>> def m := ["c" => 5]
... def flexM := m.diverge()
... flexM["b"] := 2
... flexM.removeKey("b")
... def constM := flexM.snapshot()
["c" => 5]
```

Eventual Sends

```
>>> def abacus := object mock { to add(x, y) { return x + y } }
... var out := null
...
... abacus <- add(1, 2)
3
```

```
>>> def makeCarRcvr := fn autoMake { `shiny $autoMake` }
...
... def carRcvr := makeCarRcvr <- ("Mercedes")
... Ref.whenBroken(carRcvr, def lost(brokenRef) {
...     traceln("Lost connection to carRcvr")
... })
... carRcvr
"shiny Mercedes"

>>> def [resultVow, resolver] := Ref.promise()
...
... when (resultVow) ->
...     traceln(resultVow)
... catch prob:
...     traceln(`oops: $prob`)
...
... resolver.resolve("this text is the answer")
... resultVow
"this text is the answer"
```

CHAPTER 5

Python-Monte Idioms

This is a collection of common Python idioms and their equivalent Monte idioms.

Iteration

Comprehensions

Python features list, set, and dict comprehensions. Monte has list and map comprehensions, although efficient set comprehensions are missing.

The main difference between Python and Monte here is that Monte puts the for-loop construction at the beginning of the comprehension.

Python:

```
squares = [x**2 for x in range(10)]

more_squares = {x: x**2 for x in (2, 4, 6)}
```

Monte:

```
def squares := [for x in (0..!10) x ** 2]

def moreSquares := [for x in ([2, 4, 6]) x => x ** 2]
```

Enumeration

Python's `enumerate` is usually not necessary in Monte, because Monte has two-valued iteration and iterates over key-value pairs.

Python:

```
for i, x in enumerate(xs):
    f(i, x)
```

Monte:

```
for i => x in xs:
    f(i, x)
```

Objects

Classes

Monte does not have classes, but the maker pattern is equivalent.

Python:

```
class ClassName(object):
    def __init__(self, param, namedParam=defaultValue):
        self._param = param
        self._namedParam = namedParam

    def meth(self, arg):
        return self._param(self._namedParam, arg)
```

Monte:

```
def makeClassName(param, => namedParam := defaultValue):
    return object objectName:
        to meth(arg):
            return param(namedParam, arg)
```

Inheritance

Monte doesn't have class-based inheritance. Instead, we have composition-based inheritance. This means that there is not a parent class, but a parent object.

Python:

```
class Parent(object):
    def meth(self, arg):
        return arg * 2

    def overridden(self, arg):
        return arg + 2

class Child(Parent):
    def overridden(self, arg):
        return arg + 3

child = Child()
```

Monte, styled like Python:

```
def makeParent():
    return object parent:
        to meth(arg):
            return arg * 2

        to overridden(arg):
            return arg + 2

def makeChild(parent):
    return object child extends parent:
        to overridden(arg):
            return arg + 3
```

```
def child := makeChild(makeParent())
```

Monte, styled like Monte:

```
object parent:
    to meth(arg):
        return arg * 2

    to overridden(arg):
        return arg + 2

object child extends parent:
    to overridden(arg):
        return arg + 3
```

Private Methods

Neither Python nor Monte have private methods. Python has a naming convention for methods which should not be called from outside the class. Monte has an idiom for functions which cannot be called from outside the class.

Python:

```
class ClassName(object):

    _state = 42

    def _private(self):
        return self._state

    def public(self):
        return self._private()
```

Monte, styled like Python:

```
def makeClassName():
    var state := 42

    def private():
        return state

    return object objectName:
        to public():
            return private()
```

Monte, styled like Monte:

```
def makeClassName():
    var state := 42

    return object objectName:
        to public():
            return state
```

CHAPTER 6

The Type System

This is a brief overview of Monte's type system.

Monte does not have a type system, in the type-theoretic sense. Instead, Monte features guards. However, we cannot deny that guards both syntactically and semantically resemble types, so we are happy to call our guard system our "type system" and compare it to other type systems.

We use the Smallshire classification of type system features to explain Monte's typing features in a high-level overview.

Untyped

A language is *untyped* if there is only one type of value in the language. There are two common definitions here; one is used by Smallshire, and one is used by Harper. Both are worth considering, since Monte straddles the edge.

Smallshire gives Ruby as an example of a typed language. Ruby is a close relative of Monte, and by Smallshire's definition, Monte is also a typed language, in this view, because objects still have innate distinct behaviors.

In constrast, Harper equates untyped and unityped languages. This would mark Ruby, and Monte too, as untyped.

We say that Monte is untyped, for reasons similar to Harper's. Monte has a *uniform calling interface*, which means that any message can be sent to any object, and rejection is always done inside the object's message-receiving code at runtime.

Dynamic

Monte is *dynamic*; it is possible to have a name for a value without restrictions on the type of the value.

Strong

Monte values have *strong* types which resist coercion. Indeed, in Monte, coercion is a reified object protocol. Objects do not have to be coercible, and most builtin objects cannot be coerced.

Nominal

A language has *nominal* typing if types are identifiable, comparable, substitutable, etc. only if they are identical. Monte guards and interfaces have this property; in particular, Monte interfaces are not equal just by having the same

declared names and methods.

Manifest

Monte guards are *manifest* type annotations, which means that they are never inferred by canonical expansion.

Optional

Guards are *optional* and do not have to be specified. Indeed, Monte boasts *gradual* typing, which means that a Monte program can have any mix of guarded and unguarded names without affecting the correctness of guards.

CHAPTER 7

Misuse-Resistant Language Design

Several of Monte's design decisions are based on the concept of *misuse-resistant* tools which are designed to frustrate attempts to write faulty code, whether accidentally or intentionally.

Unicode Identifers

Monte has Unicode identifiers, like many contemporary languages. However, Monte generally rejects bare identifiers which other languages would accept. Instead, we require arbitrary Unicode identifiers to be wrapped with a slight decoration which serves as warning plumage.

Here are the examples from Unicode TR39 as valid Monte identifiers:

```
::"pypl"
::"toys--us"
::"1iν"
```

None of these examples are valid bare identifiers in Monte.

Other Languages

Haskell has had Unicode identifiers since Haskell 98. Haskell support for Unicode identifiers is detailed in the Haskell 98 Report Lexical Structure. Haskell accepts "pypl" as a bare identifier for names.

Python 3 added Unicode identifiers in PEP 3131. Python 3 accepts "pypl" as a bare identifier for names and attributes.

Parenthesized Sub-Expressions

Whenever an expression is syntactically contained within another expression, it must be parenthesized, with the sole exception of common guard-exprs used in patterns. This feature, explained in more detail in The Power of Irrelevance, improves readability by clearly distinguishing patterns from expressions.

CHAPTER 8

Secure Distributed Computing

Practical Security II: The Mafia IRC Bot (WIP)

To demonstrate secure distributed programming in Monte, let's take the mafia game code developed earlier and make it into an IRC bot.

The `mafiabot.mt` module begins by importing the `mafia` module, an `irc/client` library, and the same modules for dealing with entropy that we saw before:

```
import "mafia" =~ [=> makeMafia :DeepFrozen]
import "irc/client" =~ [=> makeIRCClient :DeepFrozen,
                        => connectIRCClient :DeepFrozen]
import "lib/entropy/entropy" =~ [=> makeEntropy :DeepFrozen]
import "lib/entropy/pcg" =~ [=> makePCG :DeepFrozen]
exports (main)
```

The `main` entry point is provided with a number of powerful references as named arguments:

- To seed our random number generator, we use `currentRuntime` to get a source of true randomness, i.e. secure entropy.
- To give `makeIRCService` access to TCP/IP networking and event scheduling, we use `makeTPC4ClientEndPoint`, `getAddrInfo`, and `Timer`.

```
def main(argv,
         => makeTCP4ClientEndpoint,
         => Timer,
         => currentRuntime,
         => getAddrInfo) as DeepFrozen:
    def [_, seed] := currentRuntime.getCrypt().makeSecureEntropy().getEntropy()
    def rng := makeEntropy(makePCG(seed, 0))
    def [hostname] := argv
    def irc := makeIRCService(makeTCP4ClientEndpoint, getAddrInfo, Timer,
                              hostname)
    irc.connect(makeMafiaBot(rng))
```

We can go ahead and run this code from a file by using the `monte` commandline tool:

```
monte eval mafiabot.mt chat.freenode.net
```

Everything after the source filename is passed to main in `argv` as a list of strings.

Networking

Unlike many other contemporary programming languages, Monte does not need an additional networking library to provide solid primitive and high-level networking operations. This is because Monte was designed to handle networking as easily as any other kind of input or output.

```
def makeIRCService(makeTCP4ClientEndpoint, getAddrInfo, Timer,
                   hostname :Str) as DeepFrozen:
    def port := 6667  # TODO: named arg with default value

    return object IRC:
        to _printOn(out):
            out.print(`IRC($hostname)`)

        to connect(handler):
            def client := makeIRCClient(handler, Timer)

            def addrs := getAddrInfo(b`$hostname`, b``)
            return when (addrs) ->
                def choices := [
                    for addr in (addrs)
                    ? (addr.getFamily() == "INET" &&
                       addr.getSocketType() == "stream") addr.getAddress()]
                def [address] + _ := choices
                def ep := makeTCP4ClientEndpoint(address, port)
                connectIRCClient(client, ep)
                client
```

Distributed Systems

Monte comes with builtin explicit parallelism suitable for scaling to arbitrary numbers of processes or machines, and a well-defined concurrency system that simplifies and streamlines the task of writing event-driven code.

Monte has one concurrent operation. Monte permits messages to be passed as **eventual sends**. An eventually-sent message will be passed to the target object at a later time, generating a **promise** which can have more messages sent to it. Unlike similar mechanisms in Twisted, Node.js, etc., Monte builds promises and eventual sending directly into the language and runtime, removing the need for extraneous libraries.

Monte also has a single primitive for combining isolation and parallelism, the *vat*. Each vat isolates a collection of objects from objects in other vats. Each eventual send in a vat becomes a distinct **turn** of execution, and vats execute concurrently with one another. During a turn, a vat delivers a single queued send, which could result in more sends being queued up for subsequent turns.

```
def makeChannelVow(client, name) as DeepFrozen:
    "Return a vow because say() won't work until we have joined."
    def [wait, done] := Ref.promise()
    var waitingFor :NullOk[Set[Str]]:= null

    object chan:
        to _printOn(out):
            out.print(`<channel $name>`)
        to getName():
            return name
        to hasJoined():
            return client.hasJoined(name)
        to say(message) :Void:
            client.say(name, message)
```

```
        to getUsers(notReady):
            return client.getUsers(name, notReady)
        to waitFor(them :Set[Str]):
            waitingFor := them
            return wait
        to notify():
            if (waitingFor != null):
                escape oops:
                    def present := chan.getUsers(oops).getKeys().asSet()
                    traceln("notify present:", present, waitingFor,
                            waitingFor - present)
                    if ((waitingFor - present).size() == 0):
                        waitingFor := null
                        done.resolve(present)
        to tell(whom, what, notInChannel):
            if (chan.getUsers(notInChannel).contains(whom)):
                client.say(whom, what)
            else:
                notInChannel(`cannot tell $whom: not in $name`)
        to part(message):
            client.part(name, message)
    return when(chan.hasJoined()) ->
        chan
```

Principle of Least Authority

Straightforward object-oriented design results in each object having the least authority it needs:

- `makeIRCService` provides the full range of IRC client behavior
- `makeChannelVow` provides access to one channel
- `makeModerator` encapsulates the play of one game
- `makePlayer` represents the role of one player in one game
- `makeMafiaBot` starts games on request, routes messages to the relevant moderator during game play, and disposes of moderators when games end.

Even if one of these components is buggy or compromised, its ability to corrupt the system is limited to using the capabilities in its static scope.

Contrast this with traditional identity-based systems, where programs execute with all privileges granted to a user or role. In such a system, any compromise lets the attacker do anything that the user could do. A simple game such as solitaire executes with all authority necessary to corrupt, exfiltrate, or ransom the user's files.

With object capability discipline, when the time comes for a security inspection, we do not have to consider the possibility that any compromise in any part of our program leaves the whole system wide open in this way. Each component in the system can be reviewed independently and auditing a system for security becomes cost-effective to an extent that is infeasible with other approaches [1].

```
def makeModerator(playerNames :Set[Str], rng,
                  chan :Near, mafiaChan) as DeepFrozen:
    def [=> game, => mafiosos] := makeMafia(playerNames, rng)
    var night0 := true

    def makePlayer(me :Str):
        return object player:
```

[1] As documented in the DarpaBrowser report

```
            to _printOn(out):
                out.print(`<player $me>`)
            to voteFor(nominee :Str):
                try:
                    game.vote(me, nominee)
                catch _:
                    # nominee is not (any longer) a player
                    return
                chan.say(game.advance())

    def toPlayer := [for nick in (playerNames) nick => makePlayer(nick)]

    return object moderator:
        to _printOn(out):
            out.print(`<moderator in $chan>`)

        to begin():
            # Night 0
            chan.say(`$game`)
            when (mafiaChan) ->
                escape notHere:
                    for maf in (mafiosos):
                        chan.tell(
                            maf, `You're a mafioso in $chan.`, notHere)
                        chan.tell(
                            maf, `Join $mafiaChan to meet the others.`, notHere)
                traceln("waiting for", mafiosos, "in", mafiaChan)
                when (mafiaChan.waitFor(mafiosos)) ->
                    traceln("done waiting for", mafiosos)
                    night0 := false
                    # Morning of day 1...
                    chan.say(game.advance())

        to said(who :Str, message :Str) :Bool:
            "Return true to contine, false if game over."
            mafiaChan.notify()
            traceln("notifying", mafiaChan)
            if (night0):
                return true
            if (message =~ `lynch @whom!`):
                escape notPlaying:
                    def p := moderator.getPlayer(who, notPlaying)
                    p.voteFor(whom)
                    traceln("lynch", who, whom)

                    if (game.getWinner() =~ winner ? (winner != null)):
                        moderator.end()

            return game.getWinner() == null

        to getPlayer(name, notPlaying):
            return toPlayer.fetch(name, notPlaying)

        to end():
            chan.say(`$game`)
            chan.part("Good game!")
            mafiaChan.part("bye bye")
```

Note the way `makeMafiaBot` provides a secret channel for the mafiosos to collude at night:

```
def makeMafiaBot(rng) as DeepFrozen:
    def nick := "mafiaBot"
    def chanMod := [].asMap().diverge()
    def keys := [].asMap().diverge()

    return object mafiaBot:
        to getNick():
            return nick

        to loggedIn(client):
            return null

        to privmsg(client, user, channel, message):
            # traceln("mafiaBot got", message, "on", channel, "from", user,
            #         "channels", chanMod.getKeys())
            def who := user.getNick()

            if (message =~ `join @dest` &&
                channel == nick &&
                !keys.contains(dest)):
                mafiaBot.join(client, who, dest)
            else if (message == "start" &&
                     !keys.contains(channel)):
                when(def chan := makeChannelVow(client, channel)) ->
                    mafiaBot.startGame(client, chan, channel)
            else if (chanMod.snapshot() =~ [(channel) => m] | _):
                if (!m.said(who, message)):
                    def chKey := keys[channel]
                    chanMod.removeKey(channel)
                    chanMod.removeKey(chKey)
                    keys.removeKey(channel)
                    keys.removeKey(chKey)
                    traceln("removed", channel, chKey)

        to join(client, who :Str, channel :Str):
            when(client.hasJoined(channel)) ->
                client.say(channel, `Thank you for inviting me, $who.`)
                client.say(channel, `Say "start" to begin.`)

        to startGame(client, chan :Near, channel :Str):
            def secret := `$channel-${rng.nextInt(2 ** 32)}`
            def secretChan := makeChannelVow(client, secret)
            escape notReady:
                def users := chan.getUsers(notReady)
                def playerNames := [
                    for name => _ in (users)
                    ? (name != nick)
                    # @chanop -> chanop
                    (if (name =~ `@@@op`) { op } else { name })]
                traceln("players:", playerNames, users)

                def m := makeModerator(playerNames.asSet(), rng,
                                       chan, secretChan)
                chanMod[channel] := chanMod[secret] := m
                keys[channel] := secret
                keys[secret] := channel
                m.begin()
```

Notes

Vats

Vats are Monte's response to the vagaries of traditional operating-system-supported threads of control. Vats extend a modicum of parallelism and concurrency to Monte programs while removing the difficult data races and lock management that threads classically require.

Quickstart

From an entrypoint, the `currentVat` named argument will refer to the "top" or "first" vat:

```
> currentVat
Result: <vat(pa, immortal, 2 turns pending)>
```

Note: This vat is named "pa", is "immortal", which means that it will never terminate computation abruptly, and has two turns of computation pending in its turn queue. All of this diagnostic information is Typhon-specific and may not be available in all implementations.

We can *sprout* a new vat at any time from an existing vat. The two vats will be distinct:

```
> def newVat := currentVat.sprout("re")
Result: <vat(re, immortal, 0 turns pending)>
> newVat == currentVat
Result: false
```

We can also *seed* a vat with a computation. The computation must be `DeepFrozen`, but otherwise any object can be used as a seed. This example is a bit dry but shows off the possibilities:

```
> newVat
Result: <vat(re, immortal, 0 turns pending)>
> def seed() as DeepFrozen { traceln("Seeding!"); return fn x { traceln(`I was sent $x`) } }
Result: <seed>
> def seeded := newVat.seed(seed)
TRACE: From vat re
 ~ "Seeding!"
Result: <promise>
> seeded<-(42)
Result: <promise>
TRACE: From vat re
 ~ "I was sent 42"
> seeded<-(object popsicle as DeepFrozen {})
Result: <promise>
TRACE: From vat re
 ~ "I was sent <popsicle>"
> seeded<-(object uncopyable {})
Result: <promise>
TRACE: From vat re
 ~ "I was sent <promise>"
```

Seeding produces a far reference to the result of the seed's call, which might not be itself `DeepFrozen`. To interact with this reference, send messages to it. Note how sending `popsicle` caused the seeded object to receive a near (and thus printable) reference to it; this is because `DeepFrozen` objects travel between near vats directly.

What's in a Vat?

The Browser Analogy

A vat, by analogy, is like a tab in a modern Web browser. It contains some objects, which may have near references between themselves, and a queue of pending messages to deliver to some of those objects. A browser tab might have some JavaScript to run; a vat might choose to take a **turn**, delivering a message to an object within the vat and letting the object pass any subsequent messages to its referents. Vats can be managed just like browser tabs, with vats being spawned and destroyed according to the whims of anybody with references to those vats. Indeed, vats can be managed just like any other object, and vats are correct with regards to capability security.

Vats, Formally and Informally

This is all confusing. What, precisely, **is** a vat?

Formally, a vat is just a container of objects. Vats have a **turn queue**, a list of messages yet to be delivered to objects within the vat, along with an optional resolver for each message. Vats compute by repeatedly delivering individual messages in the turn queue; each delivery is called a **turn**. Turns are taken in the order that they are enqueued, FIFO.

If a resolver is provided for a turn, then the resolver is resolved with the result of delivery. If delivery causes an exception, then the vat catches the exception, sealing it, and smashes the resolver with the exception instead. In either case, a **membrane** is applied to all objects which come into or leave the vat, including the result of delivery; this membrane replaces all non-`DeepFrozen` values with far references.

Informally, a vat isolates an object graph. Objects inside the vat can only refer to things outside the vat by far reference; there is no way to perform an immediate call across a vat boundary.

Whenever an object sends a message into a vat, the vat prepares to take a **turn**, whence the message will be delivered to the correct object inside the vat. Sends out of the vat produce promises for references to results of those sends, and the promises have normal error-handling behavior; if you send a message to another vat, and an exception happens in that other vat, then you'll get a broken promise.

Vat Interface

Vats have two methods, `.sprout/1` and `.seed/1`.

Why is `.sprout/1` synchronous?

A common theme in Monte's vat design is implicit and convenient asynchronous computation. So why is vat sprouting synchronous? Well, Monte's guiding philosophy is to never block. But producing a vat is a non-blocking operation, since a sprouted vat starts out empty, and vats are isolated, so the new vat cannot affect the current vat's current turn.

In general, vats queue up work to do later. Since adding things to turn queues is non-blocking, vats return promises for the work to be done later.

However, this isn't the whole story. It's true that vats aren't *totally* empty; they generally acquire a safe scope as a result of pass-by-copy semantics. A Monte implementation which supports many small vats is expected to implement a copy-on-write semantics for objects in vats. This is one of the compelling use cases for `DeepFrozen`; a `DeepFrozen` object graph, like the safe scope or a vat seed, can live on a shared heap and be zero-copy shared between all vats.

To sprout a new vat, call `vat.sprout(name :Str) :Any`, which returns a new vat. The new vat starts out empty, with an empty turn queue.

To put computation into a vat, call `vat.seed(seed :DeepFrozen) :Vow`, which does several things. First, the seeded vat copies the `seed` and its object graph into itself, isolating them from the calling vat. Then, the vat adds `seed<-()` to its turn queue, and returns a promise for that pending turn.

FAQ

Vats are one of the more confusing parts of Monte, and some questions occur frequently.

So, no threads?

Correct. Monte does not have any way to block on I/O, so there is no need for threads at the application level.

Are vats parallel or concurrent?

It is implementation-dependent. Currently, Typhon is designed for an M:N threading model where up to M vats may take N turns in parallel on N distinct threads. However, Typhon currently only takes 1 turn in parallel. Other implementations may choose to do different parallelism models.

A key insight with vats is that a computation that is broken up into *concurrent* pieces on distinct vats can be transformed into *parallel* execution with maximal parallelism just by altering the underlying interpreter. The correctness of the computation does not change. This concept is from the actor model, which forms the theoretical basis for vats.

How do I perform parallel computations today?

Today, using Typhon, use the `makeProcess` entrypoint capability to run multiple processes to get node-level parallelism. We recognize that this is a very unsatisfactory solution for all involved, and we plan to eventually implement automatic parallel vats in Typhon.

For the future... Try to structure your code into modules; Typhon may parallelize module loading in the future. Also try to structure your code into vats, since we expect most interpreters to eventually implement parallel vat execution.

How do I perform concurrent operations?

Spawn more vats. All vats are concurrently turning. A vat will only ever lie fallow when it has no turns queued.

Why should we ever make synchronous calls?

In a nutshell, always make calls unless you intentionally want to create an asynchronous "edge" where your control flow stops, only to resume later. And also when you're working with promises and far references, since you can't make calls on those values!

Synchronous calls are very common. There are many kind of objects on which synchronous calls work, because they are near references. For example, all literals are near, and so is all operator syntax:

```
def lue := 6 * 7
```

There are many objects in the safe scope which are perfectly fine to use with either calls or sends.

Here are some handy idioms. To check whether a value is near:

```
Ref.isNear(value)
```

A variant that might be more useful in the future:

```
value =~ n :Near
```

No, you misunderstood; why doesn't Monte have only eventual sends?

Ah! There are several reasons, to be taken together as a measure of how difficult such a system would be to work with.

Execution speed is very bad in these systems. This is because it is quite difficult for any compiler to see, even with cutting-edge technology, where a sent message will be delivered to, since it travels in both time and space before being resolved. While our general feeling is that speed is a secondary concern in most cases, we are motivated to care here for two reasons. First, practical compilers tend to do enormous amounts of work to convert chains of monomorphic sends into calls; GHC has a strictness analyzer to avoid lazy thunk chains on the heap, which have similar delayed-evaluation properties to sends. Second, Joule, an ancestor of Monte, tried this design approach and found speed to be a serious problem.

Some edges of Monte's interaction with the external world are much better-modeled with calls than sends. A chauvanist argument can be made about how arithmetic should at least occasionally be lowered to a sequence of CPU instructions. However, we have found that a trickier and more important problem is dealing with object graph recursion, since Monte object graphs already can be quite treacherous. In Monte, object graphs can be cyclical and can hold delayed or eventual values. This poses a serious challenge, since sends for traversal can end up interleaved with sends which alter the structure or contents of the graph being traversed. Concretely:

- Equality testing: `x == y` is a question that can, if they are `Transparent`, traverse the full transitive closures of both `x` and `y`.
- Serialization: Pretty-printing, databases, RPC, DOT files, and all other serialization must traverse the full object graph as-is in order to not write out corrupted snapshots.
- Hashing: Implementations may choose to define internal object hashes to speed up sets and maps. Application-level probabalistic data structures also often perform hashing. Like serialization, but just different enough to justify three sentences and a bullet point.
- Garbage collection: GCs in the current state of the art are increasingly concurrent, running alongside mutators or only performing collections on per-mutator heaps. Nonetheless, when the GC would like to perform a collection, it often does need to traverse the object graph without worrying that an object will not race its own impending deletion with an incoming message delivery. This could be dealt with by requiring all sends to go through the vat turn queue, and pausing the vat in-between turns to collect. But then speed concerns pop up, and really this is a very deep rabbit hole...

So, for these reason, we distinguish promises at the edges of our object graphs, and we implement these traversals using calls. As a practical consequence, *uncalls* are calls and must return near values. This also influenced the design of printers, which serialize by pretty-printing, and vats, which could optionally be implemented with per-vat GC.

Brands

The *brand pattern* divides the capability of establishing a secure communication channel into two facets, called a *sealer* and *unsealer*.

```
def [ana, cata] := makeBrandPair("finney")
def box := ana.seal(42)
cata.unseal(box)
```

The resulting channel has the following properties:

- **Authentic and Unforgeable**: Boxes created by the sealer cannot be unsealed by any object other than the unsealer; to the contrapositive, any object that the unsealer unseals must have been sealed with the corresponding sealer.

- **Asynchronous**: Boxes created by the sealer can be unwrapped on any subsequent turn.
- **Untyped**: Any object can be transmitted along the channel.

Up & Down

To create a new brand, call `makeBrandPair(nickname :Str)`. The nickname is purely cosmetic, to aid readability and debugging; it does not have to be unique.

```
# Make a sealer named `ana` and an unsealer named `cata`.
def [ana, cata] := makeBrandPair("finney")
```

The brand itself is an opaque object which proves that a sealer and unsealer are paired with each other. It is accessible via the `.getBrand/0` method:

```
# Hey, these two are a pair!
ana.getBrand() == cata.getBrand() # should be true
```

Brands are usable as map keys:

```
def brandMap := [ana.getBrand() => [ana, cata]]
brandMap[cata.getBrand()] # should be `[ana, cata]`
```

The fundamental operation of a sealer is to `.seal/1` an object into a box:

```
def box := ana.seal(42)
box # <box sealed by finney>
```

The unsealer, unsurprisingly, provides `.unseal/1`, which opens a box and returns its contents:

```
cata.unseal(box) # should be 42
```

The box is opaque and yields only one useful method, `.getBrand/0`, which can be useful for determining which unsealer might be the correct one to use for unsealing:

```
brandMap[box.getBrand()] # should be `[ana, cata]`
```

Note: The implementation of `makeBrandPair` in the Typhon prelude has other methods defined on boxes, but they do not affect the security guarantees of the implementation.

Promises

> Promises are a great way of dealing with eventual values, allowing one to compose and synchronise processes that depend on values that are computed asynchronously.
>
> —Quil

Monte provides user-controllable transparent proxy objects, **promises**, for highly customized asynchronous workflows.

Basic Promises

The basic usage of promises is to create a pair of objects, called the promise and the **resolver**:

```
# Traditionally, promises are named "p" and resolvers are named "r".
def [p, r] := Ref.promise()
```

The `Ref` object in the safe scope can produce promise/resolver pairs. It also has many utility methods for manipulating promises.

A promise is a **transparent proxy**; it does not expose its own behavior via message passing, but instead forwards all received messages to another object. Instead, the resolver and `Ref` object coordinate to control the behavior of the promise:

```
# This next line will throw an exception; the promise isn't yet resolved,
# so it can't deliver this immediate call.
p.add(5)
# We can resolve the promise, at which point the promise will forward
# immediate calls to its resolved value.
r.resolve(7)
# And now we succeed!
p.add(12)
```

Promises do not just resolve; they can also **break**. A **broken** promise will never resolve, but instead refers to a **problem**, which is an object (often a string) describing a failure.

```
# Here we create a promise...
def [p, r] := Ref.promise()
# And now we break the promise!
r.smash(`Promise was broken, sorry!`)
# Referencing or using the promise will throw...
p.add(12)
# ...but some operations are still safe.
Ref.optProblem(p)
```

When-expressions and Delayed Actions

Promises are commonly used to perform delayed actions which will execute at some later time.

To queue an action, use an eventual send:

```
# This message will be delivered on some later turn.
def q := p<-add(5)
```

What is `q`? `q` is *another* promise. It will be resolved automatically, sometime after `p` resolves, with the value that `p` returned from its sent message; in this case, if `p` was `7`, then `q` would be `12`.

Suppose that the action that we want to enqueue is more complex than a single passed message. In that case, Monte provides the when-expression:

```
# When the promise resolves, notify the user and start the next section.
when (p) ->
    traceln(`Attention user: The promise $p has resolved.`)
    # This funny-looking syntax means to use the default verb of "run",
    # just like with a normal call.
    nextSection<-()
catch problem:
    # Something went wrong. Better notify the user.
    traceln(`Attention user: There was a problem: $problem`)
    nextSection<-failed()
```

The when-expression consists of a when-block and an optional catch-block. When the promise given to the when-expression becomes resolved, the when-block will run on its own turn; if the promise is broken, then the catch-block will run instead.

Streamcaps

Stream capabilities ("streamcaps") are objects which implement a protocol for streaming data. Monte directly supports the streamcap protocol with unsafe objects and standard library tooling. The protocol is designed to be simple to implement and easy to reason about.

Quick Overview

There are three interfaces to the streamcap protocol, called **sources**, **sinks**, and **pumps**. Objects may only implement one interface at a time. Sources generate data, sinks consume data, and pumps transform data.

The simplest usage is delivering a single datum from a source to a sink:

```
source(sink)
```

We can enqueue an action to execute after delivery has succeeded:

```
when (source(sink)) -> { action() }
```

We can also handle errors in case of failed delivery:

```
when (source(sink)) -> { action() } catch problem { rescue(problem) }
```

Hand-delivering data to a sink is easy:

```
for datum in (data) { sink(datum) }
```

To receive data from a source, write an inline sink object:

```
object sink:
    to run(datum):
        return process<-(datum)
    to complete():
        success()
    to abort(problem):
        throw(problem)
source(sink)
```

In the standard library, the "lib/streams" module has tools for manipulating streamcaps. To deliver all (zero or more) data from a source to a sink, we can use the `flow` helper:

```
import "lib/streams" =~ [=> flow]
when (flow(source, sink)) -> { done() }
```

Object Protocol

Pumps

Pumps are transformers of data. A pump does not participate in any sort of flow control, but merely operates on data passing through.

The sole method of pumps is `run/1`, which takes a single datum and returns a list of zero or more data.

```
var acc :Int := 0
def accumulatingPump(i :Int) :List[Int] as Pump:
    "Accumulate a sum of integers."
    acc += i
    return [acc]
```

Warning: Unlike the rest of the streamcap protocol, pumps must currently be synchronous; they must return `List`. In the future, pumps should be able to return `Vow[List]`.

Sinks

Sinks are data consumers. A sink receives data and returns asynchronous signals indicating the fate of each received datum.

Sinks have three methods: `run/1`, `complete/0`, and `abort/1`. `run/1` is for delivering data to the sink, and returns a `Vow[Void]` which succeeds when delivery completes, or breaks when delivery fails:

```
when (sink(datum)) ->
    traceln("Delivery complete!")
catch problem:
    traceln("Delivery failed:")
    traceln.exception(problem)
```

The `complete/0` and `abort/1` methods inform the sink that no more data will be delivered. `complete/0` is for successful termination, and `abort/1` is for failed termination, with a problem. After a sink has terminated, further deliveries may behave in arbitrary ways. In general, sinks will usually raise exceptions or return broken promises if data is delivered after termination.

Sink Semantics

What does "delivery" really mean? A sink could decide that data is delivered when it is enqueued in an internal buffer, or sent onward to a remote resource. A sink should not indicate that delivery has succeeded until the sink is ready to receive more data, in order to provide implicit backpressure.

Aborting a sink may alter the behavior of the sink with regards to enqueued or processing data. In particular, TCP connections and streaming file handles may close uncleanly after being aborted. Sinks are allowed to have this behavior because sinks are only required to flush upon being cleanly terminated.

Sources

Sources are data emitters. A source receives sinks and delivers data to those sinks.

Sources only have one method, `run/1`, which takes a sink:

```
source(sink)
```

Just like `run/1` of sinks, sources return a `Vow[Void]` indicating whether the sink was called successfully:

```
when (source(sink)) -> { success() }
```

A typical source will return the sink's delivery notification directly:

```
def cat():
    return "meow"

def catSource(sink) as Source:
    return sink(cat)
```

Patterns

Flow

The most common pattern for streamcaps is *flowing* all data from a source to a sink. Use the `flow` helper from "lib/streams" to make this easy. Here's a complete TCP echo server:

```
import "lib/streams" =~ [=> flow :DeepFrozen]
exports (main)

def main([via (_makeInt) port], => makeTCP4ServerEndpoint) as DeepFrozen:
    def handler(source, sink):
        return flow(source, sink)
    def ep := makeTCP4ServerEndpoint(port)
    ep.listenStream(handler)
    return 0
```

Tubes

Deprecated since version unstable: Tubes have been deprecated in favor of Streamcaps.

Tutorial

Monte provides a unified paradigm for handling streams of structured data. The paradigm is known as *tubes*.

Tubes come in two flavors: *founts* and *drains*. A fount is an object which can provide data to another tube. A drain is an object which can receive data from another tube. A tube can be just a fount, just a drain, or both a fount and a drain.

This is all pretty abstract. Let's roll up our sleeves and take a look at how to use some tubes:

```
def echo(fount, drain):
    fount.flowTo(drain)
```

This code instructs `fount` to provide data to `drain`. This providing of data will happen whenever `fount` wants, until either `fount` or `drain` indicate that flow should cease. While this example might seem trivial, it's sufficient to use as e.g. a TCP echo server.

Sometimes founts receive their data between turns, and schedule special turns to send data to drains. Other times founts are eager, and try to feed a drain immediately during `flowTo`. If you want to forcibly delay that eagerness until another turn, just use an eventual send:

```
def echo(fount, drain):
    fount<-flowTo(drain)
```

If a drain is also a fount, then `flowTo` will return a new fount which can be flowed to another drain. This is called *tube composition* or *tube fusion* and it is an important concept in tube handling.

Pumps

Sometimes an operation on streaming chunks of data only cares about the data and not about the streaming or chunking. Such an operation can be encapsulated in a *pump*, which is like a tube but with no flow control. A pump takes one item at a time and should return zero or more items.

Pumps are mostly useful because they can be wrapped into tubes, which can then be composed with other tubes:

```
def [=> makeMapPump] | _ := import("lib/tubes/mapPump")
def [=> makePumpTube] | _ := import("lib/tubes/pumpTube")

def negate(fount, drain):
    def tube := makePumpTube(makeMapPump(fn x {-x}))
    fount<-flowTo(tube)<-flowTo(drain)
```

This pump uses a mapping function to negate every element that flows through it, without any concern over flow control.

Working with Packages

The source code for the Mafia game and IRC bot are in their own git repository, https://github.com/monte-language/mt-mafia . Let's download and run it:

```
git clone https://github.com/monte-language/mt-mafia
cd mt-mafia
monte test mafiabot
monte run mafiabot chat.freenode.net
```

This should result in the bot connecting to IRC and being ready to receive commands.

Monte packages are defined by a file in the project root directory named `mt.json`. This file includes package metadata and a list of dependencies. Previous to the first run, a Nix package is built from the project and its dependencies (currently these can either be from a local directory or a Git repository). The `monte test` command collects all unit tests in the project and starts the test runner, whereas `monte run` invokes the `main` function in mafiabot.mt. (The build step can be invoked directly using `monte build`.)

The format for `mt.json` is a JSON file with the following keys:

name A name for the package.

paths A list of paths relative to the project root that contain Monte code. "." is acceptable if it's in the root.

entrypoint The name of the module with the `main` function to invoke. Optional.

dependencies An object with package names as keys and dependency descriptions as values. Dependency descriptions are objects with `url` keys naming a location to fetch the dependency from, and optionally `type` (either "git" or "local" – defaults to `git` if omitted) and `commit` (describing the git revision to fetch) keys.

Building the Nix package involves first creating an `mt-lock.json` file with a full list of all dependencies and their versions. You may keep this file to pin your builds to specific versions or get rid of it to re-run the dependency discovery process.

CHAPTER 9

Language Reference

Quasiliterals

Quasiliterals, or QLs, are an important part of Monte syntax which allows us to embed arbitrary DSLs into Monte. With the power of QLs, Monte can be extended into new territory in a very neat way.

What's a Quasiliteral?

This is a quasiliteral:

```
`Backticks start and end quasiliterals`
```

A quasiliteral can have values mixed into it with `$`. A value can be a name:

```
def name :Str := "Todd"
`Hello, $name!`
```

A value can also be an expression, using brackets:

```
`2 + 2 = ${2 + 2}`
```

Quasiliterals can be used as patterns:

```
# Equivalent to: def =="self" := "self"
def `self` := "self"
```

Quasiliteral patterns also permit pattern-matching with `@` to retrieve single names:

```
def `(@first, @second)` := "(42, 5)"
```

And any pattern can be used with brackets:

```
def `x := @{var x}` := "x := 7"
x += "-11" # What? I like slushies!
```

Finally, there are different *quasiparsers*, or QPs, which each have different behavior:

```
# `` makes strings
`def x := 42` :Str
# b`` makes bytestrings
b`def x := 42` :Bytes
# m`` makes Monte AST objects
m`def x := 42` :(astBuilder.getAstGuard())
```

How to Use QLs

A quasiliteral expression starts with the name of a quasiparser (which can be empty) followed by a backtick. Then, a mixture of strings and holes are allowed, followed by a final backtick. The holes can either be expression-holes, with `$`, or pattern-holes, with `@`.

> **Warning:** Pattern-holes cannot be used in QL expressions, only in QL patterns. Using a pattern-hole in a QL expression is a syntax error!

Builtin Quasiparsers

There are three common QPs included in Monte's safe scope.

Simple

> **Did You Know?**
>
> Monte originally used the same name as E for ::"``": `simple__quasiParser`. That's why we call ::"``" the "simple" quasiparser.

The simple or empty QP builds strings:

```
`string` == "string" # true
```

It can mix any value into a string, even values that don't pass `Str`:

```
`${7}` == "7" # true
```

The simple QP does this by calling `M.toString/1` on the values. Correspondingly, the value's `_printOn/1` is called, and can be customized:

```
object shirt { to _printOn(out) { out.print("tye-dye shirt") } }
def description :Str := `I am wearing a $shirt.`
```

When used as a pattern, the simple QP performs very simple but straightforward and powerful string parsing:

```
def container := "glass"
def `a $container of @drink` := "a glass of lemonade"
```

Bytes

The bytes QP builds bytestrings:

```
b`asdf`
```

The encoding of characters is unconditionally Latin-1. Non-Latin-1 characters cause errors to be thrown at runtime:

```
b`ErrorRaiser™`
```

Other than that quirk, the bytes QP behaves much like the simple QP, including parsing:

```
def b`@header:@value` := b`x:12`
```

Monte

Finally, the Monte QP builds Monte ASTs from literal Monte source:

```
m`def x := 42`
```

The Monte QP can be used for code generation, since it evaluates to objects usable with `eval/2`:

```
eval(m`2 + 2`, [].asMap())
```

Custom Quasiparsers

Anybody can write their own quasiparser.

Parsing with Values

The first half of the QP API deals with building the initial structure and including values.

`.valueHole(index :Int)` should create a value marker which can be used in place of some value which will be included later. `.valueMaker(pieces :List)` will be called with a list of pieces, which can be either strings or value markers, and it should return a partial structure. That structure can be completed with its `.substitute(values :List)`, which provides a list of values that can be swapped with the value markers.

To see how this API all comes together, let's look at the kernel expansion of a simple QP call:

```
`Just another $day for this humble $string.`
```

What Monte actually does is call `.valueMaker/1`, like so:

```
::"``".valueMaker(["Just another ", ::"``".valueHole(0),
                   " for this humble ", ::"``".valueHole(1),
                   "."]).substitute([day, string])
```

Parsing Patterns

The pattern API is similar and builds upon the expression API.

First, the `.patternHole/1` method allows pattern hole markers to be built, just like with value holes. Then, the structure is built with `.matchMaker/1` instead of `.valueMaker/1`. This structure should have a completion method, `.matchBind(values :List, specimen, ej)` which attempts to unify the specimen with the structure completed by the values or eject on failure.

Here's a simple pattern:

```
def `how ${hard} could it be to match @this?` := "not hard, just complex"
```

And its expansion:

```
def via (_quasiMatcher.run(::"``".matchMaker(["how ", ::"``".valueHole(0),
                                              " could it be to match ",
                                              ::"``".patternHole(0),
                                              "?"]),
                           [hard])) [this] := "not hard, just complex"
```

Note how the `_quasiMatcher` helper in the safe scope takes care of the extra runtime plumbing.

Interfaces

An *interface* is a syntactic expression which defines an object protocol. An interface has zero or more method signatures, and can be implemented by any object which has methods with equivalent signatures to the interface.

Let's jump right in:

```
interface Trivial:
    "A trivial interface."
```

This interface comes with a docstring, which is not required but certainly a good idea, and nothing else. Any object could implement this interface:

```
object trivia implements Trivial:
    "A trivial object implementing a trivial interface."
```

When an object **implements** an interface, the interface behaves like any other auditor and examines the object for compliance with the object protocol. As with other auditors, the difference between the "implements" and "as" keywords is whether the object is required to pass the auditor:

```
object levity as Trivial:
    "A trivial object which is proven to implement Trivial."
```

Let's look at a new interface. This interface carries some **method signatures**.

```
interface GetPut:
    "Getting and putting."
    to get()
    to put(value)

object getAndPut as GetPut:
    "A poor getter and putter."

    to get():
        return "get"

    to put(_):
        null
```

We can see that `getAndPut` implements the `GetPut` interface, but it isn't very faithful to that interface. Interfaces cannot enforce behavior, only signatures.

Miranda Protocol

> If you cannot afford a method, one will be appointed for you.

Monte objects, left to their own devices, are black boxes; one cannot perform any sort of introspection on them. However, there are some powers granted to anybody who can refer to an object. The runtime grants these powers automatically, and we refer to them as the **Miranda protocol**.

The Miranda protocol grants powers in the form of methods, called **Miranda methods**, which all objects automatically possess. An object may provide its own Miranda methods, but does not have to; objects are automatically granted default Miranda methods with correct behavior. Or, as stated above, "if an object does not have a Miranda method, one will be provided."

Additionally, the Miranda protocol contains **Miranda named arguments**, which are named arguments passed alongside every message to every object from the runtime.

Safety

Methods

Miranda methods should be safe to call. The default definitions will always respond without throwing exceptions. It is rude but permissible for an object to provide a custom Miranda method implementation which can throw or eject, or return incorrect or misleading information. Therefore, be aware of situations in which Miranda methods are being used.

> **Warning:** Special mention goes here to the most commonly-called Miranda method, `_printOn/1`. Any time that an object is being turned into a string, it almost certainly involves a little bit of `_printOn/1`, so be careful.

Named Arguments

See *FAIL*.

Methods

`_conformTo/1` `_conformTo` takes a guard and coerces this object to that guard, if possible. The default implementation returns `null` for all guards. Overriding this method lets an object become other objects when under scrutiny by guards.

`_getAllegedInterface/0` `_getAllegedInterface` returns an interface describing this object. If not specified, an interface which represents the object faithfully will be created and returned.

The allegedness of the interface hinges on the ability to override this method; the returned interface can be just as untrustworthy as the object that returns it.

`_printOn/1` `_printOn` writes text representing this object onto the printer passed as an argument.

Customizing `_printOn` lets an object change how it is pretty-printed. The default pretty-printing algorithm is readable but does not divulge the internal state of an object.

`_respondsTo/2` `_respondsTo(verb, arity)` returns a Boolean value indicating whether this object will respond to a message with the given verb and arity. The default implementation indicates whether the object's source code listed a method with the given verb and arity.

> **Warning:** Determining whether a given object responds to a given message is undecidable. Therefore, there are times when `_respondsTo/2` is unavoidably wrong, both with false positives and false negatives.

`_sealedDispatch/1` `_sealedDispatch` permits this object to discriminate its responses to messages based on the capabilities of the calling object.

Occasionally, a calling object will wish to prove its capabilities by passing some sort of key or token to a receiving object. The receiving object may then examine the key, and return an object based on the identity or value of the key.

We provide `_sealedDispatch/1` for a specific subset of these cases. The caller should pass a brand, and the receiver dispatches on the brand, returning either a sealed box guarded by the passed-in brand, or `null` if the brand wasn't recognized.

By default, `_sealedDispatch` returns `null`. This makes it impossible to determine whether an object actually has a customized `_sealedDispatch`.

A popular analogy for sealed dispatch is the story of the "Red Phone," a direct line of communication between certain governments in the past. The Red Phone doesn't ring often, but when it does, you generally know

who's calling. They'll identify themselves, and if you can confirm that it's the correct caller, then you can have discussions with them that you wouldn't have over an ordinary phone.

`_uncall/0` `_uncall` undoes the call that created this object. The default implementation returns `null`, because objects are, by default, not uncallable. A good implementation of `_uncall` will return a list containing `[maker, verb :Str, args :List, namedArgs :Map]` such that `M.call(maker, verb, args, namedArgs)` will produce a new object which is equal to this object. Promises or other far references may not be returned. (*No, you misunderstood; why doesn't Monte have only eventual sends?*)

Providing an instance of `_uncall` makes an object eligible for uncall-based catamorphisms (fold, reduce, ...). In particular, uncallable objects are comparable by value using `Transparent`.

Note: In order to be eligible for value comparisons, you'll need to both implement `_uncall` and also pass an audition proving that your uncall is correct. See `Selfless` and `Transparent` for details.

`_whenMoreResolved/1` `_whenMoreResolved`, by default, does nothing on near objects and sends notifications of partial fulfillment through references. It is not interesting.

Named Arguments

`FAIL` `FAIL` is an object which can be used in place of `throw.eject` when an error should propagate beyond the current turn. During asynchronous callbacks, objects might unwittingly be called as part of a subsequent turn's callback, and their errors should propagate to their original callers. `FAIL` is `throw.eject` in synchronous contexts and a wrapper for some resolver's `.smash/1` in callbacks or other asynchronous contexts.

Loops and the Iteration Protocol

Monte has only two kinds of looping constructs: `for` loops, which consume iterators to process a series of elements, and `while` loops, which repeatedly consider a predicate before doing work. Both should be familiar to any experienced programmer; let's explore them in greater detail.

`for` loops

A `for` loop is a simple structure that takes an iterable object and loops over it:

```
var x := 0
for i in (1..10):
    x += i
```

Here, we can clearly see the three elements of the `for` loop, the *pattern*, `x`; the *iterable*, `1..10`, and the loop's *body*, `x += i`. For each element in the iterable, the iterable is matched against the pattern, which is available within the body.

Within a `for` loop, the `continue` keyword will skip the current iteration of the loop, and `break` keyword will exit the loop altogether:

```
# Skip the even elements, and give up if we find multiples of three.
for i in (1..10):
    if (i % 2 == 0):
        continue
    if (i % 3 == 0):
        break
    x -= i
```

Pair Patterns

All iterables yield not just one element, but a *pair* of elements on every iteration. To access both elements at once, we can use a *pair pattern*:

```
def names := ["Scooby", "Shaggy", "Velma"]
for i => name in (names):
    traceln(`Name $i: $name`)
```

For a list, like in the previous example, the right-hand side of the pair matches the current element, and the left-hand side matches that element's index. When iterating over a map, the pair will match the key and value:

```
def animals := [
    "Bagira"     => "panther",
    "Baloo"      => "bear",
    "Shere Khan" => "tiger",
]
for animal => species in (animals):
    traceln(`Animal $animal is a $species`)
```

`while` loops

In addition to the `for` loop, Monte provides a `while` loop:

```
var x := 1
while (x < 402):
    x *= 2
```

The `while` loop admits `continue` and `break`, just like in `for` loops.

Advanced Looping

The Secret Lives of Flow Control Structures

Flow control structures actually return values. For example, the if-else returns the last value in the executed clause:

```
def a := 3
def b := 4
def max := if (a > b) {a} else {b}
```

This behavior is most useful when used with the when-catch construct described in the *When-expressions and Delayed Actions* section. The break statement, when used in a for or a while loop, can be followed by an expression, in which case the loop returns the value of that expression.

ternary conditional expression

While monte does not have the `c ? x : y` ternary conditional operator, the `if` expression works just as well. For example, to tests whether `i` is even:

```
>>> { def c := 'c'; if (c < 'e') { "Yay!" } else { "Nope" } }
"Yay!"
```

Loops as Expressions

Like all structures in Monte, `for` loops are expressions; they return values:

```
def result := for value in (0..10) { value }
```

Here, `result` is `null`, which is the default return value for `for` loops. To override that value, use `break`:

```
def result := for value in (0..10) { break value }
```

Since `break` was used, the loop exits on its first iteration, returning `value`, which was `0`. So `result` is `0`.

List & Map Comprehensions

`for` loops aren't the only way to consume iterable objects. Monte also has **comprehensions**, which generate new collections from iterables:

```
[for value in (iterable) transform(value)]
```

This will build and return a list. Maps can also be built with pair syntax:

```
[for key in (keyList) key => makeValue(key)]
```

And, of course, pair syntax can be used for both the pattern and expression in a comprehension:

```
[for key => value in (reverseMap) value => key]
```

Additionally, just like in Python and Haskell, comprehensions support filtering with a predicate; this is called the *for-such* comprehension:

```
>>> def evens := [for number in (1..10) ? (number % 2 == 0) number]
... evens
[2, 4, 6, 8, 10]
```

Just like the *such-that pattern*, this such-that clause is evaluated for every iteration, and iterations where the clause returns `false` are skipped. Also, just like the such-that pattern, and unlike some other languages' comprehension syntax, the predicate must return a `Bool`; if it doesn't, then the entire comprehension will fail with an exception.

Writing Your Own Iterables

Monte has an iteration protocol which defines iterable and iterator objects. By implementing this protocol, it is possible for user-created objects to be used in `for` loops and comprehensions.

Iterables need to have `to _makeIterator()`, which returns an iterator. Iterators need to have `to next(ej)`, which takes an ejector and either returns a list of `[key, value]` or fires the ejector with any value to end iteration. Guards do not matter but can be helpful for clarity.

As an example, let's look at an iterable that counts upward from zero to infinity:

```
object countingIterable:
    to _makeIterator():
        var i := 0
        return object counter:
            to next(_):
                def rv := [i, i]
                i += 1
                return rv
```

Since the iterators ignore their ejectors, iteration will never terminate.

For another example, let's look at an iterator that wraps another iterator and only lets even values through:

```
def onlyEvens(iterator):
    return object evens:
        to next(ej):
            var rv := iterator.next(ej)
            while (rv[1] % 2 != 0):
                rv := iterator.next(ej)
            return rv
```

Note that the ejector is threaded through `to next(ej)` into the inner iterator in order to allow iteration to terminate if/when the inner iterator becomes exhausted.

Guard Protocol

Like many other subsystems in Monte, guards can be made from any ordinary object which implements the correct methods.

Are Guards Slow?

Since guards are Monte objects and can be user-defined, concerns about performance are reasonable.
According to Semantics of Monte, every assignment acts *as if* its guard were executed; that is: once for every `def`, at definition, and for `var`, once at definition and once for every re-assignment.
But if an implementation can determine statically that the specimen will always pass (e.g. `def x :Int := 1`) then the check can be optimized away. An ahead-of-time compiler might use type inference to prove that all specimens at a definition site might be of a certain type. A just-in-time compiler might recognize at runtime that a guard's code is redundant with unboxing, and elide both the unboxing and the guard.
The Typhon virtual machine almost always can skip typical basic guards like `Int` and `Bool`.

The Basics

The main method for a guard is `coerce/2`, which takes an object to examine, called the **specimen**, and an ejector. If the specimen conforms to the guard, then the guard returns the conformed value; otherwise, the ejector is used to abort the computation.

```
object Any:
    to coerce(specimen, _):
        return specimen


object Void:
    to coerce(_, _):
        return null
```

Here are two example guards, `Any` and `Void`. `Any` passes all specimens through as-is, and `Void` ignores the specimen entirely, always returning `null`.

Here's an actual test. The `Empty` guard checks its specimen, which is a container, for emptiness and ejects on failure:

```
object Empty:
    to coerce(specimen, ej):
```

```
        if (specimen.size() != 0):
            throw.eject(ej, `$specimen was not empty`)
```

The ejector does not need to have a meaningful object (nor even a string) as its payload, but the payload may be used for diagnostic purposes by the runtime. For example, a debugger might display them to a developer, or a debugging feature of the runtime might record them to a log.

Unretractable Guards

Informally, an unretractable guard cannot be fooled by impostor objects that only pretend to be guarded, and it also will not change its mind about an object on two different coercions.

Formally, an *unretractable* guard *Un* is a guard such that for all Monte objects *o*, if *o* is successfully coerced by *Un*, then it will always be successfully coerced by *Un*, regardless of the internal state of *Un* or *o*.

Slots

Monte's values are stored in *slots*, which are also values. This nested structure permits some flexibility.

The slot of a value is accessed using the `&` unary operator:

```
def slot := &value
```

Final Slots

Final slots are created by final definitions:

```
def finalValue := 42
def finalSlot := &finalValue
```

Lazy Slots

Lazy slots are a convenient and elegant tool in the safe scope for creating simple lazy values. A lazy slot is constructed with a thunk which will be transparently evaluated once (and only once) to compute the slot's value.

```
def fib(i :Int) :Int:
    return if (i > 1) {fib(i - 1) + fib(i - 2)} else {i}
def &lazySlot := makeLazySlot(fn {fib(30)}) # or fib(40) for more drama
traceln(`$lazySlot`) # this will take a few moments
traceln(`$lazySlot`) # but this will be instantaneous
```

Note: Lazy slots can be constructed with a var slot, and it can be an enlightening exercise. `makeLazySlot` is provided as a courtesy since it acts like a final slot for auditions with DeepFrozen.

Var Slots

Var slots are created by var definitions:

```
var varValue := 7
def varSlot := &varValue
```

A var slot's value can be assigned to, and the slot's identity will not change:

```
varValue := 5
varSlot == &varValue # Still true after assignment
```

Auditors

The auditor subsystem allows objects to certify themselves as having certain properties. In order to gain certification, specimen objects must pass **audition**, a process in which the source code of the specimen object is revealed to an **auditor**, another object which examines the structure of the specimen and indicates whether it qualifies.

Stamps

Some auditors will admit any object which requests an audition. These auditors are called **stamps**. An object with a stamp is advertising behavior that is not necessarily reflected in the object's structure. Stamps can be used to indicate that an object should be preferentially treated; additionally, a stamp with limited availability can be used to indicate that an object belongs to a privileged set of objects.

A Showing of Common Auditors

DeepFrozen

The `DeepFrozen` auditor proves that objects are immutable and that the objects they refer to are also `DeepFrozen`.

```
> DeepFrozen
DeepFrozen
```

Note: The specific property proven by `DeepFrozen`: For any `DeepFrozen` object, all bindings referenced by the object are also `DeepFrozen`.

Selfless

The `Selfless` auditor is a stamp. Any object bearing `Selfless` can also bear other stamps to indicate that equality comparisons with that object should be done in a customized way.

```
> Selfless
Selfless
```

Transparent

The `Transparent` auditor proves that an object implements a custom `_uncall/0` Miranda method with certain properties. Any `Transparent` object can be compared by comparing the contents of its uncalled representation.

To prove an object `Transparent`, a small kit of facet objects must be obtained and attached to the maker definition:

```
def [makerAuditor :DeepFrozen, &&valueAuditor, &&serializer] := Transparent.makeAuditorKit()
```

Then the maker and object must both submit to audition. The maker must be `DeepFrozen` and the inner object `Selfless`:

```
def makeSwatch(color) as DeepFrozen implements makerAuditor:
    return object swatch implements Selfless, valueAuditor:
        to _uncall():
            return serializer(makeSwatch, [color])
```

The resulting maker will produce objects that can be compared as if by value:

```
> def red := makeSwatch("red")
> def xunre := makeSwatch("red")
> red == xunre
Result: true
> def blue := makeSwatch("blue")
> red == blue
Result: false
```

Note: Using the `Transparent` auditor as a guard is legal and works as expected, but is not required to obtain correct comparison behavior.

Note: Specifically, the property proven by `Transparent` is that uncalling the object is the inverse of calling the maker, and vice versa.

Bindings (WIP)

Todo

discuss bindings. Expand this section to "slots and bindings"? or discuss bindings under auditors?

Semantics of Monte

This is a brief specification of the evaluation semantics of Monte.

Monte is an object-based expression language which computes by delivering *messages* to *objects*. During computation, expressions are evaulated, resulting in either success or failure; successful evaluation yields an object, while failing evaluation yields an exceptional state.

Kernel-Monte

The Monte language as seen by the programmer has the rich set of syntactic conveniences expected of a modern scripting language. However, to be secure, Monte must have a simple analyzable semantics. We reconcile these by defining a subset of the full language called *Kernel-Monte*, and only this subset need be given a rigorous semantics. The rest of Monte is defined by syntactic expansion to this subset.

Full-Monte

We define *Full-Monte* as the complete AST of Monte, and *canonical expansion* as the syntactic expansion which expands Full-Monte to Kernel-Monte while preserving the intended semantics.

Note: Full-Monte should get its own page and have all of its rich semantics spelled out in gory detail.

Monte as a Tree

Left-to-Right Rule

The *left-to-right* rule states that *evaluation proceeds lexically from left to right.* This rule is violated only rarely:

- At the kernel level, *DefExpr* evaluates both its RHS and exit before any expressions buried in the LHS pattern. Canonical expansion from Full-Monte to Kernel-Monte resolves any recursively-defined names in order to make this less unintuitive.
- Object literals have their auditors evaluated before object creation and their patterns are unified after object creation.

Kernel-Monte is specified as an AST (abstract syntax tree). Each node in the tree is either an *expression* or a *pattern*. Expressions can be evaluated to product an object; patterns do not produce values but *unify* with values (i.e. objects) to introduce names into scopes.

Along with every node, there is a *static scope*, a compile-time constant mapping of names to declaration and usage sites. For every expression, it is known which names are visible and whether they were declared with *def* or *var*.

Computation proceeds by tree evaluation; the root of the tree is evaluated, which in turn can provoke evaluation of various branch and leaf nodes as required.

Recursion in a Monte AST is possible via self-reference; all object patterns are visible within their corresponding script's scope.

Scope Introduction & Dismissal

No Stale Stack Frames Rule

The *no stale stack frames* rule states that *A Monte expression must dismiss any scope which it introduces.*
A stale stack frame is one that isn't currently running; it is neither the current stack frame nor below the current stack frame.
Monte forbids suspending computation mid-frame. There are no coroutines or undelimited continuations in Monte. Monte also does not have an "async/await" syntax, since there is no way to implement this syntax without stale stack frames. As a direct result, no partial execution can ever require a Monte implementation to reify stack frames for suspended computation.
The policy is justified by readability concerns. Since Monte permits mutable state, one author's code's behavior could be affected by another author's code running further up the frame stack. Stale frames make comprehension of code much harder as a result.

Many expressions, during evaluation, introduce scopes. When this is done, names declared after scope introduction are said to be *visible* within the scope. An expression must pair every scope introduction with a scope dismissal. After a scope has been dismissed, the names declared within the scope are no longer visible.

Note: This scoping rule is often called "lexical scoping" and should be familiar to users of other lexically-scoped languages.

Names: Nouns, Slots, and References

Monte has a complex system underlying names.

A *noun* is an identifier which refers to a value (an object). There are three senses of reference from nouns to values, each at a different level of abstraction.

At the simplest level, nouns refer directly to values. Identifiers in patterns match values, and nouns in expressions evaluate to the values to which they were matched.

To represent mutable state, we indirect via slots. *Slots* are objects that contain values and may be updated over time (much like pointers in C). Slots can be accessed and manipulated with slot patterns and slot expressions. A final slot acts as though nouns refer directly to values, while a var slot has a `put` operation that updates its value.

A *binding* is a slot along with a guard that constrains the values in the slot. Bindings are essential to *auditors*.

To allow references across turns and vats, we indirect via references.

Exceptions

A Monte expression can yield either a successful result or an exceptional state. Exceptional states are intentionally vague; they are usually represented as panics in virtual machines or stack unwinders in interpreters.

While in an exceptional state, most expressions evaluate to that same exceptional state. A *TryExpr* can replace an exceptional state with a successful result. A *FinallyExpr* can perform some side computation despite an exceptional state.

When an error is thrown, the computation switches to an exceptional state and the thrown error is sealed in an implementation-dependent manner.

Expressions

Literals

Null

Produces `null`.

Char

Produces an object which passes `Char` and corresponds to the Unicode codepoint of the *CharExpr*.

Double

Produces an object which passes `Double` and corresponds to the IEEE 754 double-precision floating-point number of the *DoubleExpr*.

Note: Implementations may, at their discretion, substitute any higher-precision IEEE 754 number for the given one.

Int

Produces an object which passes `Int` and corresponds to the integer of the *IntExpr*.

Str

Produces an object which passes `Str` and corresponds to the sequence of Unicode codepoints of the *StrExpr*. .

The string of codepoints is not normalized; it corresponds one-to-one with the codepoints in the Monte source literal.

Names

Binding

Produces the binding for the given noun.

Todo

discuss SlotExpr

Noun

Produces the value in the slot of the given noun.

Assign

An *AssignExpr* has a name and an expression. The expression is evaluated and the result is both assigned to the name as a noun in the current scope and the produced value.

If the name's slot is not assignable, an error is thrown.

Def

A *DefExpr* has a pattern, an (optional) exit expression, and a specimen expression. The specimen is evaluated, followed by the exit (if present). The specimen is unified with the pattern, defining names into the surrounding scope. The produced value is the specimen.

If unification fails, the result of the exit expression is used as an ejector to escape; if ejecting fails, then an error is thrown.

Hide

A *HideExpr* has a single subexpression which is evaluated in a fresh scope. The produced value of the subexpression is used as the produced value.

Message Passing

Call

A *CallExpr* has a receiver expression, a *verb* (string), some argument expressions, and some named argument expressions. The receiver is evaluated, then each argument, and then each named argument. Then, a *message* consisting of the verb, arguments, and named arguments is passed to the receiver. The value returned from the receiver is the produced value.

Todo

discuss sameness and doctest *_equalizer*

Control Flow

Escape

Ejectors

An ejector is an object whose `run` method aborts the current computation and returns to where the ejector was created.
Monte implements the `return`, `break`, and `continue` expressions with ejectors.
Ejectors are so-called *single-use, delimited continuations*: their dynamic scope is delimited to downward method calls only, and any use after the first will fail.

An *EscapeExpr* has a pattern and inner expression and, optionally, a catch pattern and catch expression (not to be confused with *Try*/catch expressions).

An ejector is created and a scope is introduced. The ejector is unified with the pattern and then the inner expression is evaluated.

If the ejector was not called during evaluation of the inner expression, the scope is then dismissed and the produced value from the inner expression is used as the produced value of the entire *EscapeExpr*.

If the ejector is called within the inner expression, then control immediately leaves the inner expression and the scope is dismissed; if there is no catch pattern/expression, then the value passed to the ejector is immediately used as the produced value. Otherwise, the value passed to the ejector is used as a specimen and unified with the catch pattern in a freshly-introduced scope, and then the catch expression is evaluated. Finally, the catch scope is dismissed and the produced value from the catch expression is used as the produced value of the escape-expr.

Finally

A *FinallyExpr* contain two expressions. The first expression is evaluated in a fresh scope and its resulting object or failing state is retained. Then, the second expression is evaluated in a fresh scope. Finally, the retained state from the first expression, success or failure, is the produced value of the entire finally-expr.

The second expression is evaluated regardless of whether the first expression returns an exceptional state; its state is discarded. It is implementation-dependent whether exceptional states are chained together.

Chained Exceptions

Why doesn't Monte require chained exceptions? In many languages, the exception from the first part of a finally-expr would have a chain including the exception from the second part of the finally-expr. This faciliates debugging.
Since Monte doesn't offer tools for digging into exceptional states beyond catching them as a reified but opaque value, there is little point in mandating implementation details for that value. Instead, one might expect unsafe names like *unsealException* to have standard behavior, and that behavior might include exposing a possibly-empty list of chained exceptions. This isn't currently the case, but it might be in the future.

This table shows the possible states:

try	*finally*	result
success	success	success
error	success	error
success	error	error
error	error	error

If

An *IfExpr* has a test expression, a consequent expression, and an alternative expression. A scope is introduced, and then the test expression is evaluated, producing a value which passes `Bool`. Either the consequent or the alternative is evaluated and used as the produced value, depending on whether the test produced `true` or `false`. Finally, the scope is dismissed.

If the test's produced value does not conform to `Bool`, an error is thrown.

Sequence

A *SequenceExpr* contains zero or more expressions.

If a *SequenceExpr* contains zero expressions, then it evaluates to *null*.

Otherwise, a *SequenceExpr* evaluates each of its inner expressions in sequential order, using the final expression's produced value as the produced value of the entire sequence.

Try

A *TryExpr* has an expression and a catch pattern and expression. The first expression is evaluated in a fresh scope and used as the produced value.

If an error is thrown in the first expression, then the scope is dismissed, a new scope is introduced, the error is unified with the catch pattern, and the catch expression is evaluated and used as the produced value.

Objects

Evaluation of a message sent to an object proceeds as follows.

Matcher

A matcher has a pattern and an expression. A scope is introduced and incoming messages are unified with the pattern. If the unification succeeds, the expression is evaluated and its produced value is returned to the caller.

Method

A method has a verb, a list of argument patterns, a list of named argument patterns, a guard expression, and a body expression. When a message matches the verb of the method, a scope is introduced and each pattern is unified against the message. Each argument pattern is unified against each argument, and then each named argument pattern is unified against each named argument.

If the number of arguments in the message differs from the number of argument patterns in the method, an error is thrown. Informally, the method and message must have the same arity.

If unification fails, an error is thrown.

After unification, the guard expression is evaluated and its produced value is stored for return value guarding. The body expression is evaluated and its produced value is given as a specimen to the return value guard. The returned prize from the guard is returned to the caller.

If the return value guard fails, an error is thrown.

Note: The return value guard is evaluated before the body, but called after the body.

Object

An *ObjectExpr* has a pattern, a list of auditor expressions, a list of methods, and a list of matchers. When evaluated, a new object with the methods and matchers is created. That object is audited by each auditor in sequential order. Finally, the object is unified with its pattern in the surrounding scope, and the first auditor, if present, is used as the guard for the binding.

Objects close over all of the names which are visible in their scope. Additionally, objects close over the names defined in the pattern of the *ObjectExpr*.

Patterns

Pattern evaluation is a process of *unification*. During unification, patterns are given a specimen and an ejector. Patterns examine the specimens and create names in the surrounding scope. When patterns fail to unify, the ejector is fired. If the ejector fails to leave control, then an error is thrown.

Pattern Nodes

Ignore

An *IgnorePatt* coerces its specimen with a guard.

Binding

A *BindingPatt* coerces its specimen with the `Binding` guard and binds the resulting prize as a binding.

Final

A *FinalPatt* coerces its specimen with a guard and binds the resulting prize into a final slot.

Var

A *VarPatt* coerces its specimen with a guard and binds the resulting prize into a var slot.

List

A *ListPatt* has a list of subpatterns. It coerces its specimen to a `List` and matches the elements of the specimen to each subpattern, in sequential order.

If the *ListPatt* and specimen are different lengths, then unification fails.

Via

A *ViaPatt* contains an expression and a subpattern. The specimen and ejector are passed to the expression's produced value, and the result is unified with the subpattern.

CHAPTER 10

Appendixes, Indices and Tables

Monte Grammar

Note: Lexical details such as indented blocks are not captured in this grammar.

Todo

finish grammar productions marked @@. Meanwhile, see monte_parser.mt for details.

```
blockExpr           ::=   FunctionExpr
                        | ObjectExpr
                        | bind
                        | def
                        | InterfaceExpr
                        | IfExpr
                        | ForExpr
                        | WhileExpr
                        | SwitchExpr
                        | EscapeExpr
                        | TryExpr
                        | WhenExpr
                        | LambdaExpr
                        | MetaExpr
block               ::=   "{" (sequence | "pass" ) "}"
HideExpr            ::=   "{" ((expr ";" )+ | /* empty */) "}"
IfExpr              ::=   "if" "(" expr ")" block [ "else" ( "if" /* blockExpr@@ */ | bloc
SwitchExpr          ::=   "switch" "(" expr ")" "{" matchers "}"
matchers            ::=   ( "match" pattern block )+
TryExpr             ::=   "try" block catchers
catchers            ::=   [( "catch" pattern block )+ ] [ "finally" block ]
EscapeExpr          ::=   "escape" pattern blockCatch
WhileExpr           ::=   "while" "(" expr ")" blockCatch
ForExpr             ::=   "for" pattern [ "=>" pattern ] "in" comp blockCatch
blockCatch          ::=   block [ "catch" pattern block ]
WhenExpr            ::=   "when" "(" (expr "," )+ ")" "->" block catchers
```

```
LambdaExpr          ::=  "fn" [(pattern "," )+ ] block
def                 ::=  "def" ( ( "bind" name [guard ] | name ) (/* objectFunction@@ */
bind                ::=  "bind" name [guard ] objectExpr
ObjectExpr          ::=  "object" ( "bind" name | "_" | name ) objectExpr
objectExpr          ::=  [ "extends" order ] auditors "{" [(objectScript ";" )+ ] "}"
objectScript        ::=  [doco ] ("pass" | [("@@meth" )+ ] ) ("pass" | [(matchers )+ ] )
matchers            ::=  ( "match" pattern block )+
doco                ::=  .String.
FunctionExpr        ::=  "def" [ "." verb ] "(" [(pattern "," )+ ] ")" block
InterfaceExpr       ::=  "interface" namePatt [ "guards" pattern ] [ "extends" (order ","
guardOpt            ::=  ":" guard
                         | /* empty */
guard               ::=  IDENTIFIER "[" ((expr "," )+ | /* empty */) "]"
                         | IDENTIFIER
                         | "(" expr ")"
module_header       ::=  "imports" StrExpr "=~" ((pattern )+ ) [exports ] sequence
exports             ::=  "exports" "(" ((name "," )+ | /* empty */) ")"
sequence            ::=  ((blockExpr | expr ) ";" )+
                         | /* empty */
assign              ::=  "def" pattern [ "exit" order ] ":=" assign
                         | (VarPatt | BindPatt ) /* empty */ ":=" assign
                         | lval ":=" assign
                         | VerbAssignExpr
                         | order
lval                ::=  order "[" ((expr "," )+ | /* empty */) "]"
                         | name
VerbAssignExpr      ::=  lval VERB_ASSIGN assign
logical_or          ::=  logical_and [ "||" logical_or ]
logical_and         ::=  comp [ "&&" logical_and ]
comp                ::=  order (("=~" | "!~" ) | ("==" | "!=" ) | "&!"  | ("^" | "&" | "|"
                         | order
order               ::=  CompareExpr
                         | RangeExpr
                         | BinaryExpr
                         | prefix
CompareExpr         ::=  prefix (">" | "<" | ">=" | "<=" | "<=>" ) order
RangeExpr           ::=  prefix (..  | ..!)  order
shift               ::=  prefix ("<<" | ">>" ) order
additiveExpr        ::=  multiplicativeExpr ("+" | "-" ) additiveExpr
multiplicativeExpr  ::=  exponentiationExpr ("*" | "/" | "//" | "%" ) order
exponentiationExpr  ::=  prefix "**" order
prefix              ::=  "-" prim
                         | ("~" | "!"  )  calls
                         | SlotExpr
                         | BindingExpr
                         | CoerceExpr
                         | calls
SlotExpr            ::=  "&" name
BindingExpr         ::=  "&&" name
MetaExpr            ::=  "meta" .  ( "context" "(" ")" | "getState" "(" ")" )
CoerceExpr          ::=  calls ":"  guard
calls               ::=  prim ((( ( call | send ) | index ) )+ ) [curryTail ]
call                ::=  [ .  verb ] argList
send                ::=  "<-" [verb ] argList
```

```
curryTail           ::=   .  verb
                          | "<-" verb
index               ::=   "[" ((expr "," )+ | /* empty */) "]"
verb                ::=   IDENTIFIER
                          | .String.
argList             ::=   "(" ((expr "," )+ | /* empty */) ")"
pattern             ::=   postfixPatt
postfixPatt         ::=   SuchThatPatt
                          | prefixPatt
prefixPatt          ::=   MapPatt
                          | ListPatt
                          | SamePatt
                          | NotSamePatt
                          | QuasiliteralPatt
                          | ViaPatt
                          | IgnorePatt
                          | namePatt
namePatt            ::=   FinalPatt
                          | VarPatt
                          | BindPatt
                          | SlotPatt
                          | BindingPatt
SuchThatPatt        ::=   prefixPatt "?"  "(" expr ")"
ListPatt            ::=   "[" ((pattern "," )+ | /* empty */) "]" [ "+" pattern ]
MapPatt             ::=   "[" (mapPattItem "," )+ "]" [ "|" pattern ]
mapPattItem         ::=   ( (LiteralExpr | "(" expr ")" ) "=>" pattern | "=>" namePatt ) [
SamePatt            ::=   "==" prim
NotSamePatt         ::=   "!=" prim
QuasiliteralPatt    ::=   [IDENTIFIER] "`" ((( QUASI_TEXT | ( AT_IDENT | "@{" pattern "}" )
ViaPatt             ::=   "via" "(" expr ")" pattern
FinalPatt           ::=   name guardOpt
VarPatt             ::=   "var" name guardOpt
BindPatt            ::=   "bind" name guardOpt
SlotPatt            ::=   "&" name guardOpt
BindingPatt         ::=   "&&" name
IgnorePatt          ::=   "_" guardOpt
prim                ::=   "(" expr ")"
                          | LiteralExpr
                          | quasiliteral
                          | NounExpr
                          | HideExpr
                          | MapComprehensionExpr
                          | ListComprehensionExpr
                          | ListExpr
                          | MapExpr
expr                ::=   assign
                          | ("continue" | "break" | "return" ) ( "(" ")" | ";" | blockExpr
NounExpr            ::=   name
name                ::=   IDENTIFIER
                          | "::"  stringLiteral
LiteralExpr         ::=   StrExpr
                          | IntExpr
                          | DoubleExpr
                          | CharExpr
```

```
quasiliteral        ::=   [IDENTIFIER] "`" ((( QUASI_TEXT | ( DOLLAR_IDENT | "${" expr "}"
ListExpr            ::=   "[" ((expr "," )+ | /* empty */) "]"
comprehension       ::=   pattern "in" iter expr
                          | pattern "=>" pattern "in" iter expr "=>" expr
iter                ::=   order [ "if" comp ]
MapExpr             ::=   "[" (mapItem "," )+ "]"
mapItem             ::=   expr "=>" expr
                          | "=>" (SlotExpr | BindingExpr | NounExpr )
IntExpr             ::=   (hexLiteral | decLiteral )
decLiteral          ::=   digits
digits              ::=   digit (((digit | "_" ) )+ )+
digit               ::=   /* one of:  0, 1, 2, 3, 4, 5, 6, 7, 8, 9 */
hexLiteral          ::=   "0" ("x" | X) hexDigits
hexDigits           ::=   hexDigit (((hexDigit | "_" ) )+ )+
hexDigit            ::=   /* one of:  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B,
DoubleExpr          ::=   floatLiteral
floatLiteral        ::=   digits ( .  digits [floatExpn ] | floatExpn )
floatExpn           ::=   /* one of:  e, E */ [ /* one of:  -, + */ ] digits
CharExpr            ::=   "'" charConstant "'"
charConstant        ::=   (( "\" /* newline */ )+ )+ (/* none of:  ', \, tab */ | "\" ( ( U
StrExpr             ::=   stringLiteral
stringLiteral       ::=   '"' ((charConstant )+ )+ '"'
```

Roadmap: Montefesto

> .ia lo snura faircu'u kanji ka'e na'e nandu ("Secure distributed computation should not be hard.")
>
> —Corbin, on Monte

This is the roadmap for Monte development according to Allen and Corbin. If you want to work on anything on this list, let us know; we're very accepting of new contributors.

2015

- Finish key language features
  - Named arguments
  - m''
  - Bytes
  - Finalize on-disk (on-wire) compiled code format
  - Auditors
- Finish key runtime features
  - Expose key C libraries to user-level code
    - libsodium
    - libuv
- Finish key compiler features
  - Compiler error messages are informative

- Finish key integration features
  - Profiling
    - Time (vmprof)

2016

- "Exit stealth mode"; display a sleek and friendly front page to neophytes and visitors which explains:
  - Why Monte exists
  - How to get started using Monte
- Have stories for:
  - Developing modular Monte codebases
- Finish key language features
  - Streamcaps
  - Vats
- Finish key integration features
  - Initial IDE support
    - vim (Corbin)
    - emacs (Allen)
    - Sublime/Atom (Mike, Justin)

2017

- Make Monte desireable
- Improve the core
  - Speed: Nobody should have to wait for code to compile
  - Safe objects
    - Many method improvements to builtin collections
    - Semitransparent
    - PassByCopy
    - makeWeakMap
    - Twines
    - Elusive Eight: Useful numerical analysis methods for doubles
  - Unsafe objects
    - Filesystem
    - Timers
  - Typhon-specific improvements
    - Even faster interpreting
- Develop Monte packaging

  - Packages
  - Muffins
  - Environments
  - mtpkgs
- Develop important libraries
  - HTTP
  - Debugger
  - Records
  - Pretty-printers
- Monte-related R&D
  - Rationals
  - Capn Proto
  - CapTP/VatTP

2018

We currently don't know what we're going to do for 2018. Possibilities range from MonteCon to The Monte Foundation to nothing at all. Who knows? It is a mystery~

Contributing

If you'd like to get involved with developing or using the Monte language, start by getting in touch with us on IRC. It is useful, but not necessary, to be acquainted with Python's syntax and/or the computational concepts of E.

Then clone the repo and follow the directions below to begin running Monte code. If you have problems, join us in #monte on irc.freenode.net, ask your question (use a pastebin to share any errors, rather than pasting into the channel), and wait a few hours if nobody is around.

If you'd like to contribute to Monte, check out the Monte and Typhon issue trackers. It's also worth grepping for `TODO` in the source of both projects.

Colophon: Monte Documentation Build Tools

Restructured text

The docs are written in restructured text.

Sphinx

The docs are built with Sphinx and hosted on readthedocs.

The virtualenv for building the docs is separate from the main Monte virtualenv. Create a separate virtualenv and `pip install -r docs_requirements.txt`, then `make html` to make the docs. Locally built docs will show up in the docs/build directory.

Syntax Railroad Diagrams and Haskell Parser

`rr_ext.py` is an extension that integrates the railroad-diagrams library by Tab Atkins into the build process. It provides a custom `.. syntax::` directive.

If *syntax_dest* is set in *conf.py*, the syntax diagram info is written to a file in JSON format. download:*rr_grammar.py* converts this format to a sphinx grammar production display.

download:*rr_happy.py* is work-in-progress to generate a haskell monadic parser.

Doctests

Use *make doctest* to extract the *source/docs_examples.mt* test suite from the documentation. Then run it a la *typhon loader test docs_examples*.

TODO List

Todo

discuss bindings. Expand this section to "slots and bindings"? or discuss bindings under auditors?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/auditors.rst, line 101.)

Todo

expansion of various forms of `try`

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 147.)

Todo

while doctests, expansion

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 176.)

Todo

for doctests, expansion

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 204.)

Todo

doctest `/** docstring */`

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 235.)

Todo

interface syntax diagram @@s

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 339.)

Todo

various items marked "@@" in railroad diagrams. Also, finish re-organizing them around precedence (use haskell codegen to test).

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/block-expr.rst, line 346.)

Todo

finish grammar productions marked @@. Meanwhile, see monte_parser.mt for details.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/grammar.rst, line 8.)

Todo

When new packaging efforts are ready, update this to mention that module namespaces are either the stdlib or a package name.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/modules.rst, line 55.)

Todo

special operator rules because of security

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/operators.rst, line 30.)

Todo

VERB_ASSIGN lexical details

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/operators.rst, line 128.)

Todo

discuss, doctest SlotExpression `&x`, BindingExpression `&&x`

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/operators.rst, line 490.)

Todo

named args in argList

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/operators.rst, line 567.)

Todo

discuss matchers in object expressions

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/operators.rst, line 584.)

Todo

document docstrings

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/ordinary-programming.rst, line 59.)

Todo

document named args, defaults

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/ordinary-programming.rst, line 61.)

Todo

show: Guards play a key role in protecting security properties.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/ordinary-programming.rst, line 268.)

Todo

discuss SlotExpr

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/semantics.rst, line 208.)

Todo

discuss sameness and doctest *_equalizer*

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/semantics.rst, line 256.)

Todo

specify canStartIndentedBlock, braceStack exactly

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/symbols.rst, line 26.)

Todo

Document how to compile and run such a script.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/monte/checkouts/latest/docs/source/taste.rst, line 33.)

Glossary

ejector : Coercion An object which can be called once to prematurely end control flow.

guard : Coercion An object which provides the coercion protocol.

message An object of the form `[verb :Str, args :List, namedArgs :Map]` which is passed from calling objects to target objects to faciliate computation.

prize : Coercion The result of a successful coercion.

quasiliteral, QL An literal expression or pattern which is composed of both literal and variable pieces.

quasiparser, QP An object which provides the *Quasiliterals* protocol.

verb A string which forms the first element of a message.

- genindex
- modindex
- search

t